In the name of God

UvA Team Source files

Collected by:

Kamran Amini

List of files

Connection.h

```
#ifndef _CONNECTION_
#define _CONNECTION_

#include <stdio.h>     // printf, FILE
#include <iostream>    // ostream
#ifdef WIN32
 #include <windows.h>   // sockaddr_in
#else
 #include <netdb.h>     // sockaddr_in
 #include <arpa/inet.h> // sockaddr_in
#endif

using namespace std;

/*! Socket is a combination of a filedescriptor with a server adress.       */
typedef struct _socket{
   int    socketfd  ;              /*!< File descriptor of the socket.    */
   struct sockaddr_in  serv_addr ;   /*!< Server information of the socket. */
} Socket ;

/********************** CONNECTION ****************************************/

/*! This class creates creates a (socket) connection using a hostname and a
   port number. After the connection is created it is possible to send and
   receive messages from this connection. It is based on the client program
   supplied with the soccer server defined in client.c and created by
   Istuki Noda et al.*/
class Connection {

 Socket        m_sock;       /*!< communication protocol with the server. */
 int           m_iMaxMsgSize; /*!< max  message size for send and receive  */
public:

 // constructors
 Connection          (                                    );
 Connection          ( const char *hostname, int  port,   int iSize  );
 ~Connection          (                                    );

 // methods that deal with the connection itself
 bool connect         ( const char *host,     int  port             );
 void disconnect      ( void                                );
 bool isConnected     ( void                                )const;

 // methods that deal with the communication over the connection
#ifndef WIN32
 int  message_loop    ( FILE       *in,       FILE *out             );
```

```
#endif
  int  receiveMessage   ( char       *msg,      int  maxsize            );
  bool sendMessage      ( const char *msg                              );

  void show             ( ostream    os                                );

};

#endif
```

Connection.cpp

```
#include <errno.h>       // EWOULDBLOCK
#include <string.h>      // strlen
#include <sys/types.h>   // socket

#ifndef WIN32
  #include <unistd.h>      // close
  #include <sys/socket.h>  // socket
#endif

#ifdef Solaris
#include <sys/socket.h>  // AF_INET SOCK_DGRAM
#endif

#include "Connection.h"
#include "Logger.h"      // LOG
//#include<iostream>
using namespace std;
extern Logger Log; /*!< This is a reference to the Logger for writing info to*/

/*************************************************************************
/
/********************** CONNECTION
*****************************************/
/*************************************************************************
/

/*! Default constructor. Only sets the maximum message size. */
Connection::Connection( )
{
  m_iMaxMsgSize = 2048;
}

/*! Constructor makes a connection with the server using the connect method.
    \param hostname string representation of host machine (string or IP number)
    \param port port number for connection of the host machine
    \param iMaxSize maximum message size that can be sent or received */
Connection::Connection(const char *hostname, int port, int iMaxSize)
{
  m_iMaxMsgSize = iMaxSize;
  if(  connect( hostname, port ) )
    Log.log( 1,"(Connection:connection) Socket connection made with %s:%d",
                                          hostname, port );
  else
    Log.log( 1,"(Connection:Connection) Could not create connection with %s:%d"
                                        , hostname, port );
}
```

```
/*! Deconstructor closes the connection with the server */
Connection::~Connection()
{
  disconnect();
}

/*! This method sets up a connection with the server.
    \param hostname string representation of host machine (string or IP number)
    \param port port number for connection of the host machine
    \return bool indicating whether connection was made */
bool Connection::connect(const char *host, int port )
{
  struct hostent *host_ent;
  struct in_addr *addr_ptr;
  struct sockaddr_in  cli_addr ;
  int    sockfd ;

  m_sock.socketfd = -1 ;

#ifndef WIN32
  if( (host_ent = (struct hostent*)gethostbyname(host)) == NULL)
  {
    // if not a string, get information from IP adress.
  #ifdef Solaris
    if( inet_addr(host) == ((in_addr_t)-1) )
  #else
    if( inet_addr(host) == INADDR_NONE )
  #endif
    {
      cerr << "(Connection::connect) Cannot find host " << host << endl;
      return false ;
    }
  }
  else   // get the necessary information from the hostname (string)
  {
    addr_ptr = (struct in_addr *) *host_ent->h_addr_list;
    host = inet_ntoa(*addr_ptr);
  }
#else
    // winsock initialization
  WORD     wVersionRequested;
  WSADATA  wsaData;
  wVersionRequested = MAKEWORD( 2, 2 );

  if ( WSAStartup( wVersionRequested, &wsaData ) != 0 )
```

```
    return false;

  if ( LOBYTE( wsaData.wVersion ) != 2 ||
       HIBYTE( wsaData.wVersion ) != 2 ) {
      WSACleanup();
      return false;
  }

  if(inet_addr(host) == INADDR_NONE){
    if((host_ent = (struct hostent *)gethostbyname(host)) == NULL) {
        return false;
    } else {
        addr_ptr = (struct in_addr *) *host_ent->h_addr_list ;
        host = inet_ntoa(*addr_ptr) ;
    }
  }
#endif
  //  Open UDP socket.
  if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
  {
    cerr << "(Connection::connect) Cannot create socket " << host << endl;
    return false ;
  }

  // insert the information of the client
  cli_addr.sin_family      = AF_INET ;
  cli_addr.sin_addr.s_addr = htonl(INADDR_ANY) ;
  cli_addr.sin_port        = htons(0) ;

  // bind the client to the server socket
  if(bind(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0)
  {
    cerr << "(Connection::connect) Cannot bind local address " << host << endl;
    return false ;
  }

  //  Fill in the structure with the address of the server.
  m_sock.socketfd = sockfd ;

  m_sock.serv_addr.sin_family       = AF_INET ;
  m_sock.serv_addr.sin_addr.s_addr  = inet_addr(host);
  m_sock.serv_addr.sin_port         = htons(port) ;

  return true;
}
```

```
/*! This method closes the current socket connection.  */
void Connection::disconnect( void )
{
  if (isConnected() )
  {
#ifdef WIN32
    closesocket( m_sock.socketfd );
#else
    close(m_sock.socketfd) ;
#endif
    m_sock.socketfd = -1; // This also 'sets' isConnected() to false
  }
}

/*! This method determines whether the socket connection is connected.
    \return bool indicating whether socket connection is available */
bool Connection::isConnected(void) const
{
  return(m_sock.socketfd != -1);
}

/*! This method reads a message from the connection. When there is
    no message available, it blocks till it receives a message.
    \param msg string in which message is stored
    \param maxsize maximum size of the message.
    \return -1: error, 0: 0 bytes were read, 1 when read was succesfull */
int Connection::receiveMessage( char *msg, int maxsize )
{
#ifdef WIN32
  int       servlen;
#elif Solaris
  int       servlen;
#else
  socklen_t servlen ;
#endif
  int n;
  struct sockaddr_in serv_addr ;

  servlen = sizeof(serv_addr) ;

  // receive message from server
  n = recvfrom(m_sock.socketfd, msg, maxsize, 0,
                        (struct sockaddr *)&serv_addr, &servlen);

  if(n < 0)                           // error
  {
```

```
#ifdef WIN32
   if( n == -1 && errno == WSAEWOULDBLOCK)
#else
   if( n == -1 && errno == EWOULDBLOCK)
#endif
    {
     msg[0] = '\0' ;
     return 0 ;
    }
    else
     return -1;
  }
  else                                  // succesfull, set new server info
  {                                     // next message will go to there
    m_sock.serv_addr.sin_port = serv_addr.sin_port ;
    msg[n] = '\0' ;

    return ( n == 0 ) ? 0 : 1 ;
  }
}

/*! This method sends a message to the server using the current connection.
    \param msg string which contains message
    \return true on succes, false in case of failure */
bool Connection::sendMessage( const char *msg )
{
  int n;

  n = strlen(msg) + 1 ;
  if( sendto(m_sock.socketfd, msg, n, 0,
      (struct sockaddr *)&m_sock.serv_addr, sizeof(m_sock.serv_addr)) != n )
    return false ;
  return true ;
}

/*! This method always loops and waits for input. When input is received from
    fpin then this input is send to the server using the current connection.
    When message is received from the server, this message is sent to fpout.
    \param fpin file pointer from which input is read (i.e. stdin )
    \param fpout file pointer to which output should be directed (i.e. stdout)
    \return 0 but only when error occured in reading input */
#ifndef WIN32
int Connection::message_loop( FILE *fpin, FILE *fpout )
{
  fd_set readfds, readfds_bak;
  int in, max_fd, n, ret;
```

```
  char buf[m_iMaxMsgSize];

  in = fileno( fpin );
  FD_ZERO( &readfds );
  FD_SET( in, &readfds );
  readfds_bak = readfds;
  max_fd = ((in > m_sock.socketfd) ? in : m_sock.socketfd) = 1;

  while( 1 )
  {
   readfds = readfds_bak;
   // wait for message from socket or fpin
   if(( ret = select( max_fd, &readfds, NULL, NULL, NULL )) < 0 )
   {
    perror("select");
    break;
   }
   else if( ret != 0 )
   {
    if( FD_ISSET(in, &readfds))              // file descriptor fpin was set
    {
     fgets(buf, m_iMaxMsgSize, fpin);       // get the message
     if( sendMessage(buf ) == false )       // and send it
      break;
    }
    if( FD_ISSET(m_sock.socketfd, &readfds ) )// file descriptor was set
    {
     n = receiveMessage(buf, m_iMaxMsgSize); // receive the message
     if( n == -1 )
      break;
     else if( n > 0 )                       // and put it to fpout
     {
      fputs(buf, fpout);
      fputc( '\n', fpout);
     }
     fflush(stdout);
    }
   }
  }
  return 0;
}
#endif


/*! This method prints whether the connection is up or not.
    \param os output stream to which output should be directed */
```

```
void Connection::show( ostream os )
{
  if( ! isConnected() )
    os << "Not connected" << endl;
  else
    os << "Connected" << endl;

}

/********************** TESTING PURPOSES
************************************/

/*
int main( void )
{
  char strBuf[m_iMaxMsgSize];
  Connection c( "localhost", 6000 );
  int i;
  c.sendMessage( "(init l (version 6.07))" );
  while( 1 )
  {
    i = c.receiveMessage( strBuf, m_iMaxMsgSize );
    printf("%d|%s|\n", i, strBuf );
  }
  return 0;
}
*/
```

ActHandler.h

```
#ifndef _ACTHANDLER_
#define _ACTHANDLER_

#include "Connection.h"      // needed for Connection class
#include "WorldModel.h"      // needed for 'setPerformedActions'

#ifdef WIN32
  void CALLBACK sigalarmHandler(UINT id, UINT msg, DWORD dwUser, DWORD dw1,
DWORD dw2);
#else
  void sigalarmHandler( int i );
#endif
void   sendChangeViewCommands( int iSyncCounter );

extern Logger Log; /*!< Reference to the Logger to write log info to*/

/*************************************************************************
/
/********************** CLASS ACTHANDLER
************************************/
/*************************************************************************
/

/*!The ActHandler Class is used in the RoboCup Soccer environment to send the
   commands to the soccerserver. The ActHandler contains a queue in which the
   commands are put. When a signal arrives (set by the SenseHandler depending
   on the time of the sense_body message) the commands that are currently in
   the queue are converted to text strings and send to the server. The sent
   commands are also passed to the WorldModel, such that the WorldModel can
   update its internal state based on the performed actions.
   It is possible to send more than one command to the server at each time
   step, but some type of (primary) commands can only be sent once (kick,dash,
   move, tackle, turn and catch). Therefore internally different queues are
   stored. One with only one element, namely the last entered primary command.
   One with all the change_view commands (since these have to be sent at
   special times for the synchronization). And finally a separate queue
   containing all the other commands. Each time a command is put into the queue
   that is already there, the command is updated with the new information.
   Furthermore it is also possible to directly send commands (or text strings)
   to the server. These methods can be used when an initialization or move
   command has to be sent to the server and you're sure this information
   is final, i.e. the message will not become better when new information
   arrives from the server. */
class ActHandler {

  Connection     *connection;  /*!< Connection with the server             */
```

```
  ServerSettings *SS;         /*!< ServerSettings with server parameters    */
  WorldModel     *WM;         /*!< needed to set performed actions          */

  SoccerCommand  m_queueOneCycleCommand; /*!< primary command is saved here  */
  SoccerCommand  m_queueMultipleCommands[CMD_MAX_COMMANDS];
                              /*!< non primary commands*/
  int           m_iMultipleCommands;    /*!< number of non-primary commands */

public:
  ActHandler( Connection* c, WorldModel *wm, ServerSettings *ss);

  // methods related to putting and sending messages using the queue
  bool          putCommandInQueue( SoccerCommand command      );
  void          emptyQueue       (                            );
  bool          isQueueEmpty     (                            );
  bool          sendCommands     (                            );
  SoccerCommand  getPrimaryCommand(                           );

  // methods to send commands directly to the server
  bool          sendCommand      ( SoccerCommand soc          );
  bool          sendMessage      ( char          *str         );
  bool          sendCommandDirect( SoccerCommand soc          );
  bool          sendMessageDirect( char          *str         );

} ;
#endif
```

ActHandler.cpp

```
#include "ActHandler.h"

#ifndef WIN32
  #include <poll.h>     // poll
  #include <sys/poll.h> // poll
#endif
#include <signal.h>     // SIGALARM

ActHandler* ACT; /*!< Pointer to ActHandler class needed by signal handler */

/*! This function is executed when a SIGALARM singal arrives. The time this
    signal comes is defined by the SenseHandler (depending on incoming
    sense_body messages). When the signal arrives, the commands currently
    stored in the queue of the ActHandler are send to the server (using the
    method sendCommands).
    \param i is ignored */
#ifdef WIN32
extern void CALLBACK sigalarmHandler(UINT , UINT , DWORD , DWORD , DWORD )
#else
extern void sigalarmHandler( int i )
#endif
{
  Log.logFromSignal( 2, "alarm handler!!" );
  ACT->sendCommands( );
}

/*! This is the constructor for the ActHandler class. All the variables are
    initialized.
    \param c Connection that is connected with the soccerserver
    \param wm WorldModel, used to set performed commands
    \param ss ServerSettings in which server settings are defined */
ActHandler::ActHandler( Connection *c, WorldModel *wm, ServerSettings *ss )
{
  connection         = c;
  SS                 = ss;
  WM                 = wm;

  m_iMultipleCommands = 0;
  ACT                = this; // needed to let signal call method from class
}

/*! This method empties the queue in which all the commands are stored. */
void ActHandler::emptyQueue( )
{
  m_queueOneCycleCommand.commandType = CMD_ILLEGAL;
  for( int i = 0; i < CMD_MAX_COMMANDS - 1 ; i ++ )
```

```
    m_queueMultipleCommands[i].commandType = CMD_ILLEGAL;
  m_iMultipleCommands=0;
}

/*! This method returns whether the current queue contains no commands
    \return true when queue is empty, false otherwise */
bool ActHandler::isQueueEmpty()
{
  return m_queueOneCycleCommand.commandType == CMD_ILLEGAL &&
         m_iMultipleCommands                == 0;
}

/*! This method converts all commands in the queue to text strings and sends
    these text strings to the server (connected by Connection). When the server
    didn't execute the commands from the previous cycle (this information is
    stored in the WorldModel) the commands in the queue are not sent, since it
    is probably the case that these commands will be performed this cycle and
    we don't want a clash (two commands in one cycle). In this case false is
    returned.
    \return true when sending of messages succeeded, false otherwise */
bool ActHandler::sendCommands( )
{
  static Time timeLastSent = -1;
  bool        bNoOneCycle  = false;
  char        strCommand[MAX_MSG];
  strCommand[0]            = '\0';

  if( WM->getCurrentTime() == timeLastSent )
  {
    Log.logFromSignal( 2, " already sent message; don't send" );
    return false;
  }

  if(  WM->isQueuedActionPerformed()    == false &&   // don't send action when
     m_queueOneCycleCommand.commandType != CMD_CATCH && // previous one is not
     WM->isFullStateOn() == false )                    // fullstate sense not
  {                                          // not processed yet
    Log.logFromSignal( 2, " previous message not processed yet; don't send" );
    return false;                              // except with catch since
  }                                          // too important

  // make string of primary action and put it in 'strCommand' variable
  bool bReturn = m_queueOneCycleCommand.getCommandString( strCommand, SS );

  if( bReturn == false )
    cerr << WM->getCurrentCycle() << ", " <<  WM->getPlayerNumber() << " "
```

```
      << "Acthandler::failed to create primary command string" << endl;

if( strCommand[0] == '\0' )
{
  bNoOneCycle = true;
  Log.logFromSignal( 2, " no primary action in queue" );
}

// make string of all other commands and add them to the end of 'strCommand'
for( int i = 0; i < m_iMultipleCommands ; i ++ )
{
  bReturn = m_queueMultipleCommands[i].getCommandString(
                              &strCommand[strlen(strCommand)], SS );
  if( bReturn == false )
    cerr << WM->getCurrentCycle() << ", " <<  WM->getPlayerNumber() << " "
       << "Acthandler::failed to create secondary command string " <<
      m_queueMultipleCommands[i].commandType <<  endl;
}

char strComm[MAX_SAY_MSG];
strcpy( strComm, WM->getCommunicationString() );
if( strlen( strComm ) != 0 )
{
  sprintf( &strCommand[strlen(strCommand)], "(say \"%s\")", strComm );
  WM->setCommunicationString( "" );
}

// send the string to the server (example string: (dash 100)(turn_neck -19))
if( strCommand[0] != '\0' )
{
  timeLastSent        = WM->getCurrentTime();
  connection->sendMessage( strCommand );
  Log.logFromSignal( 2, " send queued action to server: %s", strCommand);
}
else
{
  Log.logFromSignal( 2, " no action in queue??" );
  return false;
}

if( ! bNoOneCycle ) // if primary action was send, place it at end of array
  m_queueMultipleCommands[m_iMultipleCommands++] = m_queueOneCycleCommand;

// let worldmodel know which commands were sent to the server
WM->processQueuedCommands( m_queueMultipleCommands, m_iMultipleCommands );
m_iMultipleCommands = 0;
```

```
  // decrease amount of times primary action still has to be sent, if 0 delete
  // it, furthermore set number of multiple commands to zero
  if( --m_queueOneCycleCommand.iTimes  == 0 )
    m_queueOneCycleCommand.commandType = CMD_ILLEGAL;

  for( int i = 0; i < CMD_MAX_COMMANDS; i++ )
    m_queueMultipleCommands[i].commandType = CMD_ILLEGAL;

  return true;
}


/*! This method returns the primary command that is currently stored in the
    queue. */
SoccerCommand ActHandler::getPrimaryCommand(  )
{
  return m_queueOneCycleCommand;
}

/*! This method puts a SoccerCommand in the queue. The last added command
    will be sent to the soccerserver when the method sendCommands is performed.
    Normally this is done when a signal set by the SenseHandler arrives.
    \param command SoccerCommand that should be put in the queue.
    \return true when command is added, false otherwise (queue is full) */
bool ActHandler::putCommandInQueue( SoccerCommand command )
{
  int i;
  bool bOverwritten = false;

  if( command.commandType == CMD_ILLEGAL )
    return false;
  if( SoccerTypes::isPrimaryCommand( command.commandType ) )
    m_queueOneCycleCommand = command;           // overwrite primary command
  else                                  // non-primary command
  {
    for( i = 0; i < m_iMultipleCommands ; i ++ )
      if( m_queueMultipleCommands[i].commandType == command.commandType )
      {
        m_queueMultipleCommands[i] = command;   // if command already in queue
        bOverwritten = true;                    // overwrite it
      }

    // 1 less to save space for primary command
    if( bOverwritten == false && m_iMultipleCommands == CMD_MAX_COMMANDS-1 )
    {
```

```
      cerr << "(ActHandler::putCommandInQueue) too many commands" << endl;
      return false;
    }
    if( bOverwritten == false  ) // add it when command was not yet in queue
      m_queueMultipleCommands[m_iMultipleCommands++] = command;
  }

  return true;
}

/*! This method sends a single command directly to the server. First a string
    is made from the SoccerCommand and afterwards this string is send to the
    server using the method sendMessage.
    \param soc SoccerCommand that should be send to the server.
    \return true when message was sent, false otherwise */
bool ActHandler::sendCommand( SoccerCommand soc )
{
  char strCommand[MAX_MSG];
  soc.getCommandString( strCommand, SS );
  return sendMessage( strCommand );
}

/*! This method sends a single string directly to the server. To make sure
    this message arrives, the time of one complete cycle is waited before and
    after the message is sent.
    \param str string that should be sent to the server
    \return true when message was sent, false otherwise */
bool ActHandler::sendMessage( char * str )
{
  emptyQueue( );
#ifdef WIN32
  Sleep( SS->getSimulatorStep() );
#else
  poll( 0, 0, SS->getSimulatorStep() );
#endif

  bool bReturn = connection->sendMessage( str );
  Log.logFromSignal( 2, " send message to server and wait: %s", str);

#ifdef WIN32
  Sleep( SS->getSimulatorStep() );
#else
  poll( 0, 0, SS->getSimulatorStep() );
#endif
  return bReturn;
}
```

```
/*! This method sends a single command directly to the server. First a string
    is made from the SoccerCommand and afterwards this string is send to the
    server using the method sendMessageDirect.
    \param soc SoccerCommand that should be send to the server.
    \return true when message was sent, false otherwise */
bool ActHandler::sendCommandDirect( SoccerCommand soc )
{
  char strCommand[MAX_MSG];
  if( soc.commandType == CMD_ILLEGAL )
    return false;
  soc.getCommandString( strCommand, SS );
  return sendMessageDirect( strCommand );
}

/*! This method sends a single string directly to the server.
    \param str string that should be sent to the server
    \return true when message was sent, false otherwise */
bool ActHandler::sendMessageDirect( char * str )
{
  bool bReturn = connection->sendMessage( str );
  Log.logFromSignal( 2, " send message to server directly: %s", str);
  return bReturn;
}
```

Parse.h

```
#ifndef _PARSE_
#define _PARSE_

/*****************************************************************************
/
/*******************  CLASS PARSE  ***************************************/
/*****************************************************************************
/

/*! This class contains several static methods which can be used for parsing
   string messages and uses some of the ideas contained in CMU'99 source code
   of Peter Stone. Tests shows that scanning integers has a performance
   increase of 30.3% over the method used by CMUnited and 68.0% over sscanf.
   For parsing doubles the performance increase was 15.4% compared to CMUnited
   and 85.1% compared to sscanf. */
class Parse
{
public:

  // methods which return a specific type of value from a string message
  static double parseFirstDouble            ( char** strMsg             );
  static int    parseFirstInt               ( char** strMsg             );

  // methods which move to a specific position in a string message
  static char   gotoFirstSpaceOrClosingBracket( char** strMsg            );
  static int    gotoFirstOccurenceOf        ( char   c     , char** strMsg);
  static char   gotoFirstNonSpace           ( char** strMsg             );
};
#endif
```

Parse.cpp

```
#include "Parse.h"

#include <ctype.h>    // needed for isdigit
#include <math.h>     // needed for pow
#include <string.h>   // needed for strlen

/*****************************************************************************
/
/********************** CLASS PARSE
*****************************************/
/*****************************************************************************
/

/*! This method walks through the string starting at the character where strMsg
    points to and stops when or the end of the string is reached or a non space
    is reached. strMsg is updated to this new position.
    \param strMsg pointer to a character in a string array
    \return character that is at the first non space, '\0' when end of string
    is reached. */
char Parse::gotoFirstNonSpace( char** strMsg )
{
  while(*strMsg && isspace(**strMsg) )
    (*strMsg)++;
  return  (*strMsg) ?  **strMsg : '\0';

}

/*! This method walks through the string starting at the character where strMsg
    points to and stops when<BR>
    - the end of the string is reached<BR>
    - a character different than ' ' and ')' is read<BR>
    strMsg is changed after this method is called.
    \param strMsg pointer to a character in a string array
    \return first character that is not equal to ' ' or ')', '\0' when string
          didn't contain such a character. */
char Parse::gotoFirstSpaceOrClosingBracket( char** strMsg )
{
  while( *strMsg && **strMsg!=' ' && **strMsg!=')' )
    (*strMsg)++;
  return  (*strMsg) ?  **strMsg : '\0';
}

/*! This method walks through the string starting at the character where strMsg
    points to and stops when the character c is reached. strMsg is changed
    after this method is called..
    \param c character that is searched for in strMsg.
```

```
    \param strMsg pointer to a character in a string array
    \return number of character skipped to reach c, -1 when not found */
int Parse::gotoFirstOccurenceOf( char c, char** strMsg )
{
  int i=0;
  while(**strMsg && **strMsg != c )
  {
    i++;
    (*strMsg)++;
  }
  if( ! **strMsg )
    return -1;
  return i;
}

/*! This method walks through the string starting at the character where strMsg
    points to and returns the first integer that can be read from this string.
    Any other characters in front of this integer are discarded. After this
    method is returned, strMsg points to the first character after the final
    value of the integer.
    \param strMsg pointer to a character in a string array
    \return first integer that can be read from this string. */
int Parse::parseFirstInt( char** strMsg )
{
  int  iRes  = 0;
  bool bIsMin= false;
  char *str  = *strMsg;

  while( *str != '\0' && !isdigit(*str) && *str!='-')
    str++;                          // walk to first non digit or minus sign


  if( *str != '\0' && *str=='-')       // if it was a minus sign, remember
  {
    bIsMin=true;
    str++;
  }

  while( *str != '\0' && *str<='9' && *str>='0' )   // for all digits
  {
     iRes=iRes*10+(int)(*str-'0');                 // multiply old res with 10
     str++;                                  // and add new value
  }
  *strMsg = str;
  return (bIsMin) ? -iRes : iRes;
}
```

```
/*! This method walks through the string starting at the character where strMsg
    points to and returns the first double that can be read from this string.
    Any other characters in front of this integer are discarded. After this
    method is returned, strMsg points to the first character after the final
    value of the double. 4e-3, and NaN or nan are also recognized. When input
    contains NaN or nan, -1000.0 is returned.
    \param strMsg pointer to a character in a string array
    \return first double that can be read from this string. */
double Parse::parseFirstDouble( char** strMsg )
{
  double dRes=0.0, dFrac=1.0;
  bool bIsMin=false, bInDecimal=false, bCont=true;
  char *str = *strMsg;

  // go to first part of double (digit, minus sign or '.')
  while( *str != '\0' && !isdigit(*str) && *str!='-' && *str!='.')
  {
    // when NaN or nan is double value, return -1000.0
    if( (*str=='N' && strlen(str)>3 && *(str+1)=='a' && *(str+2)=='N') ||
        (*str=='n' && strlen(str)>3 && *(str+1)=='a' && *(str+2)=='n') )
    {
      *strMsg = str+3;
      return -1000.0;
    }
    else
       str++;
  }

  if( *str != '\0' && *str=='-')              // if minus sign, remember that
  {
    bIsMin=true;
    str++;
  }

  while( *str != '\0' && bCont)               // process the number bit by bit
  {
    if( *str=='.' )                           // in decimal part after '.'
      bInDecimal = true;
    else if( bInDecimal && *str<='9' && *str>='0') // number and in decimal
    {
      dFrac=dFrac*10.0;                       // shift decimal part to right
      dRes += (double)(*str-'0')/dFrac;       // and add number
    }
    else if( *str<='9' && *str>='0' )         // if number and not decimal
      dRes=dRes*10+(double)(*str-'0');        // shift left and add number
```

```
   else if( *str=='e' ) // 10.6e-08          // if to power e
   {
    if( *(str+1) == '+' )                   // determine sign
      dRes *=  pow(10, parseFirstInt(&str)); // and multiply with power
    else if( *(str+1) == '-' )
    {
      str = str+2;                          // skip -
      dRes /=  pow(10, parseFirstInt(&str)); // and divide by power
    }
    bCont = false;                          // after 'e' stop
   }
   else
    bCont = false;                          // all other cases stop

   if( bCont == true )                       // update when not stopped yet
    str++;
  }
  *strMsg = str;
  return (bIsMin && dRes != 0.0) ? -dRes : dRes;
}

/*************************************************************************
/
/********************* TESTING PURPOSES
*************************************/
/*************************************************************************
/
/*
#include<iostream.h>

int main( void )
{
  double d = 13.6e+15;
  char str[] = "13.6e+15";
  double d2 = 13.6e-15;
  char str2[] = "13.6e-15";

  char *strmsg;
  strmsg = &str[0];
  cout  << d << endl;
  cout << Parse::parseFirstDouble( &strmsg )<< endl;
  strmsg = &str2[0];
  cout  << d2 << endl;
  cout << Parse::parseFirstDouble( &strmsg ) << endl;
  return 0;
}*/
```

SenseHandler.h

```
#ifndef _SENSEHANDLER_
#define _SENSEHANDLER_

#include "Connection.h"
#include "WorldModel.h"

extern Logger Log; /*!< This is a reference to the Logger to write  info to */

#ifdef WIN32
  DWORD WINAPI sense_callback( LPVOID v );
#else
  void* sense_callback( void *v );
#endif

using namespace std;

/*****************************************************************************
/
/******************** CLASS SENSEHANDLER
************************************/
/*****************************************************************************
/

/*! Class SenseHandler receives input from a (Robocup Simulation)
    server (through an instance of Connection). The class contains
    methods to parse the incoming messages and sends these to the
    WorldModel accordingly. After this class is instantiated, a
    specific thread must call the function sense_callback.  This
    thread will then automatically call handleMessagesFromServer()
    which loops forever. In this way all messages are received and
    parsed (since the receiveMessage from the Connection blocks till a
    message arrives). Other threads can think about the next action
    while the SenseHandler sends the new information to the
    WorldModel. */
class SenseHandler {
  WorldModel     *WM;       /*!< Worldmodel containing all data of the match*/
  ServerSettings *SS;       /*!< ServerSettings with all server settings    */
  PlayerSettings *PS;       /*!< PlayerSettings with all client settings    */
  Connection* connection;   /*!< Connection with server to receive messages */
  int        iTimeSignal;   /*!< Wait time (microsec) before sense calls act*/
  int        iTriCounter;   /*!< Indicates when see message will arive      */
  int        m_iSeeCounter; /*!<! Used to count number of see msg in 1 cycle*/
  int        iSimStep;      /*!< Length (microsec) of server cycles         */
#ifdef WIN32
  UINT       iTimer;        /*!< timer used to call sigalarmHandler function*/
  UINT       timerRes;      /*!< timer resolution for the application       */
```

```
#else
  struct      itimerval itv; /*!< timer used to set alarm to send action     */
#endif


public:
  SenseHandler( Connection* c, WorldModel* wm, ServerSettings *ss,
                PlayerSettings *ps );

  // start the loop to handle the messages from the server
  void    handleMessagesFromServer      (              );

  // methods to determine when the next action should be sent to the server.
  void    setTimeSignal                 (                          );

  // method to analyze incoming messages
  bool    analyzeMessage                ( char *strMsg               );
  bool    analyzeSeeGlobalMessage       ( char *strMsg               );
  bool    analyzeFullStateMessage       ( char *strMsg               );
  bool    analyzeSeeMessage             ( char *strMsg               );
  bool    analyzeSenseMessage           ( char *strMsg               );
  bool    analyzeInitMessage            ( char *strMsg               );
  bool    analyzeHearMessage            ( char *strMsg               );
  bool    analyzePlayerMessage          ( int  iTime  , char *strMsg );
  bool    analyzeCoachMessage           ( char *strMsg               );
  bool    analyzeChangePlayerTypeMessage( char *strMsg               );
  bool    analyzeServerParamMessage     ( char *strMsg               );
  bool    analyzeCheckBall              ( char *strMsg               );
  bool    analyzePlayerTypeMessage      ( char *strMsg               );
  bool    analyzePlayerParamMessage     ( char *strMsg               );

  // utility functions
  bool    readServerParam               ( char *strParam,
                                          char *strMsg               );
};

#endif
```

SenseHandler.cpp

```
#include "SenseHandler.h"
#include "ActHandler.h"  // sigalarmHandler
#include "Parse.h"

#include <signal.h>   // needed for SIGALARM
#include <string.h>   // needed for strlen
#include <stdio.h>    // needed for printf
#include <iostream>   // needed for cout

/*****************************************************************************
/
/******************** CLASS SENSEHANDLER
************************************/
/*****************************************************************************
/

/*! This function is needed to start the Sense Thread (thread that continually
    waits for input and parses the input). This function is needed since it is
    not possible to call a method from a class using a thread. So this function
    calls handleMessagesFromServer from the SenseHandler class.
    \param v pointer to a SenseHandler class.*/
#ifdef WIN32
DWORD WINAPI sense_callback( LPVOID v )
#else
void* sense_callback( void *v )
#endif
{
  Log.log( 1, "Starting to listen for server messages" );
  SenseHandler* s = (SenseHandler*)v;
  s->handleMessagesFromServer( );
  return NULL;
}


/*! Constructor for the SenseHandler. It needs a reference to a connection and
    a reference to a worldmodel.
    \param c Connection from which input is received
    \param wm WorldModel to which new information will be sent for processing
    \param ss ServerSettings that contain the parameters used by the server
    \param ps PlayerSettings that determine how to interact with messages. */
SenseHandler::SenseHandler( Connection *c, WorldModel *wm, ServerSettings *ss,
                            PlayerSettings *ps )
{
  connection          = c;
  SS                  = ss;
  PS                  = ps;
```

```
  WM                  = wm;
  iSimStep            = SS->getSimulatorStep()*1000;
  iTimeSignal         = (int)(iSimStep*0.85);

#ifdef WIN32
  TIMECAPS tc;
  UINT     resolution = 1;   // timer resolution for the application (ms)
  iTimer = NULL;

  // set the minimum timer resolution for an application
  if (TIMERR_NOERROR == timeGetDevCaps( &tc, sizeof(TIMECAPS) ))
  {
    timerRes = min( max( tc.wPeriodMin, resolution ), tc.wPeriodMax );
    timeBeginPeriod( timerRes );
  }
#else
  struct sigaction sigact;

  sigact.sa_flags = SA_RESTART; // do not unblock primitives (like recvfrom)
  sigact.sa_handler = (void (*)(int))sigalarmHandler;
  sigaction( SIGALRM, &sigact, NULL );

  // set timer signal to indicate when ActHandler should sent commands to the
  // server, this structure will later be filled with exact timing values
  itv.it_interval.tv_sec  = 0;
  itv.it_interval.tv_usec = 0;
  itv.it_value.tv_sec     = 0;
  itv.it_value.tv_usec    = 0;
#endif
}

/*! This is the main routine of this class. It loops forever (till the thread
    is destroyed) and receives and parses the incoming messages.   */
void SenseHandler::handleMessagesFromServer( )
{
  char strBuf[MAX_MSG];
  int i=0;

  while( 1 )
  {
    strBuf[0]='\0';
    if( i != -1 )                                  // if no error
      i = connection->receiveMessage( strBuf, MAX_MSG );  // get message
    if( strBuf[0] != '\0' )                        // if not empty
      analyzeMessage( strBuf );                    // parse message
  }
```

```
}


/*! This method sets the time signal. This is the time that should be
    waited before the next action should be sent to the server. As
    soon as a sense message arrives this method is called. Using the
    information from the member variable 'iTriCounter' which denotes
    when the see message will arrive in this cycle (0=first half,
    1=2nd half, 2=no see, all for the default view frequency) the
    timer is set. The values that denote the fraction of the
    simulation step that is waited are all defined in PlayerSettings,
    such that they can be easily changed. */
void SenseHandler::setTimeSignal( )
{
  if( WM->getAgentViewFrequency() == 1.0 ) // VA_NORMAL AND VQ_HIGH (default)
  {
    if( iTriCounter % 3 == 0 )            // see will arrive first half cycle
    {
      iTimeSignal = (int)(iSimStep * PS->getFractionWaitSeeBegin() );
      iTriCounter = 0;
    }
    else if( iTriCounter % 3 == 1 )       // see will arrive 2nd half of cycle
    {
      iTimeSignal = (int)(iSimStep * PS->getFractionWaitSeeEnd() );
    }
    else                                  // no see will arrive
      iTimeSignal = (int)(iSimStep * PS->getFractionWaitNoSee( ) );
  }
  else if( WM->getAgentViewFrequency() == 2.0 ) // VA_WIDE AND VQ_HIGH
  {
    if( iTriCounter % 3 == 0 )            // see will arrive
    {
      iTimeSignal = (int)(iSimStep * PS->getFractionWaitSeeEnd() );
      iTriCounter = 0;
    }
    else                                  // no see will arrive
      iTimeSignal = (int)(iSimStep * PS->getFractionWaitNoSee() );
  }
  else                                    // VA_NARROW AND VQ_HIGH
    iTimeSignal = (int)(iSimStep * PS->getFractionWaitSeeEnd() );

  iTriCounter++;
#ifdef WIN32
  // kill the previous timer
  if (iTimer != NULL) timeKillEvent( iTimer );
  // start a new one
```

```
  iTimer = timeSetEvent( iTimeSignal / 1000, timerRes,
                         sigalarmHandler, (DWORD)0, TIME_ONESHOT );
#else
  itv.it_value.tv_usec = iTimeSignal;
  setitimer( ITIMER_REAL, &itv, NULL );
#endif
}

/*! This method analyzes the type of the incoming message and calls the
    message that corresponds to this message.
    \param strMsg message that should be parsed.
    \return bool indicating whether the message was parsed or not */
bool SenseHandler::analyzeMessage( char *strMsg )
{
  Log.log( 1, strMsg );
  bool bReturn = false;

  switch( strMsg[1] )
  {
    case 'c':
       if( strMsg[2] == 'h' )
         return analyzeChangePlayerTypeMessage( strMsg );      // ( c hange_
       else
         ;                                                     // (clang
       break;
    case 'f':
         return analyzeFullStateMessage( strMsg );      // ( f ullstate_
    case 'o':                                             // ( o k
       if( strlen(strMsg) > 14 && strMsg[4] == 'c' && strMsg[10] == 'b' )
         analyzeCheckBall( strMsg );                          // (ok check_ball
       return true;
    case 's':
    {
      switch( strMsg[3] )
      {
      case 'e':
        if( strMsg[5] == 'g')
          return analyzeSeeGlobalMessage  ( strMsg ); // (se e_g
        else if( WM->isFullStateOn( ) == false )
          return analyzeSeeMessage        ( strMsg ); // (se e
        break;
      case 'n':
        bReturn = analyzeSenseMessage      ( strMsg ); // (se n se
        if( WM->isFullStateOn( ) == true  )
          WM->updateAfterSenseMessage( );
        return bReturn;
```

```
      break;
     case 'r': return analyzeServerParamMessage( strMsg ); // (se r ver_param
     default : break;
     }
    }
      break;
   case 'i':    return analyzeInitMessage       ( strMsg ); // ( i nit
   case 'h':    return analyzeHearMessage       ( strMsg ); // ( h ear
   case 'p':    return ( strMsg[8] == 't')
               ? analyzePlayerTypeMessage ( strMsg )  // (player_ t ype
               : analyzePlayerParamMessage( strMsg ); // (player_ p aram
   case 'e':    printf("(%d,%d) %s\n", WM->getCurrentCycle(),
                     WM->getPlayerNumber(),strMsg);// ( error
      break;
   case 't':    Log.logWithTime( 2, " incoming think message" );
               WM->processRecvThink( true );          // ( think
      break;
   default:     cerr << "(" << WM->getCurrentTime() << ", " <<
                  WM->getPlayerNumber()
                  << ") (SenseHandler::analyzeMessage) " <<
                "ignored message: " << strMsg << "\n";
               return false;
  }
  return false;
}


/*! This method analyzes a see Message. It gets the time from the message and
    tries to synchronize with the server. Then the message is stored in the
    world model, which processes it when it performs an update.
    \return bool indicating whether the message was parsed correctly. */
bool SenseHandler::analyzeSeeMessage( char *strMsg )
{
  strcpy( WM->strLastSeeMessage, strMsg );

  Log.logWithTime( 2, " %s",strMsg );

  if( WM->getRelativeDistance( OBJECT_BALL ) < SS->getVisibleDistance() )
    Log.logWithTime( 560, "%s", WM->strLastSeeMessage );

  Time    time = WM->getTimeLastRecvSenseMessage();
  int iTime = Parse::parseFirstInt( &strMsg );         // get the time
  if( time.getTime() != iTime )
  {
    cerr << "(SenseHandler:analyzeSeeMessage) see and different time as sense:"
        << time.getTime() << " vs. " << iTime << endl;
```

```
    return false;
  }

  // count number of see message in this cycle
  if( WM->getTimeLastSeeMessage() == time )
    m_iSeeCounter++;
  else
    m_iSeeCounter = 1;

  // do nothing with second see, since it adds nothings
  if( m_iSeeCounter >= 2 )
  {
    Log.logWithTime( 4, "second see message in cycle; do nothing " );
    return true;
  }

  // reset the send pattern when previous cycle(s) no see arrived
  if( WM->getAgentViewFrequency() == 1.0 && // VA_NORMAL; previous cycle no see
      time.getTimeDifference( WM->getTimeLastSeeMessage() )== 2 )
    iTriCounter = 1;              // see will arive in 2nd half in next cycle
  else if( WM->getAgentViewFrequency() == 2.0 && // VA_WIDE; two cycles no see
          time.getTimeDifference( WM->getTimeLastSeeMessage() ) == 3 )
    iTriCounter = 1;              // no see will arrive next two cycles

  WM->setTimeLastSeeMessage( time );   // set time of last see message
  return true;
}

/*! This method analyzes a see Message. All information from the different
    objects that is stored in a see message is send to worldmodel.
    A see message looks like(see 0 ((g r) 64.1 13) ((f r t) 65.4 -16) ....
    \param strMsg message that should be parsed
    \return bool indicating whether the message was parsed correctly. */
bool SenseHandler::analyzeSeeGlobalMessage( char *strMsg )
{
  Log.logWithTime( 2, " incoming see global message" );
  strcpy( WM->strLastSeeMessage, strMsg );

  ObjectT o;
  bool    isGoalie;
  double  dX, dY, dVelX, dVelY;
  int     iTime;
  AngDeg  angBody, angNeck;
  Time    time = WM->getCurrentTime();

  iTime = Parse::parseFirstInt( &strMsg );         // get the time
```

```
  time.updateTime( iTime );

  while( *strMsg != ')' )                    // " ((objname.." or ")"
  {
   dVelX = dVelY = UnknownDoubleValue;
   angBody = angNeck = UnknownAngleValue;
   strMsg += 2;          // go the start of the object name

   // get the object type at the current position in the string
   o = SoccerTypes::getObjectFromStr( &strMsg, &isGoalie, WM->getTeamName() );
   if( o == OBJECT_ILLEGAL )
   {
    Log.log( 4, "Illegal object" );
    Log.log( 4, "total messages: %s", WM->strLastSeeMessage );
    Log.log( 4, "rest of message: %s", strMsg );
   }

   dX = Parse::parseFirstDouble( &strMsg );       // parse first value
   dY = Parse::parseFirstDouble( &strMsg );       // parse second value
   if ( *strMsg != ')' )                    // if it was no flag
   {
    dVelX = Parse::parseFirstDouble( &strMsg );   // parse delta x
    dVelY = Parse::parseFirstDouble( &strMsg );   // parse delta y
    if( *strMsg != ')' )                    // stil not finished
    {                                  // get body and neck angle
     angBody = Parse::parseFirstDouble( &strMsg );
     angNeck = Parse::parseFirstDouble( &strMsg );
    }
   }
   // skip ending bracket of object information.
   Parse::gotoFirstOccurenceOf( ')', &strMsg );
   strMsg++;

   // process the parsed information (unread values are Unknown...)
   WM->processSeeGlobalInfo( o, time, VecPosition(dX,dY),
              VecPosition(dVelX,dVelY), angBody, angNeck );
   }
  WM->setTimeLastSeeGlobalMessage( time );  // set time last see global message
  return true;
}

/*! This method parses a full state message. This message contains all
   information from the playing field without noise. It will not be used
   during real tournaments. */
bool SenseHandler::analyzeFullStateMessage( char *strMsg )
{
```

```
Log.restartTimer( );
Log.logWithTime( 2, " incoming fullstate message" );
Log.log( 4, " fullstate message: %s", strMsg );
strcpy( WM->strLastSeeMessage, strMsg );

ObjectT o;
bool    isGoalie;
double  dX, dY, dVelX, dVelY;
int     iTime;
AngDeg  angBody, angNeck;
Time    time = WM->getCurrentTime();

iTime = Parse::parseFirstInt( &strMsg );        // get the time
time.updateTime( iTime );
Log.log( 4, "fullstate time: %d", time.getTime() );

strMsg++;                                       // skip space
Parse::gotoFirstOccurenceOf( ' ', &strMsg );   // skip (pmode
strMsg++;                                       // skip space

Log.log( 4, "fullstate parse ref: %s", strMsg );
RefereeMessageT rm = SoccerTypes::getRefereeMessageFromStr( strMsg );
PlayModeT       pm = SoccerTypes::getPlayModeFromRefereeMessage( rm );
if( pm != PM_ILLEGAL )
  WM->setPlayMode( pm );

Parse::gotoFirstOccurenceOf( 'e', &strMsg );   // go to end of vmode
strMsg++;                                       // skip 'e'
strMsg++;                                       // skip space

Log.log( 4, "fullstate parse qua: %s", strMsg );
ViewQualityT vq = SoccerTypes::getViewQualityFromStr( strMsg );
Parse::gotoFirstOccurenceOf( ' ', &strMsg );   // go to end of quality
strMsg++;
Log.log( 4, "fullstate parse ang: %s", strMsg );
ViewAngleT   va = SoccerTypes::getViewAngleFromStr( strMsg );

Log.log( 4, "fullstate parse count: %s", strMsg );
WM->setNrOfCommands( CMD_KICK       , Parse::parseFirstInt( &strMsg ) );
WM->setNrOfCommands( CMD_DASH       , Parse::parseFirstInt( &strMsg ) );
WM->setNrOfCommands( CMD_TURN       , Parse::parseFirstInt( &strMsg ) );
WM->setNrOfCommands( CMD_CATCH      , Parse::parseFirstInt( &strMsg ) );
WM->setNrOfCommands( CMD_MOVE       , Parse::parseFirstInt( &strMsg ) );
WM->setNrOfCommands( CMD_TURNNECK   , Parse::parseFirstInt( &strMsg ) );
WM->setNrOfCommands( CMD_CHANGEVIEW , Parse::parseFirstInt( &strMsg ) );
WM->setNrOfCommands( CMD_SAY        , Parse::parseFirstInt( &strMsg ) );
```

```
int iArmMovable = Parse::parseFirstInt( &strMsg );
int iArmExpires = Parse::parseFirstInt( &strMsg );
Parse::parseFirstDouble( &strMsg ); // skip pointto info, comes later
Parse::parseFirstDouble( &strMsg ); // skip pointto info, comes later
WM->setNrOfCommands( CMD_POINTTO    , Parse::parseFirstInt( &strMsg ) );

Parse::gotoFirstOccurenceOf( 'b', &strMsg );   // go to ball position

Log.log( 4, "fullstate parse ball: %s", strMsg );
dX    = Parse::parseFirstDouble( &strMsg );    // parse first value
dY    = Parse::parseFirstDouble( &strMsg );    // parse second value
dVelX = Parse::parseFirstDouble( &strMsg );    // parse third value
dVelY = Parse::parseFirstDouble( &strMsg );    // parse fourth value
if( WM->isBeforeKickOff() )
  dX = dY = dVelX = dVelY = 0.0;
if( WM->getSide() == SIDE_RIGHT )
{
  dX    *= -1;
  dY    *= -1;
  dVelX *= -1;
  dVelY *= -1;
}
WM->processSeeGlobalInfo( OBJECT_BALL, time, VecPosition(dX,dY),
              VecPosition(dVelX,dVelY), -1, -1 );
strMsg++;
Log.log( 4, "fullstate ball: %f %f %f %f", dX, dY, dVelX, dVelY );

while( *strMsg != ')' )                        // " ((objname.." or ")"
{
  dVelX = dVelY = UnknownDoubleValue;
  angBody = angNeck = UnknownAngleValue;
  strMsg += 2;          // go the start of the object name
  Log.log( 4, "fullstate parse object: %s", strMsg );
  // get the object type at the current position in the string
  o = SoccerTypes::getObjectFromStr( &strMsg, &isGoalie,
                    (WM->getSide() == SIDE_LEFT ) ? "l" : "r" );

  dX      = Parse::parseFirstDouble( &strMsg );   // parse x position
  dY      = Parse::parseFirstDouble( &strMsg );   // parse y position
  dVelX   = Parse::parseFirstDouble( &strMsg );   // parse x velocity
  dVelY   = Parse::parseFirstDouble( &strMsg );   // parse y velocity
  angBody = Parse::parseFirstDouble( &strMsg );   // parse body angle
  angNeck = Parse::parseFirstDouble( &strMsg );   // parse neck angle

  if( WM->getSide() == SIDE_RIGHT )
```

```
    {
      dX    *= -1;
      dY    *= -1;
      dVelX *= -1;
      dVelY *= -1;
      angBody = VecPosition::normalizeAngle( angBody + 180 );
    }

    double dStamina  = Parse::parseFirstDouble( &strMsg );  // get stamina
    double dEffort   = Parse::parseFirstDouble( &strMsg );  // get effort
                       Parse::parseFirstDouble( &strMsg );  // skip recovery

    // skip ending bracket of stamina and then of object information.
    Parse::gotoFirstOccurenceOf( ')', &strMsg );
    Parse::gotoFirstOccurenceOf( ')', &strMsg );

    strMsg++;
    strMsg++;

    Log.log( 1, "fullstate obj %d: %f %f %f %f %f %f", o, dX, dY, dVelX, dVelY,
                angBody, angNeck );
    // process the parsed information
    if( o == WM->getAgentObjectType() )
      WM->processNewAgentInfo( vq, va, dStamina, dEffort, -1.0, -1.0, -angNeck,
                               -1,iArmMovable, iArmExpires, VecPosition(0,0));

    WM->processSeeGlobalInfo( o, time, VecPosition(dX,dY),
                              VecPosition(dVelX,dVelY), angBody, angNeck );

  }
  WM->setTimeLastSeeGlobalMessage( time );  // set time last see global message
  WM->setTimeLastSenseMessage( time );      // set time last see global message

  return true;
}

/*! This method analyzes a sense message. All information from the player is
    parsed and updated in the WorldModel.
    A sense message looks like (sense_body 0 (view_mode high normal)
    (stamina 2000 1) (speed 0 0) (head_angle 0) (kick 0) (dash 0)
    (turn 0) (say 0) (turn_neck 0) (catch 0) (move 0) (change_view 0))
    \param strMsg message that should be parsed
    \return bool indicating whether the message was parsed correctly. */
bool SenseHandler::analyzeSenseMessage( char *strMsg )
{
  Log.log( 999, "%s", strMsg );
```

```
  // cerr << strMsg << endl;
  // set the synchronization counter, this is a value [0..2] indicating the
  // section of the pattern this cycle is in. It gives an indication when new
  // visual information will arrive.

  if( SS->getSynchMode() == false )
    setTimeSignal();                    // set signal when to send action
  strcpy( WM->strLastSenseMessage, strMsg );

  if( WM->getRelativeDistance( OBJECT_BALL ) < SS->getVisibleDistance() )
    Log.logWithTime( 560, "%s", WM->strLastSenseMessage );

  int iTime = Parse::parseFirstInt( &strMsg );// get time
  Time timeOld = WM->getCurrentTime();
  Time timeNew = timeOld;
  timeNew.updateTime( iTime );

  if( timeNew.getTimeDifference( timeOld ) > 1 )
    Log.log( 1, "Missed a sense!!" );

  Log.logWithTime ( 2, "\n\nSENSE (%d, %d)", timeNew.getTime(),
  timeNew.getTimeStopped() );
  Log.restartTimer( );
  iSimStep              = SS->getSimulatorStep()*1000;
  iTimeSignal           = (int)(iSimStep*0.85);
  Log.logWithTime ( 2, " alarm after %d", iTimeSignal );

  WM->setTimeLastSenseMessage( timeNew ); // set the time

//  Log.logWithTime( 2, " end analyzing sense" );
  return true;
}

/*! This method analyzes an init message. All information from the
    initialization is parsed and updated in the WorldModel.
    An init message looks like (init [l|r] 10 before_kick_off)
    \param strMsg message that should be parsed
    \return bool indicating whether the message was parsed correctly. */
bool SenseHandler::analyzeInitMessage( char *strMsg )
{
  Log.log( 999, "%s", strMsg );
  strMsg += 6;                                  // go to Side
  WM->setSide( SoccerTypes::getSideFromStr( strMsg ) );   // get and set Side
  int nr = Parse::parseFirstInt( &strMsg );             // get and set number
  if( nr == 0 )                                 // coach
  {
```

```
    WM->setPlayerNumber( nr );
    cerr << strMsg  << endl;
    return true;
  }
  WM->setAgentObjectType( SoccerTypes::getTeammateObjectFromIndex( nr - 1 ) );
  WM->setPlayerNumber( nr );
  strMsg++;                                    // skip space to pm
  WM->setPlayMode( SoccerTypes::getPlayModeFromStr( strMsg ) ); // get playmode
  return true;
}

/*! This method analyzes a hear message. When the message is from the
    referee the message is parsed and the new play mode is set or the
    goal difference is adjusted. When the message comes from another
    player the method analyzePlayerMessage is called A hear message
    looks like (hear 0 self|referee|dir message)

    \param strMsg message that should be parsed
    \return bool indicating whether the message was parsed correctly. */
bool SenseHandler::analyzeHearMessage( char *strMsg )
{
  RefereeMessageT rm;
  PlayModeT       pm;
  strcpy( WM->strLastHearMessage, strMsg);

  int iTime = Parse::parseFirstInt( &strMsg );             // ignore time
  Time time( iTime );

  switch( Parse::gotoFirstNonSpace( &strMsg ) )
  {
    case 'r':                                    // referee
      Log.log( 999, "%s", WM->strLastHearMessage );
      WM->setTimeLastRefereeMessage( time );
      Parse::gotoFirstOccurenceOf( ' ', &strMsg );         // go to start
      Parse::gotoFirstNonSpace   ( &strMsg      );         // and first part
      rm = SoccerTypes::getRefereeMessageFromStr( strMsg ); // get the ref msg
      Log.logWithTime( 2, " referee message: %s %s",
      SoccerTypes::getRefereeMessageStr(rm), WM->strLastHearMessage);
      pm = SoccerTypes::getPlayModeFromRefereeMessage( rm );// get play mode
      if( pm != PM_ILLEGAL )                          // from ref msg
        WM->setPlayMode( pm );                        // if was pm, set

      switch( rm )
      {
        case REFC_GOAL_LEFT:                        // goal left
          if( WM->getSide() == SIDE_LEFT )
```

```
        WM->addOneToGoalDiff();
      else
        WM->subtractOneFromGoalDiff();
      WM->processSeeGlobalInfo( OBJECT_BALL, time, VecPosition( 0, 0 ),
          VecPosition( 0, 0 ), 0, 0 );
      break;
    case REFC_GOAL_RIGHT:                    // goal right
      if( WM->getSide() == SIDE_RIGHT )
        WM->addOneToGoalDiff();
      else
        WM->subtractOneFromGoalDiff();
      WM->processSeeGlobalInfo( OBJECT_BALL, time, VecPosition( 0, 0 ),
          VecPosition( 0, 0 ), 0, 0 );
      break;
    case REFC_GOALIE_CATCH_BALL_LEFT:        // catch ball
    case REFC_GOALIE_CATCH_BALL_RIGHT:
      WM->processCatchedBall( rm, time );
      break;
    case REFC_PENALTY_ONFIELD_LEFT:
      WM->setSidePenalty( SIDE_LEFT );
      break;
    case REFC_PENALTY_ONFIELD_RIGHT:
      WM->setSidePenalty( SIDE_RIGHT );
      break;
    case REFC_PENALTY_MISS_LEFT:
    case REFC_PENALTY_SCORE_LEFT:
      WM->setPlayMode( PM_FROZEN );
      break;
    case REFC_PENALTY_MISS_RIGHT:
    case REFC_PENALTY_SCORE_RIGHT:
      WM->setPlayMode( PM_FROZEN );
      break;
    case REFC_PENALTY_FOUL_LEFT:
    case REFC_PENALTY_FOUL_RIGHT:
    default:
      break;
   }
   break;
  case 'o':                                  // online_coach_
    analyzeCoachMessage( strMsg );
    break;
  case 's':                                  // self
   break;                                    // do nothing
  default:                                   // from direction
    Log.logWithTime( 600, "incoming hear: %s", WM->strLastHearMessage );
    analyzePlayerMessage( iTime, strMsg );    // from player
```

```
      break;
    }

    return true;
}

/*! This message analyzes an incoming communication message. Messages from
    opponents are discarded. First it is checked whether the message arrived
    from a teammate using a specific encoding string and then the contents
    are parsed and stored in the world model, which will process it when it
    updates the world model.
    \param iTime time from the message */
bool SenseHandler::analyzePlayerMessage( int iTime, char *strMsg )
{
  Parse::gotoFirstNonSpace( &strMsg );          // skip space

  if( WM->getPlayerNumber() == 0 )              // if i am coach
    return false;                               //   skip message
  if( strlen( strMsg ) < 2 || strMsg[0] == 'o' )  // skip message since no dir.
    return false;                               // thus no content

  Parse::parseFirstInt( &strMsg );              // skip direction
  Parse::gotoFirstNonSpace( &strMsg );          // skip space
  if( strlen( strMsg ) < 2 || strMsg[1] == 'p' )  // skip message when from opp
    return false;

  int iPlayer = Parse::parseFirstInt( &strMsg );  // get player number
  Parse::gotoFirstNonSpace( &strMsg );          // skip space
  strMsg++;                                     // skip " (=quote)

  if( strlen( strMsg ) < 4 )                // < 2 + two ending charactres ")
  {
    Log.log( 600, "communication string too small" );
    return false;
  }

  // get the cycle number from the encoding
  int iModCycle = (int)(strMsg[0] - 'a');

  if( iModCycle < 0 || iModCycle > 9 )
  {
    Log.log( 600, "communication cycle nr out of bounds: %d", iModCycle );
    return false;
  }

  // get the time difference between the current time and the send time
```

```
  int iDiff = (iTime % 10) - iModCycle;
  if( iDiff < 0 )
    iDiff += 10;

  // if it is too old; skip parsing; otherwise determine actual send time
  if( iDiff > 2  )
  {
    Log.log( 600, "communication string too old time %d mod %d diff %d",
                  iTime, iModCycle, iDiff);
    return false;
  }
  iTime -= iDiff;

  Log.log( 600, "process comm msg, diff %d time %d, %s", iDiff, iTime, strMsg);
  WM->storePlayerMessage( iPlayer, strMsg, iTime );
  return true;
}

bool SenseHandler::analyzeCoachMessage( char *strMsg )
{
  Log.log( 605, "received coach messages: %s" , strMsg );

  return true;
}
/*! This method analyzes the check_ball message that is only received by the
    coach. It sets the information in the Worldmodel what the status of the
    ball is.
    The format is as follows (check_ball <time> <status>).
    \param strMsg string that contains the check_ball message
    \return bool indicating whether update succeeded. */
bool SenseHandler::analyzeCheckBall( char *strMsg )
{
  WM->setTimeCheckBall( Parse::parseFirstInt( &strMsg ) );
  strMsg++;
  WM->setCheckBallStatus( SoccerTypes::getBallStatusFromStr( strMsg ) );
  return true;
}

/*! This method analyzes the change player type message. This method checks
    whether the player that changed type equals the agent. When this is the
    case, it adjust the ServerSettings according to the values associated
    with this player type.
    \param strMsg string that contains the player type message.
    \return bool indicating whether player type of agent changed. */
bool SenseHandler::analyzeChangePlayerTypeMessage( char *strMsg )
{
```

```
  Log.log( 999, "%s", strMsg );
  int iPlayer = Parse::parseFirstInt( &strMsg );
  if( *strMsg != ')' ) // we are dealing with player of own team
  {
   int      iType = Parse::parseFirstInt( &strMsg );
   ObjectT  obj   = SoccerTypes::getTeammateObjectFromIndex( iPlayer - 1 );
   Log.log( 605, "change player from message %d -> %d", obj, iType );
   WM->setHeteroPlayerType( obj, iType );
   Log.log( 605, "changed player from message %d -> %d", obj,
            WM->getHeteroPlayerType( obj ) );
   return true;
  }
  else
  {
   ObjectT  obj   = SoccerTypes::getOpponentObjectFromIndex( iPlayer - 1 );
   return WM->setSubstitutedOpp( obj );
  }


  return false;
}

/*! This method analyzes the server_param message. This message contains all
    the server parameters. All settings of the ServerSettings are changed
    according to the supplied values. This makes the reading from a server
    configuration file obsolete.
    \param strMsg string message with all the server parameters
    \return booli indicating whether string was parsed. */
bool SenseHandler::analyzeServerParamMessage( char *strMsg )
{

  Log.log( 4, "%s", strMsg );

  readServerParam( "goal_width",              strMsg );
  readServerParam( "player_size",             strMsg );
  readServerParam( "player_decay",            strMsg );
  readServerParam( "player_rand",             strMsg );
  readServerParam( "player_weight",           strMsg );
  readServerParam( "player_speed_max",        strMsg );
  readServerParam( "player_accel_max",        strMsg );
  readServerParam( "stamina_max",             strMsg );
  readServerParam( "stamina_inc_max",         strMsg );
  readServerParam( "recover_dec_thr",         strMsg );
  readServerParam( "recover_min",             strMsg );
  readServerParam( "recover_dec",             strMsg );
  readServerParam( "effort_dec_thr",          strMsg );
```

```
readServerParam( "effort_min",              strMsg );
readServerParam( "effort_dec",              strMsg );
readServerParam( "effort_inc_thr",          strMsg );
readServerParam( "effort_inc",              strMsg );
readServerParam( "kick_rand",               strMsg );
readServerParam( "ball_size",               strMsg );
readServerParam( "ball_decay",              strMsg );
readServerParam( "ball_rand",               strMsg );
readServerParam( "ball_weight",             strMsg );
readServerParam( "ball_speed_max",          strMsg );
readServerParam( "ball_accel_max",          strMsg );
readServerParam( "dash_power_rate",         strMsg );
readServerParam( "kick_power_rate",         strMsg );
readServerParam( "kickable_margin",         strMsg );
readServerParam( "catch_probability",       strMsg );
readServerParam( "catchable_area_l",        strMsg );
readServerParam( "catchable_area_w",        strMsg );
readServerParam( "goalie_max_moves",        strMsg );
readServerParam( "maxpower",                strMsg );
readServerParam( "minpower",                strMsg );
readServerParam( "maxmoment",               strMsg );
readServerParam( "minmoment",               strMsg );
readServerParam( "maxneckmoment",           strMsg );
readServerParam( "minneckmoment",           strMsg );
readServerParam( "maxneckang",              strMsg );
readServerParam( "minneckang",              strMsg );
readServerParam( "visible_angle",           strMsg );
readServerParam( "visible_distance",        strMsg );
readServerParam( "audio_cut_dist",          strMsg );
readServerParam( "quantize_step",           strMsg );
readServerParam( "quantize_step_l",         strMsg );
readServerParam( "ckick_margin",            strMsg );
readServerParam( "wind_dir",                strMsg );
readServerParam( "wind_force",              strMsg );
readServerParam( "wind_rand",               strMsg );
readServerParam( "wind_random",             strMsg );
readServerParam( "inertia_moment",          strMsg );
readServerParam( "half_time",               strMsg );
readServerParam( "drop_ball_time",          strMsg );
readServerParam( "port",                    strMsg );
readServerParam( "coach_port",              strMsg );
readServerParam( "olcoach_port",            strMsg );
readServerParam( "say_coach_cnt_max",       strMsg );
readServerParam( "say_coach_msg_size",      strMsg );
readServerParam( "simulator_step",          strMsg );
readServerParam( "send_step",               strMsg );
```

```
  readServerParam( "recv_step",               strMsg );
  readServerParam( "sense_body_step",         strMsg );
  readServerParam( "say_msg_size",            strMsg );
  readServerParam( "clang_win_size",          strMsg );
  readServerParam( "clang_define_win",        strMsg );
  readServerParam( "clang_meta_win",          strMsg );
  readServerParam( "clang_advice_win",        strMsg );
  readServerParam( "clang_info_win",          strMsg );
  readServerParam( "clang_mess_delay",        strMsg );
  readServerParam( "clang_mess_per_cycle",    strMsg );
  readServerParam( "hear_max",                strMsg );
  readServerParam( "hear_inc",                strMsg );
  readServerParam( "hear_decay",              strMsg );
  readServerParam( "catch_ban_cycle",         strMsg );
  readServerParam( "send_vi_step",            strMsg );
  readServerParam( "use_offside",             strMsg );
  readServerParam( "offside_active_area_size", strMsg );
  readServerParam( "forbid_kick_off_offside", strMsg );
  readServerParam( "verbose",                 strMsg );
  readServerParam( "offside_kick_margin",     strMsg );
  readServerParam( "slow_down_factor",        strMsg );
  readServerParam( "synch_mode",              strMsg );
  readServerParam( "fullstate_l",             strMsg );
  readServerParam( "fullstate_r",             strMsg );
  readServerParam( "pen_dist_x",              strMsg );
  readServerParam( "pen_max_goalie_dist_x",   strMsg );
  readServerParam( "pen_allow_mult_kicks",    strMsg );
  readServerParam( "tackle_dist",             strMsg );
  readServerParam( "tackle_back_dist",        strMsg );
  readServerParam( "tackle_width",            strMsg );
  readServerParam( "tackle_cycles",           strMsg );
  readServerParam( "tackle_power_rate",       strMsg );
  readServerParam( "tackle_exponent",         strMsg );

  SS->setMaximalKickDist      ( SS->getKickableMargin() +
                          SS->getPlayerSize()     +
                          SS->getBallSize()       );
// SS->show( cerr, ":" );
  return true;
 }

 bool SenseHandler::readServerParam( char *strParam, char *strMsg )
 {
  char strFormat[128];
  char strValue[128] = "";
  sprintf( strValue, "none" );
```

```
  sprintf( strFormat, "%s ", strParam );        // add space after parameters
  char *str = strstr( strMsg, strFormat );      // and find param definition
  sprintf( strFormat, "%s %%[^)]", strParam );  // read till closing bracket

  if( str == NULL )
  {
    cerr << "(SenseHandler::readServerParam) " << WM->getPlayerNumber() <<
          "  error finding " << strParam <<endl;
    return false;
  }
  int ret = sscanf( str, strFormat, strValue ); // read in values

  if( ret == 1 )
    SS->setValue( strParam, strValue );
  else
    cerr << "(SenseHandler::readServerParam) error reading " <<strParam <<endl;
  return (ret == 1 ) ? true : false ;
}


/*! This method analyze a player type message. This message contains the
    values associated with a specific heterogeneous player type. The values
    are parsed from the message and supplied to the WorldModel method
    processNewHeteroPlayer.
    \param strMsg string that contains the player type information
    \return bool indicating whether the message was parsed correctly. */
bool SenseHandler::analyzePlayerTypeMessage ( char *strMsg )
{
  Log.log( 999, "%s", strMsg );
// cerr << strMsg << endl;

  // analyze all heterogeneous player information
  int    iIndex          = Parse::parseFirstInt( &strMsg );
  double dPlayerSpeedMax  = Parse::parseFirstDouble( &strMsg );
  double dStaminaIncMax   = Parse::parseFirstDouble( &strMsg );
  double dPlayerDecay     = Parse::parseFirstDouble( &strMsg );
  double dInertiaMoment   = Parse::parseFirstDouble( &strMsg );
  double dDashPowerRate   = Parse::parseFirstDouble( &strMsg );
  double dPlayerSize      = Parse::parseFirstDouble( &strMsg );
  double dKickableMargin  = Parse::parseFirstDouble( &strMsg );
  double dKickRand        = Parse::parseFirstDouble( &strMsg );
  double dExtraStamina    = Parse::parseFirstDouble( &strMsg );
  double dEffortMax       = Parse::parseFirstDouble( &strMsg );
  double dEffortMin       = Parse::parseFirstDouble( &strMsg );
```

```
  WM->processNewHeteroPlayer( iIndex, dPlayerSpeedMax, dStaminaIncMax,
     dPlayerDecay,    dInertiaMoment, dDashPowerRate, dPlayerSize,
     dKickableMargin, dKickRand,      dExtraStamina,  dEffortMax,
     dEffortMin );
  return true;
}

/*! This method analyzes the player_param message that indicates the ranges of
    the possible values for the heterogeneous player types. Nothing is done
    with this information.
    \param strMsg string that contains the player_param message.
    \bool will always be true. */
bool SenseHandler::analyzePlayerParamMessage( char *strMsg )
{
//  cout << strMsg << endl;
  Log.log( 999, "%s", strMsg );
  readServerParam( "player_types",                   strMsg );
  readServerParam( "subs_max",                       strMsg );
  readServerParam( "player_speed_max_delta_min",       strMsg );
  readServerParam( "player_speed_max_delta_max",       strMsg );
  readServerParam( "stamina_inc_max_delta_factor",     strMsg );
  readServerParam( "player_decay_delta_min",           strMsg );
  readServerParam( "player_decay_delta_max",           strMsg );
  readServerParam( "inertia_moment_delta_factor",      strMsg );
  readServerParam( "dash_power_rate_delta_min",        strMsg );
  readServerParam( "dash_power_rate_delta_max",        strMsg );
  readServerParam( "player_size_delta_factor",         strMsg );
  readServerParam( "kickable_margin_delta_min",        strMsg );
  readServerParam( "kickable_margin_delta_max",        strMsg );
  readServerParam( "kick_rand_delta_factor",           strMsg );
  readServerParam( "extra_stamina_delta_min",          strMsg );
  readServerParam( "extra_stamina_delta_max",          strMsg );
  readServerParam( "effort_max_delta_factor",          strMsg );
  readServerParam( "effort_min_delta_factor",          strMsg );
  readServerParam( "new_dash_power_rate_delta_min",    strMsg );
  readServerParam( "new_dash_power_rate_delta_max",    strMsg );
  readServerParam( "new_stamina_inc_max_delta_factor", strMsg );

  return true;
}

/*************************************************************************
/
/********************* TESTING PURPOSES
**************************************/
```

```
/*************************************************************************
/

/*
int main( void )
{
  Connection c( "localhost", 6000 );
  WorldModel wm;
  SenseHandler i( &c,  &wm );
  i.analyzeMessage( "(see 0 ((g r) 64.1 13) ((f r t) 65.4 -16) ((f r b) 79 38) ((f p r t) 46.1 -6) ((f p r
c) 48.4 18) ((f p r b) 58 37) ((f g r t) 62.8 7) ((f g r b) 66 19) ((f t r 20) 38.5 -38) ((f t r 30) 46.5 -
30) ((f t r 40) 55.7 -25) ((f t r 50) 64.7 -21) ((f b r 50) 80.6 41) ((f r t 30) 69.4 -12) ((f r t 20) 67.4
-4) ((f r t 10) 67.4 4) ((f r 0) 69.4 12) ((f r b 10) 72.2 20) ((f r b 20) 75.9 27) ((f r b 30) 81.5 33)
((l r) 62.8 -89))" );
   cout << "2" << endl;
  i.analyzeMessage( "(see 0 ((g l) 49.9 -24) ((f l t) 50.9 14) ((f p l t) 31.5 1 0 0) ((f p l c) 34.5 -33)
((f g l t) 47.9 -17) ((f g l b) 52.5 -32) ((f t l 50) 50.9 20) ((f t l 40) 42.5 26) ((f t l 30) 34.8 36) ((f l
t 30) 54.6 8) ((f l t 20) 53 -2) ((f l t 10) 53 -12) ((f l 0) 54.6 -23) ((f l b 10) 58 -32) ((f l b 20) 62.8
-41) ((p \"l\" 2) 5 -7 0 0 172 172) ((l l) 47.9 82))" );
  c.disconnect();
  cout << "exit" << endl ;

}

*/
```

Geometry.h

```
#ifndef _GEOMETRY_
#define _GEOMETRY_

#include "math.h"       // needed for M_PI constant
#include <string>       // needed for string
#include <iostream>

using namespace std;

typedef double AngRad;  /*!< Type definition for angles in degrees. */
typedef double AngDeg;  /*!< Type definition for angles in radians. */

#define EPSILON 0.0001  /*!< Value used for floating point equality tests. */

// auxiliary numeric functions for determining the
// maximum and minimum of two given double values and the sign of a value
double max     ( double d1, double d2 );
double min     ( double d1, double d2 );
int    sign    ( double d1            );

// auxiliary goniometric functions which enable you to
// specify angles in degrees rather than in radians
AngDeg Rad2Deg ( AngRad x             );
AngRad Deg2Rad ( AngDeg x             );
double cosDeg  ( AngDeg x             );
double sinDeg  ( AngDeg x             );
double tanDeg  ( AngDeg x             );
AngDeg atanDeg ( double x             );
double atan2Deg( double x,  double y  );
AngDeg acosDeg ( double x             );
AngDeg asinDeg ( double x             );

// various goniometric functions
bool   isAngInInterval    ( AngDeg ang,    AngDeg angMin,    AngDeg angMax );
AngDeg getBisectorTwoAngles( AngDeg angMin, AngDeg angMax );

/*! CoordSystem is an enumeration of the different specified
    coordinate systems.  The two possibilities are CARTESIAN or
    POLAR. These values are for instance used in the initializing a
    VecPosition. The CoordSystem indicates whether the supplied
    arguments represent the position in cartesian or in polar
    coordinates. */
enum CoordSystemT {
  CARTESIAN,
  POLAR
};
```

```
/*************************************************************************
/
/*******************  CLASS VECPOSITION  ********************************/
/*************************************************************************
/

/*! This class contains an x- and y-coordinate of a position (x,y) as
    member data and methods which operate on this position. The
    standard arithmetic operators are overloaded and can thus be
    applied to positions (x,y). It is also possible to represent a
    position in polar coordinates (r,phi), since the class contains a
    method to convert these into cartesian coordinates (x,y). */
class VecPosition
{
  // private member data
private:

  double m_x;   /*!< x-coordinate of this position */
  double m_y;   /*!< y-coordinate of this position */

  // public methods
public:
  // constructor for VecPosition class
  VecPosition                ( double          vx = 0,
                               double          vy = 0,
                               CoordSystemT    cs =CARTESIAN);

  // overloaded arithmetic operators
  VecPosition      operator -        (                          );
  VecPosition      operator +        ( const double      &d       );
  VecPosition      operator +        ( const VecPosition &p       );
  VecPosition      operator -        ( const double      &d       );
  VecPosition      operator -        ( const VecPosition &p       );
  VecPosition      operator *        ( const double      &d       );
  VecPosition      operator *        ( const VecPosition &p       );
  VecPosition      operator /        ( const double      &d       );
  VecPosition      operator /        ( const VecPosition &p       );
  void             operator =        ( const double      &d       );
  void             operator +=       ( const VecPosition &p       );
  void             operator +=       ( const double      &d       );
  void             operator -=       ( const VecPosition &p       );
  void             operator -=       ( const double      &d       );
  void             operator *=       ( const VecPosition &p       );
  void             operator *=       ( const double      &d       );
  void             operator /=       ( const VecPosition &p       );
```

```
void        operator /=     ( const double      &d          );
bool        operator !=     ( const VecPosition &p          );
bool        operator !=     ( const double      &d          );
bool        operator ==     ( const VecPosition &p          );
bool        operator ==     ( const double      &d          );

// methods for producing output
friend ostream&  operator <<    ( ostream          &os,
                                  VecPosition      p          );
void        show                ( CoordSystemT     cs =CARTESIAN);
string      str                 ( CoordSystemT     cs =CARTESIAN);

// set- and get methods for private member variables
bool        setX                ( double           dX         );
double      getX                (                     ) const;
bool        setY                ( double           dY         );
double      getY                (                     ) const;

// set- and get methods for derived position information
void        setVecPosition      ( double           dX = 0,
                                  double           dY = 0,
                                  CoordSystemT     cs =CARTESIAN);
double      getDistanceTo       ( const VecPosition p          );
VecPosition setMagnitude        ( double           d          );
double      getMagnitude        (                     ) const;
AngDeg      getDirection        (                     ) const;

// comparison methods for positions
bool        isInFrontOf         ( const VecPosition &p         );
bool        isInFrontOf         ( const double      &d         );
bool        isBehindOf          ( const VecPosition &p         );
bool        isBehindOf          ( const double      &d         );
bool        isLeftOf            ( const VecPosition &p         );
bool        isLeftOf            ( const double      &d         );
bool        isRightOf           ( const VecPosition &p         );
bool        isRightOf           ( const double      &d         );
bool        isBetweenX          ( const VecPosition &p1,
                                  const VecPosition &p2        );
bool        isBetweenX          ( const double      &d1,
                                  const double      &d2        );
bool        isBetweenY          ( const VecPosition &p1,
                                  const VecPosition &p2        );
bool        isBetweenY          ( const double      &d1,
                                  const double      &d2        );

// conversion methods for positions
```

```
  VecPosition       normalize           (                       );
  VecPosition       rotate              ( AngDeg          angle       );
  VecPosition       globalToRelative      ( VecPosition       orig,
                                AngDeg          ang         );
  VecPosition       relativeToGlobal      ( VecPosition       orig,
                                AngDeg          ang         );
  VecPosition       getVecPositionOnLineFraction( VecPosition  &p,
                                double          dFrac       );

  // static class methods
  static VecPosition getVecPositionFromPolar( double          dMag,
                                AngDeg          ang         );
  static AngDeg     normalizeAngle        ( AngDeg          angle       );
};

/*************************************************************************
/
/*********************   CLASS GEOMETRY
**************************************/
/*************************************************************************
/

/*! This class contains several static methods dealing with geometry.*/
class Geometry
{

public:

  // geometric series
  static double getLengthGeomSeries(double dFirst,double dRatio,double dSum  );
  static double getSumGeomSeries   (double dFirst,double dRatio,double dLen  );
  static double getSumInfGeomSeries(double dFirst,double dRatio              );
  static double getFirstGeomSeries (double dSum,  double dRatio,double dLen  );
  static double getFirstInfGeomSeries(double dSum,double dRatio              );

  // abc formula
  static int    abcFormula(double a,double b, double c, double *s1,double *s2);
};

/*************************************************************************
/
/*********************** CLASS CIRCLE
*****************************************/
/*************************************************************************
/
```

```
/*!This class represents a circle. A circle is defined by one VecPosition
  (which denotes the center) and its radius. */
class Circle
{
   VecPosition m_posCenter;          /*!< Center of the circle  */
   double      m_dRadius;            /*!< Radius of the circle  */

public:
   Circle( );
   Circle( VecPosition pos, double dR );

   void        show                ( ostream& os = cout );

   // get and set methods
   bool        setCircle           ( VecPosition pos,
                                     double      dR  );
   bool        setRadius           ( double dR       );
   double      getRadius           (                 );
   bool        setCenter           ( VecPosition pos );
   VecPosition getCenter           (                 );
   double      getCircumference    (                 );
   double      getArea             (                 );

   // calculate intersection points and area with other circle
   bool        isInside            ( VecPosition pos );
   int         getIntersectionPoints ( Circle      c,
                                       VecPosition *p1,
                                       VecPosition *p2 );
   double      getIntersectionArea   ( Circle c        );


} ;

/*****************************************************************************
/
/********************** CLASS LINE
*****************************************/
/*****************************************************************************
/

/*!This class contains the representation of a line. A line is defined
  by the formula ay + bx + c = 0. The coefficients a, b and c are stored
  and used in the calculations. */
class Line
{
 // a line is defined by the formula: ay + bx + c = 0
```

```
  double m_a; /*!< This is the a coefficient in the line ay + bx + c = 0 */
  double m_b; /*!< This is the b coefficient in the line ay + bx + c = 0 */
  double m_c; /*!< This is the c coefficient in the line ay + bx + c = 0 */

public:
  Line( double a, double b, double c );

  // print methods
  void       show( ostream& os = cout );
  friend     ostream& operator << (ostream & os, Line l);

  // get intersection points with this line
  VecPosition getIntersection          ( Line       line               );
  int        getCircleIntersectionPoints( Circle     circle,
                                VecPosition *posSolution1,
                                VecPosition *posSolution2        );
  Line       getTangentLine            ( VecPosition pos               );
  VecPosition getPointOnLineClosestTo    ( VecPosition pos               );
  double     getDistanceWithPoint      ( VecPosition pos               );
  bool       isInBetween               ( VecPosition pos,
                                VecPosition point1,
                                VecPosition point2              );

  // calculate associated variables in the line
  double     getYGivenX                ( double     x );
  double     getXGivenY                ( double     y );
  double     getACoefficient           (             ) const;
  double     getBCoefficient           (             ) const;
  double     getCCoefficient           (             ) const;

  // static methods to make a line using an easier representation.
  static Line makeLineFromTwoPoints     ( VecPosition pos1,
                                VecPosition pos2               );
  static Line makeLineFromPositionAndAngle( VecPosition vec,
                                AngDeg angle                   );
};

/*****************************************************************************
/
/********************** CLASS RECTANGLE
*************************************/
/*****************************************************************************
*/
/*!This class represents a rectangle. A rectangle is defined by two
   VecPositions the one at the upper left corner and the one at the
   right bottom. */
```

```
class Rect
{
  VecPosition m_posLeftTop;     /*!< top left position of the rectangle      */
  VecPosition m_posRightBottom; /*!< bottom right position of the rectangle  */

public:
  Rect                    ( VecPosition pos, VecPosition pos2 );

  void        show        ( ostream& os = cout              );

  // checks whether point lies inside the rectangle
  bool        isInside    ( VecPosition pos                 );

  // standard get and set methosd
  void        setRectanglePoints( VecPosition pos1,
                                  VecPosition pos2            );
  bool        setPosLeftTop     ( VecPosition pos             );
  VecPosition getPosLeftTop     (                             );
  bool        setPosRightBottom ( VecPosition pos             );
  VecPosition getPosRightBottom (                             );
};

#endif
```

Geometry.cpp

```
#include "Geometry.h"
#include <stdio.h>    // needed for sprintf

/*! This function returns the sign of a give double.
    1 is positive, -1 is negative
    \param d1 first parameter
    \return the sign of this double */
int sign( double d1 )
{
  return (d1>0)?1:-1;
}

/*! This function returns the maximum of two given doubles.
    \param d1 first parameter
    \param d2 second parameter
    \return the maximum of these two parameters */
double max( double d1, double d2 )
{
  return (d1>d2)?d1:d2;
}

/*! This function returns the minimum of two given doubles.
    \param d1 first parameter
    \param d2 second parameter
    \return the minimum of these two parameters */
double min( double d1, double d2 )
{
  return (d1<d2)?d1:d2;
}


/*! This function converts an angle in radians to the corresponding angle in
    degrees.
    \param x an angle in radians
    \return the corresponding angle in degrees */
AngDeg Rad2Deg( AngRad x )
{
  return ( x * 180 / M_PI );
}

/*! This function converts an angle in degrees to the corresponding angle in
    radians.
    \param x an angle in degrees
    \return the corresponding angle in radians */
AngRad Deg2Rad( AngDeg x )
{
```

```
  return ( x * M_PI / 180 );
}

/*! This function returns the cosine of a given angle in degrees using the
    built-in cosine function that works with angles in radians.
    \param x an angle in degrees
    \return the cosine of the given angle */
double cosDeg( AngDeg x )
{
  return ( cos( Deg2Rad( x ) ) );
}

/*! This function returns the sine of a given angle in degrees using the
    built-in sine function that works with angles in radians.
    \param x an angle in degrees
    \return the sine of the given angle */
double sinDeg( AngDeg x )
{
  return ( sin( Deg2Rad( x ) ) );
}

/*! This function returns the tangent of a given angle in degrees using the
    built-in tangent function that works with angles in radians.
    \param x an angle in degrees
    \return the tangent of the given angle */
double tanDeg( AngDeg x )
{
  return ( tan( Deg2Rad( x ) ) );
}

/*! This function returns the principal value of the arc tangent of x
    in degrees using the built-in arc tangent function which returns
    this value in radians.
    \param x a double value
    \return the arc tangent of the given value in degrees */
AngDeg atanDeg( double x )
{
  return ( Rad2Deg( atan( x ) ) );
}

/*! This function returns the principal value of the arc tangent of y/x in
    degrees using the signs of both arguments to determine the quadrant of the
    return value. For this the built-in 'atan2' function is used which returns
    this value in radians.
    \param x a double value
    \param y a double value
```

```
    \return the arc tangent of y/x in degrees taking the signs of x and y into
    account */
double atan2Deg( double x, double y )
{
  if( fabs( x ) < EPSILON && fabs( y ) < EPSILON )
    return ( 0.0 );

  return ( Rad2Deg( atan2( x, y ) ) );
}

/*! This function returns the principal value of the arc cosine of x in degrees
    using the built-in arc cosine function which returns this value in radians.
    \param x a double value
    \return the arc cosine of the given value in degrees */
AngDeg acosDeg( double x )
{
  if( x >= 1 )
    return ( 0.0 );
  else if( x <= -1 )
    return ( 180.0 );

  return ( Rad2Deg( acos( x ) ) );
}

/*! This function returns the principal value of the arc sine of x in degrees
    using the built-in arc sine function which returns this value in radians.
    \param x a double value
    \return the arc sine of the given value in degrees */
AngDeg asinDeg( double x )
{
  if( x >= 1 )
    return ( 90.0 );
  else if ( x <= -1 )
    return ( -90.0 );

  return ( Rad2Deg( asin( x ) ) );
}

/*! This function returns a boolean value which indicates whether the value
   'ang' (from interval [-180..180] lies in the interval [angMin..angMax].
    Examples: isAngInInterval( -100, 4, -150) returns false
              isAngInInterval(   45, 4, -150) returns true
    \param ang angle that should be checked
    \param angMin minimum angle in interval
    \param angMax maximum angle in interval
    \return boolean indicating whether ang lies in [angMin..angMax] */
```

```
bool isAngInInterval( AngDeg ang, AngDeg angMin, AngDeg angMax )
{
 // convert all angles to interval 0..360
 if( ( ang    + 360 ) < 360 ) ang    += 360;
 if( ( angMin + 360 ) < 360 ) angMin += 360;
 if( ( angMax + 360 ) < 360 ) angMax += 360;

 if( angMin < angMax ) // 0 ---false-- angMin ---true-----angMax---false--360
  return angMin < ang && ang < angMax ;
 else                  // 0 ---true--- angMax ---false----angMin---true---360
  return !( angMax < ang && ang < angMin );
}

/*! This method returns the bisector (average) of two angles. It deals
   with the boundary problem, thus when 'angMin' equals 170 and 'angMax'
   equals -100, -145 is returned.
   \param angMin minimum angle [-180,180]
   \param angMax maximum angle [-180,180]
   \return average of angMin and angMax. */
AngDeg getBisectorTwoAngles( AngDeg angMin, AngDeg angMax )
{
 // separate sine and cosine part to circumvent boundary problem
 return VecPosition::normalizeAngle(
         atan2Deg( (sinDeg( angMin) + sinDeg( angMax ) )/2.0,
                 (cosDeg( angMin) + cosDeg( angMax ) )/2.0 ) );
}

/*****************************************************************************
/
/********************   CLASS VECPOSITION   ***********************************/
/*****************************************************************************
/

/*! Constructor for the VecPosition class. When the supplied
   Coordinate System type equals CARTESIAN, the arguments x and y
   denote the x- and y-coordinates of the new position. When it
   equals POLAR however, the arguments x and y denote the polar
   coordinates of the new position; in this case x is thus equal to
   the distance r from the origin and y is equal to the angle phi
   that the polar vector makes with the x-axis.
   \param x the x-coordinate of the new position when cs == CARTESIAN; the
   distance of the new position from the origin when cs = POLAR
   \param y the y-coordinate of the new position when cs = CARTESIAN; the
   angle that the polar vector makes with the x-axis when cs = POLAR
   \param cs a CoordSystemT indicating whether x and y denote cartesian
   coordinates or polar coordinates
```

```
    \return the VecPosition corresponding to the given arguments */
VecPosition::VecPosition( double x, double y, CoordSystemT cs )
{
  setVecPosition( x, y, cs );
}

/*! Overloaded version of unary minus operator for VecPositions. It returns the
    negative VecPosition, i.e. both the x- and y-coordinates are multiplied by
    -1. The current VecPosition itself is left unchanged.
    \return a negated version of the current VecPosition */
VecPosition VecPosition::operator - ( )
{
  return ( VecPosition( -m_x, -m_y ) );
}

/*! Overloaded version of the binary plus operator for adding a given double
    value to a VecPosition. The double value is added to both the x- and
    y-coordinates of the current VecPosition. The current VecPosition itself is
    left unchanged.
    \param d a double value which has to be added to both the x- and
    y-coordinates of the current VecPosition
    \return the result of adding the given double value to the current
    VecPosition */
VecPosition VecPosition::operator + ( const double &d )
{
  return ( VecPosition( m_x + d, m_y + d ) );
}

/*! Overloaded version of the binary plus operator for VecPositions. It returns
    the sum of the current VecPosition and the given VecPosition by adding their
    x- and y-coordinates. The VecPositions themselves are left unchanged.
    \param p a VecPosition
    \return the sum of the current VecPosition and the given VecPosition */
VecPosition VecPosition::operator + ( const VecPosition &p )
{
  return ( VecPosition( m_x + p.m_x, m_y + p.m_y ) );
}

/*! Overloaded version of the binary minus operator for subtracting a
    given double value from a VecPosition. The double value is
    subtracted from both the x- and y-coordinates of the current
    VecPosition. The current VecPosition itself is left unchanged.
    \param d a double value which has to be subtracted from both the x- and
    y-coordinates of the current VecPosition
    \return the result of subtracting the given double value from the current
    VecPosition */
```

```
VecPosition VecPosition::operator - ( const double &d )
{
  return ( VecPosition( m_x - d, m_y - d ) );
}

/*! Overloaded version of the binary minus operator for
    VecPositions. It returns the difference between the current
    VecPosition and the given VecPosition by subtracting their x- and
    y-coordinates. The VecPositions themselves are left unchanged.

    \param p a VecPosition
    \return the difference between the current VecPosition and the given
    VecPosition */
VecPosition VecPosition::operator - ( const VecPosition &p )
{
  return ( VecPosition( m_x - p.m_x, m_y - p.m_y ) );
}

/*! Overloaded version of the multiplication operator for multiplying a
    VecPosition by a given double value. Both the x- and y-coordinates of the
    current VecPosition are multiplied by this value. The current VecPosition
    itself is left unchanged.
    \param d the multiplication factor
    \return the result of multiplying the current VecPosition by the given
    double value */
VecPosition VecPosition::operator * ( const double &d  )
{
  return ( VecPosition( m_x * d, m_y * d  ) );
}

/*! Overloaded version of the multiplication operator for
    VecPositions. It returns the product of the current VecPosition
    and the given VecPosition by multiplying their x- and
    y-coordinates. The VecPositions themselves are left unchanged.

    \param p a VecPosition
    \return the product of the current VecPosition and the given VecPosition */
VecPosition VecPosition::operator * ( const VecPosition &p )
{
  return ( VecPosition( m_x * p.m_x, m_y * p.m_y ) );
}

/*! Overloaded version of the division operator for dividing a
    VecPosition by a given double value. Both the x- and y-coordinates
    of the current VecPosition are divided by this value. The current
    VecPosition itself is left unchanged.
```

```
    \param d the division factor
    \return the result of dividing the current VecPosition by the given double
    value */
VecPosition VecPosition::operator / ( const double &d )
{
  return ( VecPosition( m_x / d, m_y / d  ) );
}

/*! Overloaded version of the division operator for VecPositions. It
    returns the quotient of the current VecPosition and the given
    VecPosition by dividing their x- and y-coordinates. The
    VecPositions themselves are left unchanged.

    \param p a VecPosition
    \return the quotient of the current VecPosition and the given one */
VecPosition VecPosition::operator / ( const VecPosition &p )
{
  return ( VecPosition( m_x / p.m_x, m_y / p.m_y ) );
}

/*! Overloaded version of the assignment operator for assigning a given double
    value to both the x- and y-coordinates of the current VecPosition. This
    changes the current VecPosition itself.
    \param d a double value which has to be assigned to both the x- and
    y-coordinates of the current VecPosition */
void VecPosition::operator = ( const double &d )
{
  m_x = d;
  m_y = d;
}

/*! Overloaded version of the sum-assignment operator for VecPositions. It
    returns the sum of the current VecPosition and the given VecPosition by
    adding their x- and y-coordinates. This changes the current VecPosition
    itself.
    \param p a VecPosition which has to be added to the current VecPosition */
void VecPosition::operator +=( const VecPosition &p )
{
  m_x += p.m_x;
  m_y += p.m_y;
}

/*! Overloaded version of the sum-assignment operator for adding a given double
    value to a VecPosition. The double value is added to both the x- and
    y-coordinates of the current VecPosition. This changes the current
```

```
    VecPosition itself.
    \param d a double value which has to be added to both the x- and
    y-coordinates of the current VecPosition */
void VecPosition::operator += ( const double &d )
{
  m_x += d;
  m_y += d;
}

/*! Overloaded version of the difference-assignment operator for
    VecPositions.  It returns the difference between the current
    VecPosition and the given VecPosition by subtracting their x- and
    y-coordinates. This changes the current VecPosition itself.

    \param p a VecPosition which has to be subtracted from the current
    VecPosition */
void VecPosition::operator -=( const VecPosition &p )
{
  m_x -= p.m_x;
  m_y -= p.m_y;
}

/*! Overloaded version of the difference-assignment operator for
    subtracting a given double value from a VecPosition. The double
    value is subtracted from both the x- and y-coordinates of the
    current VecPosition. This changes the current VecPosition itself.

    \param d a double value which has to be subtracted from both the x- and
    y-coordinates of the current VecPosition */
void VecPosition::operator -=( const double &d )
{
  m_x -= d;
  m_y -= d;
}

/*! Overloaded version of the multiplication-assignment operator for
    VecPositions. It returns the product of the current VecPosition
    and the given VecPosition by multiplying their x- and
    y-coordinates. This changes the current VecPosition itself.

    \param p a VecPosition by which the current VecPosition has to be
    multiplied */
void VecPosition::operator *=( const VecPosition &p )
{
  m_x *= p.m_x;
  m_y *= p.m_y;
```

```
}

/*! Overloaded version of the multiplication-assignment operator for
    multiplying a VecPosition by a given double value. Both the x- and
    y-coordinates of the current VecPosition are multiplied by this
    value. This changes the current VecPosition itself.

    \param d a double value by which both the x- and y-coordinates of the
    current VecPosition have to be multiplied */
void VecPosition::operator *=( const double &d )
{
  m_x *= d;
  m_y *= d;
}

/*! Overloaded version of the division-assignment operator for
    VecPositions. It returns the quotient of the current VecPosition
    and the given VecPosition by dividing their x- and
    y-coordinates. This changes the current VecPosition itself.

    \param p a VecPosition by which the current VecPosition is divided */
void VecPosition::operator /=( const VecPosition &p )
{
  m_x /= p.m_x;
  m_y /= p.m_y;
}

/*! Overloaded version of the division-assignment operator for
    dividing a VecPosition by a given double value. Both the x- and
    y-coordinates of the current VecPosition are divided by this
    value. This changes the current VecPosition itself.

    \param d a double value by which both the x- and y-coordinates of the
    current VecPosition have to be divided */
void VecPosition::operator /=( const double &d )
{
  m_x /= d;
  m_y /= d;
}

/*! Overloaded version of the inequality operator for VecPositions. It
    determines whether the current VecPosition is unequal to the given
    VecPosition by comparing their x- and y-coordinates.

    \param p a VecPosition
    \return true when either the x- or y-coordinates of the given VecPosition
```

```
    and the current VecPosition are different; false otherwise */
bool VecPosition::operator !=( const VecPosition &p )
{
  return ( ( m_x != p.m_x ) || ( m_y != p.m_y ) );
}

/*! Overloaded version of the inequality operator for comparing a
    VecPosition to a double value. It determines whether either the x-
    or y-coordinate of the current VecPosition is unequal to the given
    double value.

    \param d a double value with which both the x- and y-coordinates of the
    current VecPosition have to be compared.
    \return true when either the x- or y-coordinate of the current VecPosition
    is unequal to the given double value; false otherwise */
bool VecPosition::operator !=( const double &d )
{
  return ( ( m_x != d ) || ( m_y != d ) );
}

/*! Overloaded version of the equality operator for VecPositions. It
    determines whether the current VecPosition is equal to the given
    VecPosition by comparing their x- and y-coordinates.

    \param p a VecPosition
    \return true when both the x- and y-coordinates of the given
    VecPosition and the current VecPosition are equal; false
    otherwise */
bool VecPosition::operator ==( const VecPosition &p )
{
  return ( ( m_x == p.m_x ) && ( m_y == p.m_y ) );
}

/*! Overloaded version of the equality operator for comparing a
    VecPosition to a double value. It determines whether both the x-
    and y-coordinates of the current VecPosition are equal to the
    given double value.

    \param d a double value with which both the x- and y-coordinates of the
    current VecPosition have to be compared.
    \return true when both the x- and y-coordinates of the current VecPosition
    are equal to the given double value; false otherwise */
bool VecPosition::operator ==( const double &d )
{
  return ( ( m_x == d ) && ( m_y == d ) );
}
```

```
/*! Overloaded version of the C++ output operator for
    VecPositions. This operator makes it possible to use VecPositions
    in output statements (e.g.  cout << v). The x- and y-coordinates
    of the VecPosition are printed in the format (x,y).

    \param os output stream to which information should be written
    \param v a VecPosition which must be printed
    \return output stream containing (x,y) */
ostream& operator <<( ostream &os, VecPosition v )
{
  return ( os << "( " << v.m_x << ", " << v.m_y << " )" );
}

/*! This method writes the current VecPosition to standard output. It
    can also print a polar representation of the current VecPosition.

    \param cs a CoordSystemtT indicating whether a POLAR or CARTESIAN
     representation of the current VecPosition should be printed */
void VecPosition::show( CoordSystemT cs )
{
  if( cs == CARTESIAN )
    cout << *this << endl;
  else
    cout << "( r: " << getMagnitude( ) << ", phi: " << getDirection( ) << "  )";
}

/*! This method writes the current VecPosition to a string. It can
    also write a polar representation of the current VecPosition.

    \param cs a CoordSystemtT indicating whether a POLAR or CARTESIAN
     representation of the current VecPosition should be written
    \return a string containing a polar or Cartesian representation of the
    current VecPosition depending on the value of the boolean argument */
string VecPosition::str( CoordSystemT cs )
{
  char buf[ 1024 ];

  if( cs == CARTESIAN )
    sprintf( buf, "( %f, %f )", getX( ), getY( ) );
  else
    sprintf( buf, "( r: %f, phi: %f )", getMagnitude( ), getDirection( ) );

  string str( buf );
  return ( str );
}
```

```
/*! Set method for the x-coordinate of the current VecPosition.

    \param dX a double value representing a new x-coordinate
    \return a boolean indicating whether the update was successful */
bool VecPosition::setX( double dX )
{
  m_x = dX;
  return ( true );
}

/*! Get method for the x-coordinate of the current VecPosition.

    \return the x-coordinate of the current VecPosition */
double VecPosition::getX( ) const
{
  return ( m_x );
}

/*! Set method for the y-coordinate of the current VecPosition.

    \param dY a double value representing a new y-coordinate
    \return a boolean indicating whether the update was successful */
bool VecPosition::setY( double dY )
{
  m_y = dY;
  return ( true );
}

/*! Get method for the y-coordinate of the current VecPosition.

    \return the y-coordinate of the current VecPosition */
double VecPosition::getY( ) const
{
  return ( m_y );
}

/*! This method (re)sets the coordinates of the current
    VecPosition. The given coordinates can either be polar or
    Cartesian coordinates. This is indicated by the value of the third
    argument.

    \param dX a double value indicating either a new Cartesian
    x-coordinate when cs=CARTESIAN or a new polar r-coordinate
    (distance) when cs=POLAR
```

```
    \param dY a double value indicating either a new Cartesian
    y-coordinate when cs=CARTESIAN or a new polar phi-coordinate
    (angle) when cs=POLAR

    \param cs a CoordSystemT indicating whether x and y denote
    cartesian coordinates or polar coordinates */
void VecPosition::setVecPosition( double dX, double dY, CoordSystemT cs)
{
  if( cs == CARTESIAN )
  {
    m_x = dX;
    m_y = dY;
  }
  else
    *this = getVecPositionFromPolar( dX, dY );
}

/*! This method determines the distance between the current
    VecPosition and a given VecPosition. This is equal to the
    magnitude (length) of the vector connecting the two positions
    which is the difference vector between them.

    \param p a Vecposition
    \return the distance between the current VecPosition and the given
    VecPosition */
double VecPosition::getDistanceTo( const VecPosition p )
{
  return ( ( *this - p ).getMagnitude( ) );
}

/*! This method adjusts the coordinates of the current VecPosition in
    such a way that the magnitude of the corresponding vector equals
    the double value which is supplied as an argument. It thus scales
    the vector to a given length by multiplying both the x- and
    y-coordinates by the quotient of the argument and the current
    magnitude. This changes the VecPosition itself.

    \param d a double value representing a new magnitude

    \return the result of scaling the vector corresponding with the
    current VecPosition to the given magnitude thus yielding a
    different VecPosition */
VecPosition VecPosition::setMagnitude( double d )
{
  if( getMagnitude( ) > EPSILON )
    ( *this ) *= ( d / getMagnitude( ) );
```

```
  return ( *this );
}

/*! This method determines the magnitude (length) of the vector
    corresponding with the current VecPosition using the formula of
    Pythagoras.

    \return the length of the vector corresponding with the current
    VecPosition */
double VecPosition::getMagnitude( ) const
{
  return ( sqrt( m_x * m_x + m_y * m_y ) );
}

/*! This method determines the direction of the vector corresponding
    with the current VecPosition (the phi-coordinate in polar
    representation) using the arc tangent function. Note that the
    signs of x and y have to be taken into account in order to
    determine the correct quadrant.

    \return the direction in degrees of the vector corresponding with
    the current VecPosition */
AngDeg VecPosition::getDirection( ) const
{
  return ( atan2Deg( m_y, m_x ) );
}

/*! This method determines whether the current VecPosition is in front
    of a given VecPosition, i.e. whether the x-coordinate of the
    current VecPosition is larger than the x-coordinate of the given
    VecPosition.

    \param p a VecPosition to which the current VecPosition must be compared
    \return true when the current VecPosition is in front of the given
    VecPosition; false otherwise */
bool VecPosition::isInFrontOf( const VecPosition &p )
{
  return ( ( m_x > p.getX( ) ) ? true : false );
}

/*! This method determines whether the x-coordinate of the current
    VecPosition is in front of (i.e. larger than) a given double
    value.

    \param d a double value to which the current x-coordinate must be
```

```
    compared

    \return true when the current x-coordinate is in front of the
    given value; false otherwise */
bool VecPosition::isInFrontOf( const double &d )
{
  return ( ( m_x > d ) ? true : false );
}

/*! This method determines whether the current VecPosition is behind a
    given VecPosition, i.e. whether the x-coordinate of the current
    VecPosition is smaller than the x-coordinate of the given
    VecPosition.

    \param p a VecPosition to which the current VecPosition must be
    compared

    \return true when the current VecPosition is behind the given
    VecPosition; false otherwise */
bool VecPosition::isBehindOf( const VecPosition &p )
{
  return ( ( m_x < p.getX( ) ) ? true : false );
}

/*! This method determines whether the x-coordinate of the current
    VecPosition is behind (i.e. smaller than) a given double value.

    \param d a double value to which the current x-coordinate must be
    compared

    \return true when the current x-coordinate is behind the given
    value; false otherwise */
bool VecPosition::isBehindOf( const double &d )
{
  return ( ( m_x < d ) ? true : false );
}

/*! This method determines whether the current VecPosition is to the
    left of a given VecPosition, i.e. whether the y-coordinate of the
    current VecPosition is smaller than the y-coordinate of the given
    VecPosition.

    \param p a VecPosition to which the current VecPosition must be
    compared

    \return true when the current VecPosition is to the left of the
```

```
    given VecPosition; false otherwise */
bool VecPosition::isLeftOf( const VecPosition &p )
{
  return ( ( m_y < p.getY( ) ) ? true : false );
}

/*! This method determines whether the y-coordinate of the current
    VecPosition is to the left of (i.e. smaller than) a given double
    value.

    \param d a double value to which the current y-coordinate must be
    compared

    \return true when the current y-coordinate is to the left of the
    given value; false otherwise */
bool VecPosition::isLeftOf( const double &d )
{
  return ( ( m_y < d ) ? true : false );
}

/*! This method determines whether the current VecPosition is to the
    right of a given VecPosition, i.e. whether the y-coordinate of the
    current VecPosition is larger than the y-coordinate of the given
    VecPosition.

    \param p a VecPosition to which the current VecPosition must be
    compared

    \return true when the current VecPosition is to the right of the
    given VecPosition; false otherwise */
bool VecPosition::isRightOf( const VecPosition &p )
{
  return ( ( m_y > p.getY( ) ) ? true : false );
}

/*! This method determines whether the y-coordinate of the current
    VecPosition is to the right of (i.e. larger than) a given double
    value.

    \param d a double value to which the current y-coordinate must be
    compared

    \return true when the current y-coordinate is to the right of the
    given value; false otherwise */
bool VecPosition::isRightOf( const double &d )
{
```

```
  return ( ( m_y > d ) ? true : false );
}

/*! This method determines whether the current VecPosition is in
    between two given VecPositions when looking in the x-direction,
    i.e. whether the current VecPosition is in front of the first
    argument and behind the second.

    \param p1 a VecPosition to which the current VecPosition must be
    compared

    \param p2 a VecPosition to which the current VecPosition must be
    compared

    \return true when the current VecPosition is in between the two
    given VecPositions when looking in the x-direction; false
    otherwise */
bool VecPosition::isBetweenX( const VecPosition &p1, const VecPosition &p2 )
{
  return ( ( isInFrontOf( p1 ) && isBehindOf( p2 ) ) ? true : false );
}

/*! This method determines whether the x-coordinate of the current
    VecPosition is in between two given double values, i.e. whether
    the x-coordinate of the current VecPosition is in front of the
    first argument and behind the second.

    \param d1 a double value to which the current x-coordinate must be
    compared

    \param d2 a double value to which the current x-coordinate must be
    compared

    \return true when the current x-coordinate is in between the two
    given values; false otherwise */
bool VecPosition::isBetweenX( const double &d1, const double &d2 )
{
  return ( ( isInFrontOf( d1 ) && isBehindOf( d2 ) ) ? true : false );
}

/*! This method determines whether the current VecPosition is in
    between two given VecPositions when looking in the y-direction,
    i.e. whether the current VecPosition is to the right of the first
    argument and to the left of the second.

    \param p1 a VecPosition to which the current VecPosition must be
```

```
    compared

    \param p2 a VecPosition to which the current VecPosition must be
    compared

    \return true when the current VecPosition is in between the two
    given VecPositions when looking in the y-direction; false
    otherwise */
bool VecPosition::isBetweenY( const VecPosition &p1, const VecPosition &p2 )
{
  return ( ( isRightOf( p1 ) && isLeftOf( p2 ) ) ? true : false );
}

/*! This method determines whether the y-coordinate of the current
    VecPosition is in between two given double values, i.e. whether
    the y-coordinate of the current VecPosition is to the right of the
    first argument and to the left of the second.

    \param d1 a double value to which the current y-coordinate must be
    compared

    \param d2 a double value to which the current y-coordinate must be
    compared

    \return true when the current y-coordinate is in between the two
    given values; false otherwise */
bool VecPosition::isBetweenY( const double &d1, const double &d2 )
{
  return ( ( isRightOf( d1 ) && isLeftOf( d2 ) ) ? true : false );
}

/*! This method normalizes a VecPosition by setting the magnitude of
    the corresponding vector to 1. This thus changes the VecPosition
    itself.

    \return the result of normalizing the current VecPosition thus
    yielding a different VecPosition */
VecPosition VecPosition::normalize( )
{
  return ( setMagnitude( 1.0 ) );
}

/*! This method rotates the vector corresponding to the current
    VecPosition over a given angle thereby changing the current
    VecPosition itself. This is done by calculating the polar
    coordinates of the current VecPosition and adding the given angle
```

```
    to the phi-coordinate in the polar representation. The polar
    coordinates are then converted back to Cartesian coordinates to
    obtain the desired result.

    \param angle an angle in degrees over which the vector
    corresponding to the current VecPosition must be rotated

    \return the result of rotating the vector corresponding to the
    current VecPosition over the given angle thus yielding a different
    VecPosition */
VecPosition VecPosition::rotate( AngDeg angle )
{
  // determine the polar representation of the current VecPosition
  double dMag    = this->getMagnitude( );
  double dNewDir = this->getDirection( ) + angle;  // add rotation angle to phi
  setVecPosition( dMag, dNewDir, POLAR );          // convert back to Cartesian
  return ( *this );
}

/*! This method converts the coordinates of the current VecPosition
    (which are represented in an global coordinate system with the
    origin at (0,0)) into relative coordinates in a different
    coordinate system (e.g. relative to a player). The new coordinate
    system is defined by the arguments to the method. The relative
    coordinates are now obtained by aligning the relative coordinate
    system with the global coordinate system using a translation to
    make both origins coincide followed by a rotation to align the
    axes.

    \param origin the origin of the relative coordinate frame

    \param ang the angle between the world frame and the relative
    frame (reasoning from the world frame)

    \return the result of converting the current global VecPosition
    into a relative VecPosition */
VecPosition VecPosition::globalToRelative( VecPosition origin, AngDeg ang )
{
  // convert global coordinates into relative coordinates by aligning
  // relative frame and world frame. First perform translation to make
  // origins of both frames coincide. Then perform rotation to make
  // axes of both frames coincide (use negative angle since you rotate
  // relative frame to world frame).
  *this -= origin;
  return ( rotate( -ang ) );
}
```

```
/*! This method converts the coordinates of the current VecPosition
    (which are represented in a relative coordinate system) into
    global coordinates in the world frame (with origin at (0,0)). The
    relative coordinate system is defined by the arguments to the
    method. The global coordinates are now obtained by aligning the
    world frame with the relative frame using a rotation to align the
    axes followed by a translation to make both origins coincide.

    \param origin the origin of the relative coordinate frame

    \param ang the angle between the world frame and the relative
    frame (reasoning from the world frame)

    \return the result of converting the current relative VecPosition
    into an global VecPosition */
VecPosition VecPosition::relativeToGlobal( VecPosition origin, AngDeg ang )
{
  // convert relative coordinates into global coordinates by aligning
  // world frame and relative frame. First perform rotation to make
  // axes of both frames coincide (use positive angle since you rotate
  // world frame to relative frame). Then perform translation to make
  // origins of both frames coincide.
  rotate( ang );
  *this += origin;
  return ( *this );
}

/*! This method returns a VecPosition that lies somewhere on the
    vector between the current VecPosition and a given
    VecPosition. The desired position is specified by a given fraction
    of this vector (e.g. 0.5 means exactly in the middle of the
    vector). The current VecPosition itself is left unchanged.

    \param p a VecPosition which defines the vector to the current
    VecPosition

    \param dFrac double representing the fraction of the connecting
    vector at which the desired VecPosition lies.

    \return the VecPosition which lies at fraction dFrac on the vector
    connecting p and the current VecPosition */
VecPosition VecPosition::getVecPositionOnLineFraction( VecPosition &p,
                                       double      dFrac )
{
  // determine point on line that lies at fraction dFrac of whole line
```

```
  // example: this --- 0.25 ---------  p
  // formula: this + dFrac * ( p - this ) = this - dFrac * this + dFrac * p =
  //          ( 1 - dFrac ) * this + dFrac * p
  return ( ( *this ) * ( 1.0 - dFrac ) + ( p * dFrac ) );
}

/*! This method converts a polar representation of a VecPosition into
    a Cartesian representation.

    \param dMag a double representing the polar r-coordinate, i.e. the
    distance from the point to the origin

    \param ang the angle that the polar vector makes with the x-axis,
    i.e. the polar phi-coordinate

    \return the result of converting the given polar representation
    into a Cartesian representation thus yielding a Cartesian
    VecPosition */
VecPosition VecPosition::getVecPositionFromPolar( double dMag, AngDeg ang )
{
  // cos(phi) = x/r <=> x = r*cos(phi); sin(phi) = y/r <=> y = r*sin(phi)
  return ( VecPosition( dMag * cosDeg( ang ), dMag * sinDeg( ang ) ) );
}

/*! This method normalizes an angle. This means that the resulting
    angle lies between -180 and 180 degrees.

    \param angle the angle which must be normalized

    \return the result of normalizing the given angle */
AngDeg VecPosition::normalizeAngle( AngDeg angle )
{
  while( angle > 180.0  ) angle -= 360.0;
  while( angle < -180.0 ) angle += 360.0;

  return ( angle );
}


/*************************************************************************
/
/********************** CLASS GEOMETRY
*************************************/
/*************************************************************************
/
```

```
/*! A geometric series is one in which there is a constant ratio between each
    element and the one preceding it. This method determines the
    length of a geometric series given its first element, the sum of the
    elements in the series and the constant ratio between the elements.
    Normally: s = a + ar + ar^2 + ...  + ar^n
    Now: dSum = dFirst + dFirst*dRatio + dFirst*dRatio^2 + .. + dFist*dRatio^n
    \param dFirst first term of the series
    \param dRatio ratio with which the the first term is multiplied
    \param dSum the total sum of all the serie
    \return the length(n in above example) of the series */
double Geometry::getLengthGeomSeries( double dFirst, double dRatio, double dSum )
{
  if( dRatio < 0 )
    cerr << "(Geometry:getLengthGeomSeries): negative ratio" << endl;

  // s = a + ar + ar^2 + .. + ar^n-1 and thus sr = ar + ar^2 + .. + ar^n
  // subtract: sr - s = - a + ar^n) =>  s(1-r)/a + 1 = r^n = temp
  // log r^n / n = n log r / log r = n = length
  double temp = (dSum * ( dRatio - 1 ) / dFirst) + 1;
  if( temp <= 0 )
    return -1.0;
  return log( temp ) / log( dRatio ) ;
}

/*! A geometric series is one in which there is a constant ratio between each
    element and the one preceding it. This method determines the sum of a
    geometric series given its first element, the ratio and the number of steps
    in the series
    Normally: s = a + ar + ar^2 + ...  + ar^n
    Now: dSum = dFirst + dFirst*dRatio + ... + dFirst*dRatio^dSteps
    \param dFirst first term of the series
    \param dRatio ratio with which the the first term is multiplied
    \param dSum the number of steps to be taken into account
    \return the sum of the series */
double Geometry::getSumGeomSeries( double dFirst, double dRatio, double dLength)
{
  // s = a + ar + ar^2 + .. + ar^n-1 and thus sr = ar + ar^2 + .. + ar^n
  // subtract: s - sr = a - ar^n) =>  s = a(1-r^n)/(1-r)
  return dFirst * ( 1 - pow( dRatio, dLength ) ) / ( 1 - dRatio ) ;
}

/*! A geometric series is one in which there is a constant ratio between each
    element and the one preceding it. This method determines the sum of an
    infinite geometric series given its first element and the constant ratio
    between the elements. Note that such an infinite series will only converge
    when 0<r<1.
```

```
    Normally: s = a + ar + ar^2 + ar^3 + ....
    Now: dSum = dFirst + dFirst*dRatio + dFirst*dRatio^2...
    \param dFirst first term of the series
    \param dRatio ratio with which the the first term is multiplied
    \return the sum of the series */
double Geometry::getSumInfGeomSeries( double dFirst, double dRatio )
{
  if( dRatio > 1 )
    cerr << "(Geometry:CalcLengthGeomSeries): series does not converge" <<endl;

  // s = a(1-r^n)/(1-r) with n->inf and 0<r<1 => r^n = 0
  return dFirst / ( 1 - dRatio );
}

/*! A geometric series is one in which there is a constant ratio between each
    element and the one preceding it. This method determines the first element
    of a geometric series given its element, the ratio and the number of steps
    in the series
    Normally: s = a + ar + ar^2 + ...  + ar^n
    Now: dSum = dFirst + dFirst*dRatio + ... + dFirst*dRatio^dSteps
    \param dSum sum of the series
    \param dRatio ratio with which the the first term is multiplied
    \param dSum the number of steps to be taken into account
    \return the first element (a) of a serie */
double Geometry::getFirstGeomSeries( double dSum, double dRatio, double dLength)
{
  // s = a + ar + ar^2 + .. + ar^n-1 and thus sr = ar + ar^2 + .. + ar^n
  // subtract: s - sr = a - ar^n) =>  s = a(1-r^n)/(1-r) => a = s*(1-r)/(1-r^n)
  return dSum *  ( 1 - dRatio )/( 1 - pow( dRatio, dLength ) ) ;
}

/*! A geometric series is one in which there is a constant ratio between each
    element and the one preceding it. This method determines the first element
    of an infinite geometric series given its first element and the constant
    ratio between the elements. Note that such an infinite series will only
    converge when 0<r<1.
    Normally: s = a + ar + ar^2 + ar^3 + ....
    Now: dSum = dFirst + dFirst*dRatio + dFirst*dRatio^2...
    \param dSum sum of the series
    \param dRatio ratio with which the the first term is multiplied
    \return the first term of the series */
double Geometry::getFirstInfGeomSeries( double dSum, double dRatio )
{
  if( dRatio > 1 )
    cerr << "(Geometry:getFirstInfGeomSeries):series does not converge" <<endl;
```

```
  // s = a(1-r^n)/(1-r) with r->inf and 0<r<1 => r^n = 0 => a = s ( 1 - r)
  return dSum * ( 1 - dRatio );
}

/*! This method performs the abc formula (Pythagoras' Theorem) on the given
    parameters and puts the result in *s1 en *s2. It returns the number of
    found coordinates.
    \param a a parameter in abc formula
    \param b b parameter in abc formula
    \param c c parameter in abc formula
    \param *s1 first result of abc formula
    \param *s2 second result of abc formula
    \return number of found x-coordinates */
int Geometry::abcFormula(double a, double b, double c, double *s1, double *s2)
{
  double dDiscr = b*b - 4*a*c;       // discriminant is b^2 - 4*a*c
  if (fabs(dDiscr) < EPSILON )       // if discriminant = 0
  {
    *s1 = -b / (2 * a);              //  only one solution
    return 1;
  }
  else if (dDiscr < 0)               // if discriminant < 0
    return 0;                        //  no solutions
  else                               // if discriminant > 0
  {
    dDiscr = sqrt(dDiscr);           //  two solutions
    *s1 = (-b + dDiscr ) / (2 * a);
    *s2 = (-b - dDiscr ) / (2 * a);
    return 2;
  }
}

/*****************************************************************************
/
/********************* CLASS CIRCLE
*****************************************/
/*****************************************************************************
/

/*! This is the constructor of a circle.
    \param pos first point that defines the center of circle
    \param dR the radius of the circle
    \return circle with pos as center and radius as radius*/
Circle::Circle( VecPosition pos, double dR )
{
  setCircle( pos, dR );
```

```
}

/*! This is the constructor of a circle which initializes a circle with a
    radius of zero. */
Circle::Circle( )
{
  setCircle( VecPosition(-1000.0,-1000.0), 0);
}

/*! This method prints the circle information to the specified output stream
    in the following format: "c: (c_x,c_y), r: rad" where (c_x,c_y) denotes
    the center of the circle and rad the radius.
    \param os output stream to which output is written. */
void Circle::show( ostream& os)
{
  os << "c:" << m_posCenter << ", r:" << m_dRadius;
}

/*! This method sets the values of the circle.
    \param pos new center of the circle
    \param dR new radius of the circle
    ( > 0 )
    \return bool indicating whether radius was set */
bool Circle::setCircle( VecPosition pos, double dR )
{
  setCenter( pos );
  return setRadius( dR  );
}
/*! This method sets the radius of the circle.
    \param dR new radius of the circle ( > 0 )
    \return bool indicating whether radius was set */
bool Circle::setRadius( double dR )
{
  if( dR > 0 )
  {
    m_dRadius = dR;
    return true;
  }
  else
  {
    m_dRadius = 0.0;
    return false;
  }
}

/*! This method returns the radius of the circle.
```

```
    \return radius of the circle */
double Circle::getRadius()
{
  return m_dRadius;
}

/*! This method sets the center of the circle.
    \param pos new center of the circle
    \return bool indicating whether center was set */
bool Circle::setCenter( VecPosition pos )
{
  m_posCenter = pos;
  return true;
}

/*! This method returns the center of the circle.
    \return center of the circle */
VecPosition Circle::getCenter()
{
  return m_posCenter;
}

/*! This method returns the circumference of the circle.
    \return circumference of the circle */
double Circle::getCircumference()
{
  return 2.0*M_PI*getRadius();
}

/*! This method returns the area inside the circle.
    \return area inside the circle */
double Circle::getArea()
{
  return M_PI*getRadius()*getRadius();
}

/*! This method returns a boolean that indicates whether 'pos' is
     located inside the circle.

   \param pos position of which should be checked whether it is
   located in the circle

   \return bool indicating whether pos lies inside the circle */
bool Circle::isInside( VecPosition pos )
{
  return m_posCenter.getDistanceTo( pos ) < getRadius() ;
```

```
}

/*! This method returns the two possible intersection points between two
    circles. This method returns the number of solutions that were found.
    \param c circle with which intersection should be found
    \param p1 will be filled with first solution
    \param p2 will be filled with second solution
    \return number of solutions. */
int Circle::getIntersectionPoints( Circle c, VecPosition *p1, VecPosition *p2)
{
    double x0, y0, r0;
    double x1, y1, r1;

    x0 = getCenter( ).getX();
    y0 = getCenter( ).getY();
    r0 = getRadius( );
    x1 = c.getCenter( ).getX();
    y1 = c.getCenter( ).getY();
    r1 = c.getRadius( );

    double      d, dx, dy, h, a, x, y, p2_x, p2_y;

    // first calculate distance between two centers circles P0 and P1.
    dx = x1 - x0;
    dy = y1 - y0;
    d = sqrt(dx*dx + dy*dy);

    // normalize differences
    dx /= d; dy /= d;

    // a is distance between p0 and point that is the intersection point P2
    // that intersects P0-P1 and the line that crosses the two intersection
    // points P3 and P4.
    // Define two triangles: P0,P2,P3 and P1,P2,P3.
    // with distances a, h, r0 and b, h, r1 with d = a + b
    // We know a^2 + h^2 = r0^2 and b^2 + h^2 = r1^2 which then gives
    // a^2 + r1^2 - b^2 = r0^2 with d = a + b ==> a^2 + r1^2 - (d-a)^2 = r0^2
    // ==> r0^2 + d^2 - r1^2 / 2*d
    a = (r0*r0 + d*d - r1*r1) / (2.0 * d);

    // h is then a^2 + h^2 = r0^2 ==> h = sqrt( r0^2 - a^2 )
    double      arg = r0*r0 - a*a;
    h = (arg > 0.0) ? sqrt(arg) : 0.0;

    // First calculate P2
    p2_x = x0 + a * dx;
```

```
    p2_y = y0 + a * dy;

    // and finally the two intersection points
    x =  p2_x - h * dy;
    y =  p2_y + h * dx;
    p1->setVecPosition( x, y );
    x =  p2_x + h * dy;
    y =  p2_y - h * dx;
    p2->setVecPosition( x, y );

    return (arg < 0.0) ? 0 : ((arg == 0.0 ) ? 1 :  2);
}

/*! This method returns the size of the intersection area of two circles.
    \param c circle with which intersection should be determined
    \return size of the intersection area. */
double Circle::getIntersectionArea( Circle c )
{
  VecPosition pos1, pos2, pos3;
  double d, h, dArea;
  AngDeg ang;

  d = getCenter().getDistanceTo( c.getCenter() ); // dist between two centers
  if( d > c.getRadius() + getRadius() )           // larger than sum radii
    return 0.0;                                   // circles do not intersect
  if( d <= fabs(c.getRadius() - getRadius() ) )   // one totally in the other
  {
    double dR = min( c.getRadius(), getRadius() );// return area smallest circ
    return M_PI*dR*dR;
  }

  int iNrSol = getIntersectionPoints( c, &pos1, &pos2 );
  if( iNrSol != 2 )
    return 0.0;

  // the intersection area of two circles can be divided into two segments:
  // left and right of the line between the two intersection points p1 and p2.
  // The outside area of each segment can be calculated by taking the part
  // of the circle pie excluding the triangle from the center to the
  // two intersection points.
  // The pie equals pi*r^2 * rad(2*ang) / 2*pi = 0.5*rad(2*ang)*r^2 with ang
  // the angle between the center c of the circle and one of the two
  // intersection points. Thus the angle between c and p1 and c and p3 where
  // p3 is the point that lies halfway between p1 and p2.
  // This can be calculated using ang = asin( d / r ) with d the distance
  // between p1 and p3 and r the radius of the circle.
```

```
  // The area of the triangle is 2*0.5*h*d.

  pos3 = pos1.getVecPositionOnLineFraction( pos2, 0.5 );
  d = pos1.getDistanceTo( pos3 );
  h = pos3.getDistanceTo( getCenter() );
  ang = asin( d / getRadius() );

  dArea = ang*getRadius()*getRadius();
  dArea = dArea - d*h;

  // and now for the other segment the same story
  h = pos3.getDistanceTo( c.getCenter() );
  ang = asin( d / c.getRadius() );
  dArea = dArea + ang*c.getRadius()*c.getRadius();
  dArea = dArea - d*h;

  return dArea;
}


/*************************************************************************
/
/*********************  CLASS LINE ***********************************/
/*************************************************************************
/

/*! This constructor creates a line by given the three coefficents of the line.
    A line is specified by the formula ay + bx + c = 0.
    \param dA a coefficients of the line
    \param dB b coefficients of the line
    \param dC c coefficients of the line */
Line::Line( double dA, double dB, double dC )
{
  m_a = dA;
  m_b = dB;
  m_c = dC;
}

/*! This function prints the line to the specified output stream in the
    format y = ax + b.
    \param os output stream to which output is written
    \param l line that is written to output stream
    \return output sream to which output is appended. */
ostream& operator <<(ostream & os, Line l)
{
  double a = l.getACoefficient();
```

```
  double b = l.getBCoefficient();
  double c = l.getCCoefficient();

  // ay + bx + c = 0 -> y = -b/a x - c/a
  if( a == 0 )
    os << "x = " << -c/b;
  else
  {
    os << "y = ";
    if( b != 0 )
      os << -b/a << "x ";
    if( c > 0 )
       os << "- " <<  fabs(c/a);
    else if( c < 0 )
       os << "+ " <<  fabs(c/a);
  }
  return os;
}

/*! This method prints the line information to the specified output stream.
    \param os output stream to which output is written. */
void Line::show( ostream& os)
{
  os << *this;
}

/*! This method returns the intersection point between the current Line and
    the specified line.
    \param line line with which the intersection should be calculated.
    \return VecPosition position that is the intersection point. */
VecPosition Line::getIntersection( Line line )
{
  VecPosition pos;
  double x, y;
  if( ( m_a / m_b ) ==  (line.getACoefficient() / line.getBCoefficient() ))
    return pos; // lines are parallel, no intersection
  if( m_a == 0 )           // bx + c = 0 and a2*y + b2*x + c2 = 0 ==> x = -c/b
  {                        // calculate x using the current line
    x = -m_c/m_b;               // and calculate the y using the second line
    y = line.getYGivenX(x);
  }
  else if( line.getACoefficient() == 0 )
  {                        // ay + bx + c = 0 and b2*x + c2 = 0 ==> x = -c2/b2
   x = -line.getCCoefficient()/line.getBCoefficient(); // calculate x using
   y = getYGivenX(x);       // 2nd line and calculate y using current line
  }
```

```
  // ay + bx + c = 0 and a2y + b2*x + c2 = 0
  // y = (-b2/a2)x - c2/a2
  // bx = -a*y - c =>  bx = -a*(-b2/a2)x -a*(-c2/a2) - c ==>
  // ==> a2*bx = a*b2*x + a*c2 - a2*c ==> x = (a*c2 - a2*c)/(a2*b - a*b2)
  // calculate x using the above formula and the y using the current line
  else
  {
    x = (m_a*line.getCCoefficient() - line.getACoefficient()*m_c)/
                (line.getACoefficient()*m_b - m_a*line.getBCoefficient());
    y = getYGivenX(x);
  }

  return VecPosition( x, y );
}


/*! This method calculates the intersection points between the current line
    and the circle specified with as center 'posCenter' and radius 'dRadius'.
    The number of solutions are returned and the corresponding points are put
    in the third and fourth argument of the method
    \param c circle with which intersection points should be found
    \param posSolution1 first intersection (if any)
    \param posSolution2 second intersection (if any) */
int Line::getCircleIntersectionPoints( Circle circle,
            VecPosition *posSolution1, VecPosition *posSolution2 )
{
  int    iSol;
  double dSol1=0, dSol2=0;
  double h = circle.getCenter().getX();
  double k = circle.getCenter().getY();

  // line:   x = -c/b (if a = 0)
  // circle: (x-h)^2 + (y-k)^2 = r^2, with h = center.x and k = center.y
  // fill in:(-c/b-h)^2 + y^2 -2ky + k^2 - r^2 = 0
  //         y^2 -2ky + (-c/b-h)^2 + k^2 - r^2 = 0
  // and determine solutions for y using abc-formula
  if( fabs(m_a) < EPSILON )
  {
    iSol = Geometry::abcFormula( 1, -2*k, ((-m_c/m_b) - h)*((-m_c/m_b) - h)
            + k*k - circle.getRadius()*circle.getRadius(), &dSol1, &dSol2);
    posSolution1->setVecPosition( (-m_c/m_b), dSol1 );
    posSolution2->setVecPosition( (-m_c/m_b), dSol2 );
    return iSol;
  }

  // ay + bx + c = 0 => y = -b/a x - c/a, with da = -b/a and db = -c/a
```

```
  // circle: (x-h)^2 + (y-k)^2 = r^2, with h = center.x and k = center.y
  // fill in:x^2 -2hx + h^2 + (da*x-db)^2 -2k(da*x-db) + k^2 - r^2 = 0
  //         x^2 -2hx + h^2 + da^2*x^2 + 2da*db*x + db^2 -2k*da*x -2k*db
  //                                              + k^2 - r^2 = 0
  //       (1+da^2)*x^2 + 2(da*db-h-k*da)*x + h2 + db^2  -2k*db + k^2 - r^2 = 0
  // and determine solutions for x using abc-formula
  // fill in x in original line equation to get y coordinate
  double da = -m_b/m_a;
  double db = -m_c/m_a;

  double dA = 1 + da*da;
  double dB = 2*( da*db - h - k*da );
  double dC = h*h + db*db-2*k*db + k*k - circle.getRadius()*circle.getRadius();

  iSol = Geometry::abcFormula( dA, dB, dC, &dSol1, &dSol2 );

  posSolution1->setVecPosition( dSol1, da*dSol1 + db );
  posSolution2->setVecPosition( dSol2, da*dSol2 + db );
  return iSol;

}

/*! This method returns the tangent line to a VecPosition. This is the line
    between the specified position and the closest point on the line to this
    position.
    \param pos VecPosition point with which tangent line is calculated.
    \return Line line tangent to this position */
Line Line::getTangentLine( VecPosition pos )
{
  // ay + bx + c = 0 -> y = (-b/a)x + (-c/a)
  // tangent: y = (a/b)*x + C1 -> by - ax + C2 = 0 => C2 = ax - by
  // with pos.y = y, pos.x = x
  return Line( m_b, -m_a, m_a*pos.getX() - m_b*pos.getY() );
}

/*! This method returns the closest point on a line to a given position.
    \param pos point to which closest point should be determined
    \return VecPosition closest point on line to 'pos'. */
VecPosition Line::getPointOnLineClosestTo( VecPosition pos )
{
  Line l2 = getTangentLine( pos );  // get tangent line
  return getIntersection( l2 );     // and intersection between the two lines
}

/*! This method returns the distance between a specified position and the
    closest point on the given line.
```

```
    \param pos position to which distance should be calculated
    \return double indicating the distance to the line. */
double Line::getDistanceWithPoint( VecPosition pos )
{
  return pos.getDistanceTo( getPointOnLineClosestTo( pos ) );
}

/*! This method determines whether the projection of a point on the
    current line lies between two other points ('point1' and 'point2')
    that lie on the same line.

    \param pos point of which projection is checked.
    \param point1 first point on line
    \param point2 second point on line
    \return true when projection of 'pos' lies between 'point1' and 'point2'.*/
bool Line::isInBetween( VecPosition pos, VecPosition point1,VecPosition point2)
{
  pos          = getPointOnLineClosestTo( pos ); // get closest point
  double dDist = point1.getDistanceTo( point2 ); // get distance between 2 pos

  // if the distance from both points to the projection is smaller than this
  // dist, the pos lies in between.
  return pos.getDistanceTo( point1 ) <= dDist &&
         pos.getDistanceTo( point2 ) <= dDist;
}

/*! This method calculates the y coordinate given the x coordinate
    \param x coordinate
    \return y coordinate on this line */
double Line::getYGivenX( double x )
{
 if( m_a == 0 )
 {
   cerr << "(Line::getYGivenX) Cannot calculate Y coordinate: " ;
   show( cerr );
   cerr << endl;
   return 0;
 }
  // ay + bx + c = 0 ==> ay = -(b*x + c)/a
  return -(m_b*x+m_c)/m_a;
}

/*! This method calculates the x coordinate given the x coordinate
    \param y coordinate
    \return x coordinate on this line */
double Line::getXGivenY( double y )
```

```
{
 if( m_b == 0 )
 {
  cerr << "(Line::getXGivenY) Cannot calculate X coordinate\n" ;
  return 0;
 }
 // ay + bx + c = 0 ==> bx = -(a*y + c)/a
 return -(m_a*y+m_c)/m_b;
}

/*! This method creates a line given two points.
    \param pos1 first point
    \param pos2 second point
    \return line that passes through the two specified points. */
Line Line::makeLineFromTwoPoints( VecPosition pos1, VecPosition pos2 )
{
 // 1*y + bx + c = 0 => y = -bx - c
 // with -b the direction coefficient (or slope)
 // and c = - y - bx
 double dA, dB, dC;
 double dTemp = pos2.getX() - pos1.getX(); // determine the slope
 if( fabs(dTemp) < EPSILON )
 {
  // ay + bx + c = 0 with vertical slope=> a = 0, b = 1
  dA = 0.0;
  dB = 1.0;
 }
 else
 {
  // y = (-b)x -c with -b the slope of the line
  dA = 1.0;
  dB = -(pos2.getY() - pos1.getY())/dTemp;
 }
 // ay + bx + c = 0 ==> c = -a*y - b*x
 dC =  - dA*pos2.getY()  - dB * pos2.getX();
 return Line( dA, dB, dC );
}

/*! This method creates a line given a position and an angle.
    \param vec position through which the line passes
    \param angle direction of the line.
    \return line that goes through position 'vec' with angle 'angle'. */
Line Line::makeLineFromPositionAndAngle( VecPosition vec, AngDeg angle )
{
 // calculate point somewhat further in direction 'angle' and make
 // line from these two points.
```

```
  return makeLineFromTwoPoints( vec, vec+VecPosition(1,angle,POLAR));
}

/*! This method returns the a coefficient from the line ay + bx + c = 0.
    \return a coefficient of the line. */
double Line::getACoefficient() const
{
  return m_a;
}

/*! This method returns the b coefficient from the line ay + bx + c = 0.
    \return b coefficient of the line. */
double Line::getBCoefficient() const
{
 return m_b;
}

/*! This method returns the c coefficient from the line ay + bx + c = 0.
    \return c coefficient of the line. */
double Line::getCCoefficient() const
{
 return m_c;
}

/*****************************************************************************
/
/********************* CLASS RECTANGLE
*****************************************/
/*****************************************************************************
/

/*! This is the constructor of a Rectangle. Two points will be given. The
    order does not matter as long as two opposite points are given (left
    top and right bottom or right top and left bottom).
    \param pos first point that defines corner of rectangle
    \param pos2 second point that defines other corner of rectangle
    \return rectangle with 'pos' and 'pos2' as opposite corners. */
Rect::Rect( VecPosition pos, VecPosition pos2 )
{
  setRectanglePoints( pos, pos2 );
}

/*! This method sets the upper left and right bottom point of the current
    rectangle.
    \param pos first point that defines corner of rectangle
    \param pos2 second point that defines other corner of rectangle */
```

```
void Rect::setRectanglePoints( VecPosition pos1, VecPosition pos2 )
{
  m_posLeftTop.setX    ( max( pos1.getX(), pos2.getX() ) );
  m_posLeftTop.setY    ( min( pos1.getY(), pos2.getY() ) );
  m_posRightBottom.setX( min( pos1.getX(), pos2.getX() ) );
  m_posRightBottom.setY( max( pos1.getY(), pos2.getY() ) );
}

/*! This method prints the rectangle to the specified output stream in the
    format rect( top_left_point, bottom_right_point ).
    \param os output stream to which rectangle is written. */
void Rect::show( ostream& os )
{
  os << "rect(" << m_posLeftTop << " " << m_posRightBottom << ")";
}

/*! This method determines whether the given position lies inside the current
    rectangle.
    \param pos position which is checked whether it lies in rectangle
    \return true when 'pos' lies in the rectangle, false otherwise */
bool Rect::isInside( VecPosition pos )
{
  return pos.isBetweenX( m_posRightBottom.getX(), m_posLeftTop.getX() ) &&
         pos.isBetweenY( m_posLeftTop.getY(),     m_posRightBottom.getY() );

}

/*! This method sets the top left position of the rectangle
    \param pos new top left position of the rectangle
    \return true when update was successful */
bool Rect::setPosLeftTop( VecPosition pos )
{
  m_posLeftTop = pos;
  return true;
}

/*! This method returns the top left position of the rectangle
    \return top left position of the rectangle */
VecPosition Rect::getPosLeftTop(  )
{
  return m_posLeftTop;
}

/*! This method sets the right bottom position of the rectangle
    \param pos new right bottom position of the rectangle
    \return true when update was succesfull */
```

```
bool Rect::setPosRightBottom( VecPosition pos )
{
  m_posRightBottom = pos;
  return true;
}

/*! This method returns the right bottom position of the rectangle
    \return top right bottom of the rectangle */
VecPosition Rect::getPosRightBottom(  )
{
  return m_posRightBottom;
}

/**************************************************************************
/
/********************** TESTING PURPOSES
***************************************/
/**************************************************************************
/

/*
#include<iostream.h>

int main( void )
{
  double dFirst = 1.0;
  double dRatio = 2.5;
  double dSum   = 63.4375;
  double dLength = 4.0;

  printf( "sum: %f\n", Geometry::getSumGeomSeries( dFirst, dRatio, dLength));
  printf( "length: %f\n", Geometry::getLengthGeomSeries( dFirst, dRatio, dSum));
}

int main( void )
{
  Line l1(1,-1,3 );
  Line l2(1,-0.2,10 );
 Line l3 = Line::makeLineFromTwoPoints( VecPosition(1,-1), VecPosition(2,-2) );
 l3.show();
 cout << endl;
 l1.show();
 l2.show();
  l1.getIntersection( l2 ).show();
}
```

```
int main( void )
{
  Line l( 1, -1, 0 );
  VecPosition s1, s2;
  int i = l.getCircleIntersectionPoints( Circle( VecPosition(1,1),1) &s1,&s2 );
  printf( "number of solutions: %d\n", i );
  if( i == 2 )
  {
    cout << s1 << " " << s2 ;
  }
  else if( i == 1 )
  {
    cout << s1;
  }
  cout << "line: " << l;
}

int main( void )
{
  Circle c11( VecPosition( 10, 0 ), 10);
  Circle c12( VecPosition( 40, 3 ), 40 );
  Circle c21( VecPosition(  0,0 ), 5);
  Circle c22( VecPosition(  3,0 ), 40 );

  VecPosition p1, p2;

  cout << c11.getIntersectionArea( c21 ) << endl;
  cout << c12.getIntersectionArea( c21 ) << endl;
  cout << c22.getIntersectionArea( c11 ) << endl;
  cout << c12.getIntersectionArea( c22 ) << endl;
  return 0;
}

int main( void )
{
  cout << getBisectorTwoAngles( -155.3, 179.0 ) << endl;
  cout << getBisectorTwoAngles( -179.3, 179.0 ) << endl;
}
*/
```

ServerSettings.h

```
#ifndef _SERVERSETTINGS_
#define _SERVERSETTINGS_

#include "GenericValues.h"

/*****************************************************************************
/
/*******************   CLASS SERVERSETTINGS
*********************************/
/*****************************************************************************
/

/*! This class contains all the Soccerserver parameters that are available in
    the configuration file 'server.conf' along with their default values and
    standard set- and get methods for manipulating these values. The
    ServerSettings class is a subclass of the GenericValues class and therefore
    each value in this class can be reached through the string name of the
    corresponding parameter. */
class ServerSettings:public GenericValues
{
private:
  // private member data

  // all the parameters available in server.conf
  // NOTE: names in server.conf corresponding with member variables
  //       are listed between doxygen-tags to enable quick searching

  // goal-related parameters
  double dGoalWidth;       /*!< goal_width: the width of the goal          */

  // player-related parameters
  double dPlayerSize;      /*!< player_size: the size (=radius) of a player */
  double dPlayerDecay;     /*!< player_decay: player speed decay per cycle  */
  double dPlayerRand;      /*!< player_rand: random error in player movement*/
  double dPlayerWeight;    /*!< player_weight: weight of a player (for wind)*/
  double dPlayerSpeedMax;  /*!< player_speed_max: maximum speed of a player */
  double dPlayerAccelMax;  /*!< player_accel_max: maximum acceleration of a
                            player per cycle                          */

  // stamina-related parameters
  double dStaminaMax;      /*!< stamina_max: maximum stamina of a player    */
  double dStaminaIncMax;   /*!< stamina_inc_max: maximum stamina increase of a
                            player per cycle                          */
  double dRecoverDecThr;   /*!< recover_dec_thr: percentage of stamina_max
                            below which player recovery decreases       */
  double dRecoverDec;      /*!< recover_dec: decrement step per cycle for
```

```
                    player recovery                         */
double dRecoverMin;     /*!< recover_min: minimum player recovery      */
double dEffortDecThr;   /*!< effort_dec_thr: percentage of stamina_max
                    below which player effort capacity decreases*/
double dEffortDec;      /*!< effort_dec: decrement step per cycle for
                    player effort capacity                  */
double dEffortIncThr;   /*!< effort_incr_thr: percentage of stamina_max
                    above which player effort capacity increases*/
double dEffortInc;      /*!< effort_inc: increment step per cycle for
                    player effort capacity                  */
double dEffortMin;      /*!< effort_min: minimum value for player effort */
double dEffortMax;      /*!< effort_max: maximum player effort capacity  */


// parameters related to auditory perception
int   iHearMax;         /*!< hear_max: maximum hearing capacity of a plyer
                    a player can hear hear_inc messages in
                    hear_decay simulation cycles            */
int   iHearInc;         /*!< hear_inc: minimum hearing capacity of a player
                    i.e. the number of messages a player can hear
                    in hear_decay simulation cycles         */
int   iHearDecay;       /*!< hear_decay: decay rate of player hearing
                    capacity, i.e. minimum number of cycles for
                    hear_inc messages                       */

// parameters related to player turn actions
double dInertiaMoment;  /*!< inertia_moment: inertia moment of a player;
                    affects actual turn angle depending on speed*/

// parameters related to sense_body information
int   iSenseBodyStep;   /*!< sense_body_step: length of the interval (ms)
                    between sense_body information messages   */

// goalkeeper-related parameters
double dCatchableAreaL;  /*!< catchable_area_l: length of area around
                    goalkeeper in which he can catch the ball  */
double dCatchableAreaW;  /*!< catchable_area_w: width of area around
                    goalkeeper in which he can catch the ball  */
double dCatchProbability; /*!< catch_probability: the probability for a
                    goalkeeper to catch the ball            */
int   iCatchBanCycle;   /*!< catch_ban_cycle: number of cycles after catch
                    in which goalkeeper cannot catch again    */
int   iGoalieMaxMoves;  /*!< goalie_max_moves: maximum number of 'move'
                    actions allowed for goalkeeper after catch  */

// ball-related parameters
```

```
double dBallSize;        /*!< ball_size: the size (=radius) of the ball   */
double dBallDecay;       /*!< ball_decay: ball speed decay per cycle      */
double dBallRand;        /*!< ball_rand: random error in ball movement    */
double dBallWeight;      /*!< ball_weight: weight of the ball (for wind)  */
double dBallSpeedMax;    /*!< ball_speed_max: maximum speed of the ball   */
double dBallAccelMax;    /*!< ball_accel_max: maximum acceleration of the
                         ball per cycle                                    */

// wind-related parameters
double dWindForce;       /*!< wind_force: the force of the wind           */
double dWindDir;         /*!< wind_dir: the direction of the wind         */
double dWindRand;        /*!< wind_rand: random error in wind direction   */
bool   bWindRandom;      /*!< wind_random: random wind force and direction*/

// parameters related to 'dash' and 'kick' commands
double dKickableMargin;  /*!< kickable_margin: margin around player in which
                         ball is kickable; kickable area thus equals
                         kickable_margin + ball_size + player_size   */
double dCkickMargin;     /*!< ckick_margin: corner kick margin, i.e. the
                         minimum distance to the ball for offending
                         players when a corner kick is taken         */
double dDashPowerRate;   /*!< dash_power_rate: rate by which the 'Power'
                         argument in a 'dash' command is multiplied
                         (thus determining the amount of displacement
                         of the player as a result of the 'dash')    */
double dKickPowerRate;   /*!< kick_power_rate: rate by which the 'Power'
                         argument in a 'kick' command is multiplied
                         (thus determining the amount of displacement
                         of the ball as a result of the 'kick')      */
double dKickRand;        /*!< kick_rand: random error in kick direction   */

// parameters related to visual and auditory perception range
double dVisibleAngle;    /*!< visible_angle: angle of the view cone of a
                         player in the standard view mode            */
double dAudioCutDist;    /*!< audio_cut_dist: maximum distance over which a
                         spoken message can be heard                 */

// quantization parameters
double dQuantizeStep;    /*!< quantize_step: quantization step for distance
                         of moving objects                           */
double dQuantizeStepL;   /*!< quantize_step_l: quantization step for
                         distance of landmarks                       */

// range parameters for basic actuator commands
int    iMaxPower;        /*!< maxpower: maximum power for dash/kick       */
int    iMinPower;        /*!< minpower: minimum power for dash/kick       */
```

```
int    iMaxMoment;       /*!< maxmoment: maximum angle for turn/kick      */
int    iMinMoment;       /*!< minmoment: minimum angle for turn/kick      */
int    iMaxNeckMoment;   /*!< maxneckmoment: maximum angle for turnneck   */
int    iMinNeckMoment;   /*!< minneckmoment: minimum angle for turnneck   */
int    iMaxNeckAng;      /*!< maxneckang: maximum neck angle rel. to body */
int    iMinNeckAng;      /*!< minneckang: minimum neck angle rel. to body */

// port-related parameters
int    iPort;            /*!< port: port number for player connection     */
int    iCoachPort;       /*!< coach_port: port number for coach connection*/
int    iOlCoachPort;     /*!< ol_coach_port: port number for online coach */

// coach-related parameters
int    iSayCoachCntMax;  /*!< say_coach_cnt_max: maximum number of coach
                          messages possible                               */
int    iSayCoachMsgSize; /*!< say_coach_msg_size: maximum size of coach
                          messages                                        */
int    iClangWinSize;    /*!< clang_win_size: time window which controls how
                          many coach messages can be sent                 */
int    iClangDefineWin;  /*!< clang_define_win: number of define messages by
                          coach per time window                           */
int    iClangMetaWin;    /*!< clang_meta_win: number of meta messages by
                          coach per time window                           */
int    iClangAdviceWin;  /*!< clang_advice_win: number of advice messages by
                          coach per time window                           */
int    iClangInfoWin;    /*!< clang_info_win: number of info messages by
                          coach per time window                           */
int    iClangMessDelay;  /*!< clang_mess_delay: delay of coach messages, ie
                          the number of cycles between send to player
                          and receival of message                         */
int    iClangMessPerCycle;/*!< clang_mess_per_cycle: number of coach messages
                          per cycle                                       */
int    iSendViStep;      /*!< send_vi_step: interval of coach's look, i.e.
                          the length of the interval (in ms) between
                          visual messages to the coach                    */

// time-related parameters
int    iSimulatorStep;   /*!< simulator_step: the length (in ms) of a
                          simulator cycle                                 */
int    iSendStep;        /*!< send_step: the length of the interval (in ms)
                          between visual messages to a player in the
                          standard view mode                              */
int    iRecvStep;        /*!< recv_step: the length of the interval (in ms)
                          for accepting commands from a player            */
int    iHalfTime;        /*!< half_time: the length (in seconds) of a single
                          game half                                       */
```

```
int    iDropBallTime;     /*!< drop_ball_time: the number of cycles to wait
                          until dropping the ball automatically for free
                          kicks, corner kicks, etc.                */

// speech-related parameters
int    iSayMsgSize;       /*!< say_msg_size: the maximum length (in bytes) of
                          a spoken message                         */

// offside-related parameters
bool   bUseOffside;       /*!< use_offside: a boolean flag indicating whether
                          the offside rule should be applied or not   */
double dOffsideActiveAreaSize;/*!< offside_active_area_size: offside active
                            area size, i.e. radius of circle around
                            the ball in which player can be offside */
bool   bForbidKickOffOffside; /*!< forbid_kick_off_offside: a boolean flag
                            indicating whether a kick from offside
                            position is allowed                  */
double dOffsideKickMargin;/*!< offside_kick_margin: offside kick margin,
                          i.e. the minimum distance to the ball for
                          offending players when a free kick for
                          offside is taken                         */

// log-related parameters
bool   bVerbose;          /*!< verbose: flag indicating whether verbose mode
                          is active or not; in verbose mode server sends
                          extra error-information                  */
int    iRecordVersion;    /*!< record_version: the type of log record      */
bool   bRecordLog;        /*!< record_log: flag indicating whether log record
                          for game should be created               */
bool   bSendLog;          /*!< send_log: flag indicating whether send client
                          command log for game should be created      */
bool   bLogTimes;         /*!< log_times: flag indicating whether ms should
                          be written between cycles in log file       */
char   strLogFile[ 256 ]; /*!< server log to store all actions received    */
bool   bSynchMode;        /*!< synch_mode: indicates whether in sync. mode */
bool   bFullStateL;       /*!< fullstate_l: indicates full_state left team.*/
bool   bFullStateR;       /*!< fullstate_r: indicates full_state right team*/

// all the parameters available in player.conf (for heterogeneous players)
// NOTE: names in player.conf corresponding with member variables
//       are listed between doxygen-tags to enable quick searching

int    iPlayerTypes;           /*!< player_types: the number of player
                               types including the default types    */
int    iSubsMax;               /*!< subs_max: the maximum number of
                               substitutions allowed during a game;
```

```
                        the value also indicates the maximum
                        number of players allowed for each
                        type                          */
  double dPlayerSpeedMaxDeltaMin;  /*!< player_speed_max_delta_min: minimum
                        delta for adjusting player_speed_max */
  double dPlayerSpeedMaxDeltaMax;  /*!< player_speed_max_delta_max: maximum
                        delta for adjusting player_speed_max */
  double dStaminaIncMaxDeltaFactor;/*!< stamina_inc_max_delta_factor: amount by
                        which delta is multiplied for
                        stamina_inc_max               */
  double dPlayerDecayDeltaMin;     /*!< player_decay_delta_min: minimum delta
                        for adjusting player_decay          */
  double dPlayerDecayDeltaMax;     /*!< player_decay_delta_max: maximum delta
                        for adjusting player_decay          */
  double dInertiaMomentDeltaFactor;/*!< inertia_moment_delta_factor: amount by
                        which delta is multiplied for
                        inertia_moment                */
  double dDashPowerRateDeltaMin;   /*!< dash_power_rate_delta_min: minimum
                        delta for adjusting dash_power_rate  */
  double dDashPowerRateDeltaMax;   /*!< dash_power_rate_delta_max: maximum
                        delta for adjusting dash_power_rate  */
  double dPlayerSizeDeltaFactor;   /*!< player_size_delta_factor: amount delta
                        is multiplied by for player_size     */
  double dKickableMarginDeltaMin;  /*!< kickable_margin_delta_min: minimum
                        delta for adjusting kickable_margin  */
  double dKickableMarginDeltaMax;  /*!< kickable_margin_delta_max: maximum
                        delta for adjusting kickable_margin  */
  double dKickRandDeltaFactor;     /*!< kick_rand_delta_factor: amount delta is
                        multiplied by for kick_rand         */
  double dExtraStaminaDeltaMin;    /*!< extra_stamina_delta_min: minimum delta
                        for adjusting extra_stamina         */
  double dExtraStaminaDeltaMax;    /*!< extra_stamina_delta_max: maximum delta
                        for adjusting extra_stamina         */
  double dEffortMaxDeltaFactor;    /*!< effort_max_delta_factor: amount delta
                        is multiplied by for effort_max      */
  double dEffortMinDeltaFactor;    /*!< effort_min_delta_factor: amount delta
                        is multiplied by for effort_min      */
  double dNewDashPowerRateDeltaMin;/*!< new_dash_power_rate_delta_min: minimum
                                                                    delta for
adjusting dash_power_rate,
                                                                        used
from server 8.05              */
  double dNewDashPowerRateDeltaMax;/*!< new_dash_power_rate_delta_min: maximum
                        delta for adjusting dash_power_rate,
                        used from server 8.05              */
  double dNewStaminaIncMaxDeltaFactor;/*!< stamina_inc_max_delta_factor: amount
```

```
                        which delta is multiplied for
                        stamina_inc_max, used from server 8  */

  // other parameters
  int    iSlowDownFactor;         /*!< slow_down_factor: factor to slow down
                        simulator and send intervals         */
  double dVisibleDistance;        /*!< visible_distance: distance within which
                        objects are always 'visible' (even when
                        not in view cone)                    */
  double dExtraStamina;           /*!< extra_stamina: extra stamina for a
                        heterogeneous player                 */

  // penalty parameters
  double dPenDistX;               /*!< pen_dist_x: x distance for ball from
                        goalline.                            */
  double dPenMaxGoalieDistX;      /*!< pen_max_goalie_dist_x: max goalie
                        distance before the goalline.        */
  bool   bPenAllowMultKicks;      /*!< pen_allow_mult_kicks: allow multiple
                        kicks by the penalty kicker          */

  // tackle parameters
  double dTackleDist;             /*!< tackle_dist: distance in front player
                        where tackle is possible.            */
  double dTackleBackDist;         /*!< tackle_dist: distance at back of  player
                        where tackle is possible.            */
  double dTackleWidth;            /*!< tackle_width: distance to side of player
                        where tackle is possible.            */
  double dTackleExponent;         /*!< tackle_exponent: exponent need to
                        calculate prob.of success tackle     */
  int    iTackleCycles;           /*!< tackle_cycles: cycles immobile after
                        tackle                               */
  double dTacklePowerRate;        /*!< tacke_power_rate: acc. power tackle  */

  // parameters which depend on other values
  double dMaximalKickDist;        /*!< the maximum distance from a player for
                        which the ball is still kickable     */

  // public methods
public:
  // constructor for ServerSettings class
  ServerSettings( );

  // methods 'setValue' and 'readValues' from superclass
  // GenericValues are overridden in this subclass
  bool setValue  ( const char *strName    , const char *strValue       );
  bool readValues( const char *strFilename, const char *Separator      );
```

```
// set- and get methods for private member variables

// set- and get methods for goal-related parameters
bool   setGoalWidth              ( double d    );
double getGoalWidth               (             ) const;

// set- and get methods for player-related parameters
bool   setPlayerSize             ( double d    );
double getPlayerSize              (             ) const;
bool   setPlayerDecay             ( double d    );
double getPlayerDecay             (             ) const;
bool   setPlayerRand              ( double d    );
double getPlayerRand              (             ) const;
bool   setPlayerWeight            ( double d    );
double getPlayerWeight            (             ) const;
bool   setPlayerSpeedMax          ( double d    );
double getPlayerSpeedMax          (             ) const;
bool   setPlayerAccelMax          ( double d    );
double getPlayerAccelMax          (             ) const;

// set- and get methods for stamina-related parameters
bool   setStaminaMax              ( double d    );
double getStaminaMax              (             ) const;
bool   setStaminaIncMax           ( double d    );
double getStaminaIncMax           (             ) const;
bool   setRecoverDecThr           ( double d    );
double getRecoverDecThr           (             ) const;
bool   setRecoverDec              ( double d    );
double getRecoverDec              (             ) const;
bool   setRecoverMin              ( double d    );
double getRecoverMin              (             ) const;
bool   setEffortDecThr            ( double d    );
double getEffortDecThr            (             ) const;
bool   setEffortDec               ( double d    );
double getEffortDec               (             ) const;
bool   setEffortIncThr            ( double d    );
double getEffortIncThr            (             ) const;
bool   setEffortInc               ( double d    );
double getEffortInc               (             ) const;
bool   setEffortMin               ( double d    );
double getEffortMin               (             ) const;

// set- and get methods for parameters related to auditory perception
bool   setHearMax                 ( int i       );
int    getHearMax                 (             ) const;
```

```
bool   setHearInc                ( int i       );
int    getHearInc                (             ) const;
bool   setHearDecay              ( int i       );
int    getHearDecay              (             ) const;

// set- and get methods for parameters related to player turn actions
bool   setInertiaMoment          ( double d    );
double getInertiaMoment          (             ) const;

// set- and get methods for parameters related to sense_body information
bool   setSenseBodyStep          ( int i       )     ;
int    getSenseBodyStep          (             ) const;

// set- and get methods for goalkeeper-related parameters
bool   setCatchableAreaL         ( double d    );
double getCatchableAreaL         (             ) const;
bool   setCatchableAreaW         ( double d    );
double getCatchableAreaW         (             ) const;
bool   setCatchProbability       ( double d    );
double getCatchProbability       (             ) const;
bool   setCatchBanCycle          ( int    i    )     ;
int    getCatchBanCycle          (             ) const;
bool   setGoalieMaxMoves         ( int    i    )     ;
int    getGoalieMaxMoves         (             ) const;

// set- and get methods for ball-related parameters
bool   setBallSize               ( double d    );
double getBallSize               (             ) const;
bool   setBallDecay              ( double d    );
double getBallDecay              (             ) const;
bool   setBallRand               ( double d    );
double getBallRand               (             ) const;
bool   setBallWeight             ( double d    );
double getBallWeight             (             ) const;
bool   setBallSpeedMax           ( double d    );
double getBallSpeedMax           (             ) const;
bool   setBallAccelMax           ( double d    );
double getBallAccelMax           (             ) const;

// set- and get methods for wind-related parameters
bool   setWindForce              ( double d    );
double getWindForce              (             ) const;
bool   setWindDir                ( double d    );
double getWindDir                (             ) const;
bool   setWindRand               ( double d    );
double getWindRand               (             ) const;
```

```
bool   setWindRandom              ( bool   b    );
bool   getWindRandom              (             ) const;

// set- and get methods for parameters related to 'dash' and 'kick' commands
bool   setKickableMargin          ( double d    );
double getKickableMargin          (             ) const;
bool   setCkickMargin             ( double d    );
double getCkickMargin             (             ) const;
bool   setDashPowerRate           ( double d    );
double getDashPowerRate           (             ) const;
bool   setKickPowerRate           ( double d    );
double getKickPowerRate           (             ) const;
bool   setKickRand                ( double d    );
double getKickRand                (             ) const;

// set- and get methods for parameters related
// to visual and auditory perception range
bool   setVisibleAngle            ( double d    );
double getVisibleAngle            (             ) const;
bool   setAudioCutDist            ( double d    );
double getAudioCutDist            (             ) const;

// set- and get methods for quantization parameters
bool   setQuantizeStep            ( double d    );
double getQuantizeStep            (             ) const;
bool   setQuantizeStepL           ( double d    );
double getQuantizeStepL           (             ) const;

// set- and get methods for range parameters for basic actuator commands
bool setMaxPower                  ( int    i    )     ;
int  getMaxPower                  (             ) const;
bool setMinPower                  ( int    i    )     ;
int  getMinPower                  (             ) const;
bool setMaxMoment                 ( int    i    )     ;
int  getMaxMoment                 (             ) const;
bool setMinMoment                 ( int    i    )     ;
int  getMinMoment                 (             ) const;
bool setMaxNeckMoment             ( int    i    )     ;
int  getMaxNeckMoment             (             ) const;
bool setMinNeckMoment             ( int    i    )     ;
int  getMinNeckMoment             (             ) const;
bool setMaxNeckAng                ( int    i    )     ;
int  getMaxNeckAng                (             ) const;
bool setMinNeckAng                ( int    i    )     ;
int  getMinNeckAng                (             ) const;
```

```
// set- and get methods for port-related parameters
bool setPort                ( int  i  )   ;
int  getPort                (          ) const;
bool setCoachPort           ( int  i  )   ;
int  getCoachPort           (          ) const;
bool setOlCoachPort         ( int  i  )   ;
int  getOlCoachPort         (          ) const;

// set- and get methods for coach-related parameters
bool setSayCoachCntMax      ( int  i  )   ;
int  getSayCoachCntMax      (          ) const;
bool setSayCoachMsgSize     ( int  i  )   ;
int  getSayCoachMsgSize     (          ) const;
bool setClangWinSize        ( int  i  )   ;
int  getClangWinSize        (          ) const;
bool setClangDefineWin      ( int  i  )   ;
int  getClangDefineWin      (          ) const;
bool setClangMetaWin        ( int  i  )   ;
int  getClangMetaWin        (          ) const;
bool setClangAdviceWin      ( int  i  )   ;
int  getClangAdviceWin      (          ) const;
bool setClangInfoWin        ( int  i  )   ;
int  getClangInfoWin        (          ) const;
bool setClangMessDelay      ( int  i  )   ;
int  getClangMessDelay      (          ) const;
bool setClangMessPerCycle   ( int  i  )   ;
int  getClangMessPerCycle   (          ) const;
bool setSendViStep          ( int  i  )   ;
int  getSendViStep          (          ) const;

// set- and get methods for time-related parameters
bool   setSimulatorStep     ( int  i  )   ;
int    getSimulatorStep     (          ) const;
bool   setSendStep          ( int  i  )   ;
int    getSendStep          (          ) const;
bool   setRecvStep          ( int  i  )   ;
int    getRecvStep          (          ) const;
bool   setHalfTime          ( int  i  )   ;
int    getHalfTime          (          ) const;
bool   setDropBallTime      ( int  i  )   ;
int    getDropBallTime      (          ) const;

// set- and get methods for speech-related parameters
bool   setSayMsgSize        ( int  i  )   ;
int    getSayMsgSize        (          ) const;
```

```
// set- and get methods for offside-related parameters
bool   setUseOffside              ( bool   b    )     ;
bool   getUseOffside              (            ) const;
bool   setOffsideActiveAreaSize   ( double d    )     ;
double getOffsideActiveAreaSize   (            ) const;
bool   setForbidKickOffOffside    ( bool   b    )     ;
bool   getForbidKickOffOffside    (            ) const;
bool   setOffsideKickMargin       ( double d    )     ;
double getOffsideKickMargin       (            ) const;

// set- and get methods for log-related parameters
bool   setVerbose                 ( bool   b    )     ;
bool   getVerbose                 (            ) const;
bool   setRecordVersion           ( int    i    )     ;
int    getRecordVersion           (            ) const;
bool   setRecordLog               ( bool   b    )     ;
bool   getRecordLog               (            ) const;
bool   setSendLog                 ( bool   b    )     ;
bool   getSendLog                 (            ) const;
bool   setLogTimes                ( bool   b    )     ;
bool   getLogTimes                (            ) const;
bool   setLogFile                 ( char   *str )     ;
char*  getLogFile                 (            )     ;
bool   setSynchMode               ( bool   b    )     ;
bool   getSynchMode               (            ) const;
bool   setFullStateLeft           ( bool   b    )     ;
bool   getFullStateLeft           (            ) const;
bool   setFullStateRight          ( bool   b    )     ;
bool   getFullStateRight          (            ) const;

// set- and get methods for heterogeneous player parameters from player.conf
bool   setPlayerTypes             ( int    i    )     ;
int    getPlayerTypes             (            ) const;
bool   setSubsMax                 ( int    i    )     ;
int    getSubsMax                 (            ) const;
bool   setPlayerSpeedMaxDeltaMin  ( double d    )     ;
double getPlayerSpeedMaxDeltaMin  (            ) const;
bool   setPlayerSpeedMaxDeltaMax  ( double d    )     ;
double getPlayerSpeedMaxDeltaMax  (            ) const;
bool   setStaminaIncMaxDeltaFactor ( double d   )     ;
double getStaminaIncMaxDeltaFactor (           ) const;
bool   setPlayerDecayDeltaMin     ( double d    )     ;
double getPlayerDecayDeltaMin     (            ) const;
bool   setPlayerDecayDeltaMax     ( double d    )     ;
double getPlayerDecayDeltaMax     (            ) const;
bool   setInertiaMomentDeltaFactor ( double d   )     ;
```

```
double getInertiaMomentDeltaFactor   (            ) const;
bool   setDashPowerRateDeltaMin      ( double d    )     ;
double getDashPowerRateDeltaMin      (            ) const;
bool   setDashPowerRateDeltaMax      ( double d    )     ;
double getDashPowerRateDeltaMax      (            ) const;
bool   setPlayerSizeDeltaFactor      ( double d    )     ;
double getPlayerSizeDeltaFactor      (            ) const;
bool   setKickableMarginDeltaMin     ( double d    )     ;
double getKickableMarginDeltaMin     (            ) const;
bool   setKickableMarginDeltaMax     ( double d    )     ;
double getKickableMarginDeltaMax     (            ) const;
bool   setKickRandDeltaFactor        ( double d    )     ;
double getKickRandDeltaFactor        (            ) const;
bool   setExtraStaminaDeltaMin       ( double d    )     ;
double getExtraStaminaDeltaMin       (            ) const;
bool   setExtraStaminaDeltaMax       ( double d    )     ;
double getExtraStaminaDeltaMax       (            ) const;
bool   setEffortMaxDeltaFactor       ( double d    )     ;
double getEffortMaxDeltaFactor       (            ) const;
bool   setEffortMinDeltaFactor       ( double d    )     ;
double getEffortMinDeltaFactor       (            ) const;
bool   setNewDashPowerRateDeltaMin   ( double d    )     ;
double getNewDashPowerRateDeltaMin   (            ) const;
bool   setNewDashPowerRateDeltaMax   ( double d    )     ;
double getNewDashPowerRateDeltaMax   (            ) const;
bool   setNewStaminaIncMaxDeltaFactor( double d    )     ;
double getNewStaminaIncMaxDeltaFactor(            ) const;

// penalty parameters
bool   setPenDistX                ( double d    )     ;
double getPenDistX                (            ) const;
bool   setPenMaxGoalieDistX       ( double d    )     ;
double getPenMaxGoalieDistX       (            ) const;
bool   setPenAllowMultKicks       ( bool   b    )     ;
bool   getPenAllowMultKicks       (            ) const;

// tackle parameters
bool   setTackleDist              ( double d    )     ;
double getTackleDist              (            ) const;
bool   setTackleBackDist          ( double d    )     ;
double getTackleBackDist          (            ) const;
bool   setTackleWidth             ( double d    )     ;
double getTackleWidth             (            ) const;
bool   setTackleExponent          ( double d    )     ;
double getTackleExponent          (            ) const;
bool   setTackleCycles            ( int    i    )     ;
```

```
  int    getTackleCycles             (            ) const;
  bool   setTacklePowerRate          ( double d    )     ;
  double getTacklePowerRate          (            ) const;

  // set- and get methods for parameters not in server.conf or player.conf
  bool   setEffortMax                ( double d    )     ;
  double getEffortMax                (            ) const;
  bool   setSlowDownFactor           ( int    i    )     ;
  int    getSlowDownFactor           (            ) const;
  bool   setVisibleDistance          ( double d    )     ;
  double getVisibleDistance          (            ) const;
  bool   setExtraStamina             ( double d    )     ;
  double getExtraStamina             (            ) const;

  // set- and get methods for parameters which depend on other values
  bool   setMaximalKickDist          ( double d    )     ;
  double getMaximalKickDist          (            ) const;
};

/*************************************************************************
*/
/*********************   CLASS HETEROPLAYERTYPES
*****************************/
/*************************************************************************
*/

/*! This class contains all the SoccerServer parameters which together define a
    heterogeneous player type. For each player type these parameters are
    initialized when the server is started. */
class HeteroPlayerSettings
{
  // public member data
public:
  double dPlayerSpeedMax;   /*!< player_speed_max: maximum speed of a player */
  double dStaminaIncMax;    /*!< stamina_inc_max: maximum stamina increase of a
                             player per cycle                          */
  double dPlayerDecay;      /*!< player_decay: player speed decay per cycle  */
  double dInertiaMoment;    /*!< inertia_moment: inertia moment of a player;
                             affects actual turn angle depending on speed*/
  double dDashPowerRate;    /*!< dash_power_rate: rate by which the 'Power'
                             argument in a 'dash' command is multiplied
                             (thus determining the amount of displacement
                             of the player as a result of the 'dash')    */
  double dPlayerSize;       /*!< dash_power_rate: rate by which the 'Power'
                             argument in a 'dash' command is multiplied
                             (thus determining the amount of displacement
```

```
                        of the player as a result of the 'dash')    */
  double dKickableMargin;   /*!< kickable_margin: margin around player in which
                        ball is kickable; kickable area thus equals
                        kickable_margin + ball_size + player_size   */
  double dKickRand;         /*!< kick_rand: random error in kick direction   */
  double dExtraStamina;     /*!< extra_stamina: extra stamina for heterogeneous
                        player                                      */
  double dEffortMax;        /*!< effort_max: maximum value for player effort */
  double dEffortMin;        /*!< effort_min: minimum value for player effort */

  double dMaximalKickDist;  /*!< the maximum distance from a player for
                        which the ball is still kickable            */


  void show( ostream &os = cout );
};

#endif
```

ServerSettings.cpp

```
#include "ServerSettings.h"
#include <stdio.h>
#include <string.h>          // needed for 'strcpy'

/*****************************************************************************
/
/*******************   CLASS SERVERSETTINGS
*********************************/
/*****************************************************************************
/

/*! Constructor for the ServerSettings class. It sets all the private member
    variables in this class to the values specified in the configuration files
    (server.conf and player.conf) of Soccer Server version 8.xx. These values
    can be changed by calling the method 'readValues' with a new configuration
    file or by calling the method 'setValue' for a specific variable. */
ServerSettings::ServerSettings( ):GenericValues( "ServerSettings", 121 )
{
  // goal-related parameters
  dGoalWidth        = 14.02; // goal_width: the width of the goal

  // player-related parameters
  dPlayerSize       = 0.3;   // player_size: the size (=radius) of a player
  dPlayerDecay      = 0.4;   // player_decay: player speed decay per cycle
  dPlayerRand       = 0.1;   // player_rand: random error in player movement
  dPlayerWeight     = 60.0;  // player_weight: weight of a player (for wind)
  dPlayerSpeedMax   = 1.2;   // player_speed_max: maximum speed of a player
  dPlayerAccelMax   = 1.0;   // player_accel_max: maximum acceleration of a
                             // player per cycle

  // stamina-related parameters
  dStaminaMax       = 4000.0;// stamina_max: maximum stamina of a player
  dStaminaIncMax    = 45.0;  // stamina_inc_max: maximum stamina increase of a
                             // player per cycle
  dRecoverDecThr    = 0.3;   // recover_dec_thr: percentage of stamina_max
                             // below which player recovery decreases
  dRecoverDec       = 0.002; // recover_dec: decrement step per cycle for
                             // player recovery
  dRecoverMin       = 0.5;   // recover_min: minimum player recovery
  dEffortDecThr     = 0.3;   // effort_dec_thr: % of stamina_max below
                             // which player effort capacity decreases
  dEffortDec        = 0.005; // effort_dec: decrement step per cycle for
                             // player effort capacity
  dEffortIncThr     = 0.6;   // effort_incr_thr: percentage of stamina_max
                             // above which player effort capacity increases
  dEffortInc        = 0.01;  // effort_inc: increment step per cycle for
```

```
                    // player effort capacity
dEffortMin        = 0.6;   // effort_min: minimum value for player effort

// parameters related to auditory perception
iHearMax          = 2;     // hear_max: max hearing capacity of a player;
                    // a player can hear hear_inc messages in
                    // hear_decay simulation cycles
iHearInc          = 1;     // hear_inc: min hearing capacity of a player,
                    // i.e. the number of messages a player can hear
                    // in hear_decay simulation cycles
iHearDecay        = 2;     // hear_decay: decay rate of player hearing
                    // capacity, i.e. minimum number of cycles for
                    // hear_inc messages

// parameters related to player turn actions
dInertiaMoment    = 5.0;   // inertia_moment: inertia moment of a player;
                    // affects actual turn angle depending on speed

// parameters related to sense_body information
iSenseBodyStep    = 100;   // sense_body_step: length of the interval (ms)
                    // between sense_body information messages

// goalkeeper-related parameters
dCatchableAreaL   = 2.0;   // catchable_area_l: length of area around
                    // goalkeeper in which he can catch the ball
dCatchableAreaW   = 1.0;   // catchable_area_w: width of area around
                    // goalkeeper in which he can catch the ball
dCatchProbability = 1.0;   // catch_probability: the probability for a
                    // goalkeeper to catch the ball
iCatchBanCycle    = 5 ;    // catch_ban_cycle: number of cycles after catch
                    // in which goalkeeper cannot catch again
iGoalieMaxMoves   = 2;     // goalie_max_moves: maximum number of 'move'
                    // actions allowed for goalkeeper after catch

// ball-related parameters
dBallSize         = 0.085; // ball_size: the size (=radius) of the ball
dBallDecay        = 0.94;  // ball_decay: ball speed decay per cycle
dBallRand         = 0.05;  // ball_rand: random error in ball movement
dBallWeight       = 0.2;   // ball_weight: weight of the ball (for wind)
dBallSpeedMax     = 2.7;   // ball_speed_max: maximum speed of the ball
dBallAccelMax     = 2.7;   // ball_accel_max: maximum acceleration of the
                    // ball per cycle

// wind-related parameters
dWindForce        = 0.0;   // wind_force: the force of the wind
dWindDir          = 0.0;   // wind_dir: the direction of the wind
```

```
dWindRand         = 0.0;   // wind_rand: random error in wind direction
bWindRandom       = false; // wind_random: is wind force and dir random

// parameters related to 'dash' and 'kick' commands
dKickableMargin   = 0.7;   // kickable_margin: margin around player in which
                           // ball is kickable; kickable area thus equals
                           // kickable_margin + ball_size + player_size
dCkickMargin      = 1.0;   // ckick_margin: corner kick margin, i.e. the
                           // minimum distance to the ball for offending
                           // players when a corner kick is taken

dDashPowerRate    = 0.006; // dash_power_rate: rate by which the 'Power'
                           // argument in a 'dash' command is multiplied
                           // (thus determining the amount of displacement
                           // of the player as a result of the 'dash')
dKickPowerRate    = 0.027; // kick_power_rate: rate by which the 'Power'
                           // argument in a 'kick' command is multiplied
                           // (thus determining the amount of displacement
                           // of the ball as a result of the 'kick')
dKickRand         = 0.0;   // kick_rand: random error in kick direction

// parameters related to visual and auditory perception range
dVisibleAngle     = 90.0;  // visible_angle: angle of the view cone of a
                           // player in the standard view mode
dAudioCutDist     = 50.0;  // audio_cut_dist: maximum distance over which a
                           // spoken message can be heard

// quantization parameters
dQuantizeStep     = 0.1;   // quantize_step: quantization step for distance
                           // of moving objects
dQuantizeStepL    = 0.01;  // quantize_step_l: quantization step for dist
                           // of landmarks

// range parameters for basic actuator commands
iMaxPower         = 100;   // maxpower: maximum power for dash/kick
iMinPower         = -100;  // minpower: minimum power for dash/kick
iMaxMoment        = 180;   // maxmoment: maximum angle for turn/kick
iMinMoment        = -180;  // minmoment: minimum angle for turn/kick
iMaxNeckMoment    = 180;   // maxneckmoment: maximum angle for turnneck
iMinNeckMoment    = -180;  // minneckmoment: minimum angle for turnneck
iMaxNeckAng       = 90;    // maxneckang: maximum neck angle rel to body
iMinNeckAng       = -90;   // minneckang: minimum neck angle rel to body

// port-related parameters
iPort             = 6000;  // port: port number for player connection
iCoachPort        = 6001;  // coach_port: port number for coach connection
```

```
iOlCoachPort      = 6002;  // ol_coach_port: port number for online coach

// coach-related parameters
iSayCoachCntMax    = 128;   // say_coach_cnt_max: maximum number of coach
                    // messages possible
iSayCoachMsgSize   = 128;   // say_coach_msg_size: maximum size of coach
                    // messages
iClangWinSize      = 300;   // clang_win_size: time window which controls how
                    // many coach messages can be sent
iClangDefineWin    = 1;     // clang_define_win: number of define messages by
                    // coach per time window
iClangMetaWin      = 1;     // clang_meta_win: number of meta messages by
                    // coach per time window
iClangAdviceWin    = 1;     // clang_advice_win: number of advice messages by
                    // coach per time window
iClangInfoWin      = 1;     // clang_info_win: number of info messages by
                    // coach per time window
iClangMessDelay    = 50;    // clang_mess_delay: delay of coach messages, ie
                    // the number of cycles between send to player
                    // and receival of message
iClangMessPerCycle = 1;     // clang_mess_per_cycle: number of coach messages
                    // per cycle
iSendViStep        = 100;   // send_vi_step: interval of coach's look, i.e.
                    // the length of the interval (in ms) between
                    // visual messages to the coach

// time-related parameters
iSimulatorStep     = 100;   // simulator_step: the length (in ms) of a
                    // simulator cycle
iSendStep          = 150;   // send_step: the length of the interval (in ms)
                    // between visual messages to a player in the
                    // standard view mode
iRecvStep          = 10;    // recv_step: the length of the interval (in ms)
                    // for accepting commands from a player
iHalfTime          = 300;   // half_time: the length (in seconds) of a single
                    // game half
iDropBallTime      = 200;   // drop_ball_time: the number of cycles to wait
                    // until dropping the ball automatically for free
                    // kicks, corner kicks, etc.

// speech-related parameters
iSayMsgSize        = 512;   // say_msg_size: the maximum length (in bytes) of
                    // a spoken message

// offside-related parameters
bUseOffside            = true; // use_offside: a boolean flag indicating
```

```
                    // whether the offside rule should be applied
                    // or not
dOffsideActiveAreaSize = 5.0;  // offside_active_area_size: offside active
                    // area size, i.e. radius of circle around
                    // the ball in which player can be offside
bForbidKickOffOffside  = true; // forbid_kick_off_offside: a boolean flag
                    // indicating whether a kick from offside
                    // position is allowed
dOffsideKickMargin     = 9.15; // offside_kick_margin: offside kick margin
                    // i.e. the minimum distance to the ball for
                    // offending players when a free kick for
                    // offside is taken

// log-related parameters
bVerbose          = false; // verbose: flag indicating whether verbose mode
                    // is active or not; in verbose mode server sends
                    // extra error-information
iRecordVersion     = 3;     // record_version: the type of log record
bRecordLog         = true;  // record_log: flag indicating whether log record
                    // for game should be created
bSendLog          = true;  // send_log: flag indicating whether send client
                    // command log for game should be created
bLogTimes          = false; // log_times: flag indicating whether ms should
                    // be written between cycles in log file
strcpy( strLogFile, "server.log" );// server log to store all actions receiv
bSynchMode         = false; // synch_mode: in synchronization mode?
bFullStateL        = false; // fullstate_l, full information left team
bFullStateR        = false; // fulstate_r, full information right team

// heterogeneous player parameters from player.conf
iPlayerTypes             = 7;     // player_types: the number of player type
                         // including the default player type
iSubsMax                 = 3;     // subs_max: the maximum number of
                         // substitutions allowed during a game the
                         // value also indicates the maximum number
                         // of players allowed for each type
dPlayerSpeedMaxDeltaMin   = 0.0;   // player_speed_max_delta_min: minimum
                         // delta for adjusting player_speed_max
dPlayerSpeedMaxDeltaMax   = 0.0;   // player_speed_max_delta_max: maximum
                         // delta for adjusting player_speed_max
dStaminaIncMaxDeltaFactor = 0.0;   // stamina_inc_max_delta_factor: amount by
                         // which delta is multiplied for
                         // stamina_inc_max
dPlayerDecayDeltaMin      = 0.0;   // player_decay_delta_min: minimum delta
                         // for adjusting player_decay
dPlayerDecayDeltaMax      = 0.2;   // player_decay_delta_max: maximum delta
```

```
                        // for adjusting player_decay
  dInertiaMomentDeltaFactor = 25.0;  // inertia_moment_delta_factor: amount by
                        // which delta is multiplied for
                        // inertia_moment
  dDashPowerRateDeltaMin    = 0.0;   // dash_power_rate_delta_min: min delta
                        // for adjusting dash_power_rate
  dDashPowerRateDeltaMax    = 0.0;   // dash_power_rate_delta_max: max delta
                        // for adjusting dash_power_rate
  dPlayerSizeDeltaFactor    = -100.0;// player_size_delta_factor: amount delta
                        // is multiplied by for player_size
  dKickableMarginDeltaMin   = 0.0;   // kickable_margin_delta_min: min delta
                        // for adjusting kickable_margin
  dKickableMarginDeltaMax   = 0.2;   // kickable_margin_delta_max: max delta
                        // for adjusting kickable_margin
  dKickRandDeltaFactor      = 0.5;   // kick_rand_delta_factor: amount delta is
                        // multiplied by for kick_rand
  dExtraStaminaDeltaMin     = 0.0;   // extra_stamina_delta_min: minimum delta
                        // for adjusting extra_stamina
  dExtraStaminaDeltaMax     = 100.0; // extra_stamina_delta_max: maximum delta
                        // for adjusting extra_stamina
  dEffortMaxDeltaFactor     = -0.002;// effort_max_delta_factor: amount delta
                        // is multiplied by for effort_max
  dEffortMinDeltaFactor     = -0.002;// effort_min_delta_factor: amount delta
                        // is multiplied by for effort_min
  dNewDashPowerRateDeltaMin = 0.0;   // dash_power_rate_delta_min: minimum
                        // delta for adjusting dash_power_rate
  dNewDashPowerRateDeltaMax = 0.002; // dash_power_rate_delta_max: maximum
                        // delta for adjusting dash_power_rate
  dNewStaminaIncMaxDeltaFactor=-100000.0;// stamina_inc_max_delta_factor:
                                                                        // amount by
which delta is multiplied for
                        // stamina_inc_max

  // important server parameters not in server.conf or player.conf
  dEffortMax        = 1.0; // effort_max: maximum player effort capacity
  iSlowDownFactor    = 1;   // slow_down_factor: factor to slow down simulator
                    // and send intervals
  dVisibleDistance   = 3.0; // visible_distance: distance within which objects
                    // are always 'visible' even when not in view cone)
  dExtraStamina      = 0.0; // extra_stamina: extra stamina for heterogeneous
                    // player

  // penalty parameters
  dPenDistX         = 11.0;         // pen_dist_x: distance ball from goal
  dPenMaxGoalieDistX = 4.0;         // pen_goalie_max_dist_x: maximum distance
                        // for goalie during penalties
```

```
bPenAllowMultKicks = false;       // pen_allow_mult_kicks: can penalty
                                  // kicker kick multiple times

// tackle parameters
dTackleDist       = 2.0;          // tackle_dist: allowed distance in front
dTackleBackDist   = 0.5;          // tackle_back_dist: allowed distance back
dTackleWidth      = 1.0;          // tackle_width: allowed distance side
dTackleExponent   = 6;            // tackle_exponent: exponent need
iTackleCycles     = 10;           // tackle_cycles: cycles immobile
dTacklePowerRate  = 0.027;        // tackle_power_rate: power rate tackle

// parameters which depend on other values
dMaximalKickDist = dKickableMargin +   // the maximum distance from a player
                   dPlayerSize +       // for which the ball is kickable
                   dBallSize;

// add all the settings, i.e. each parameter
// becomes a new generic value for the class

// goal-related parameters
addSetting( "goal_width"       , &dGoalWidth       , GENERIC_VALUE_DOUBLE );

// player-related parameters
addSetting( "player_size"      , &dPlayerSize      , GENERIC_VALUE_DOUBLE );
addSetting( "player_decay"     , &dPlayerDecay     , GENERIC_VALUE_DOUBLE );
addSetting( "player_rand"      , &dPlayerRand      , GENERIC_VALUE_DOUBLE );
addSetting( "player_weight"    , &dPlayerWeight    , GENERIC_VALUE_DOUBLE );
addSetting( "player_speed_max" , &dPlayerSpeedMax  , GENERIC_VALUE_DOUBLE );
addSetting( "player_accel_max" , &dPlayerAccelMax  , GENERIC_VALUE_DOUBLE );

// stamina-related parameters
addSetting( "stamina_max"      , &dStaminaMax      , GENERIC_VALUE_DOUBLE );
addSetting( "stamina_inc_max"  , &dStaminaIncMax   , GENERIC_VALUE_DOUBLE );
addSetting( "recover_dec_thr"  , &dRecoverDecThr   , GENERIC_VALUE_DOUBLE );
addSetting( "recover_dec"      , &dRecoverDec      , GENERIC_VALUE_DOUBLE );
addSetting( "recover_min"      , &dRecoverMin      , GENERIC_VALUE_DOUBLE );
addSetting( "effort_dec_thr"   , &dEffortDecThr    , GENERIC_VALUE_DOUBLE );
addSetting( "effort_dec"       , &dEffortDec       , GENERIC_VALUE_DOUBLE );
addSetting( "effort_inc_thr"   , &dEffortIncThr    , GENERIC_VALUE_DOUBLE );
addSetting( "effort_inc"       , &dEffortInc       , GENERIC_VALUE_DOUBLE );
addSetting( "effort_min"       , &dEffortMin       , GENERIC_VALUE_DOUBLE );

// parameters related to auditory perception
addSetting( "hear_max"         , &iHearMax         , GENERIC_VALUE_INTEGER);
addSetting( "hear_inc"         , &iHearInc         , GENERIC_VALUE_INTEGER);
addSetting( "hear_decay"       , &iHearDecay       , GENERIC_VALUE_INTEGER);
```

```
// parameters related to player turn actions
addSetting( "inertia_moment"    , &dInertiaMoment   , GENERIC_VALUE_DOUBLE );

// parameters related to sense_body information
addSetting( "sense_body_step"   , &iSenseBodyStep   , GENERIC_VALUE_INTEGER);

// goalkeeper-related parameters
addSetting( "catchable_area_l"  , &dCatchableAreaL  , GENERIC_VALUE_DOUBLE );
addSetting( "catchable_area_w"  , &dCatchableAreaW  , GENERIC_VALUE_DOUBLE );
addSetting( "catch_probability" , &dCatchProbability, GENERIC_VALUE_DOUBLE );
addSetting( "catch_ban_cycle"   , &iCatchBanCycle   , GENERIC_VALUE_INTEGER);
addSetting( "goalie_max_moves"  , &iGoalieMaxMoves  , GENERIC_VALUE_INTEGER);

// ball-related parameters
addSetting( "ball_size"         , &dBallSize        , GENERIC_VALUE_DOUBLE );
addSetting( "ball_decay"        , &dBallDecay       , GENERIC_VALUE_DOUBLE );
addSetting( "ball_rand"         , &dBallRand        , GENERIC_VALUE_DOUBLE );
addSetting( "ball_weight"       , &dBallWeight      , GENERIC_VALUE_DOUBLE );
addSetting( "ball_speed_max"    , &dBallSpeedMax    , GENERIC_VALUE_DOUBLE );
addSetting( "ball_accel_max"    , &dBallAccelMax    , GENERIC_VALUE_DOUBLE );

// wind-related parameters
addSetting( "wind_force"        , &dWindForce       , GENERIC_VALUE_DOUBLE );
addSetting( "wind_dir"          , &dWindDir         , GENERIC_VALUE_DOUBLE );
addSetting( "wind_rand"         , &dWindRand        , GENERIC_VALUE_DOUBLE );
addSetting( "wind_random"       , &bWindRandom      , GENERIC_VALUE_BOOLEAN);
// parameters related to 'dash' and 'kick' commands
addSetting( "kickable_margin"   , &dKickableMargin  , GENERIC_VALUE_DOUBLE );
addSetting( "ckick_margin"      , &dCkickMargin     , GENERIC_VALUE_DOUBLE );
addSetting( "dash_power_rate"   , &dDashPowerRate   , GENERIC_VALUE_DOUBLE );
addSetting( "kick_power_rate"   , &dKickPowerRate   , GENERIC_VALUE_DOUBLE );
addSetting( "kick_rand"         , &dKickRand        , GENERIC_VALUE_DOUBLE );

// parameters related to visual and auditory perception range
addSetting( "visible_angle"     , &dVisibleAngle    , GENERIC_VALUE_DOUBLE );
addSetting( "audio_cut_dist"    , &dAudioCutDist    , GENERIC_VALUE_DOUBLE );

// quantization parameters
addSetting( "quantize_step"     , &dQuantizeStep    , GENERIC_VALUE_DOUBLE );
addSetting( "quantize_step_l"   , &dQuantizeStepL   , GENERIC_VALUE_DOUBLE );

// range parameters for basic actuator commands
addSetting( "maxpower"          , &iMaxPower        , GENERIC_VALUE_INTEGER);
addSetting( "minpower"          , &iMinPower        , GENERIC_VALUE_INTEGER);
addSetting( "maxmoment"         , &iMaxMoment       , GENERIC_VALUE_INTEGER);
```

```
addSetting( "minmoment"         , &iMinMoment       , GENERIC_VALUE_INTEGER);
addSetting( "maxneckmoment"     , &iMaxNeckMoment   , GENERIC_VALUE_INTEGER);
addSetting( "minneckmoment"     , &iMinNeckMoment   , GENERIC_VALUE_INTEGER);
addSetting( "maxneckang"        , &iMaxNeckAng      , GENERIC_VALUE_INTEGER);
addSetting( "minneckang"        , &iMinNeckAng      , GENERIC_VALUE_INTEGER);

// port-related parameters
addSetting( "port"              , &iPort            , GENERIC_VALUE_INTEGER);
addSetting( "coach_port"        , &iCoachPort       , GENERIC_VALUE_INTEGER);
addSetting( "ol_coach_port"     , &iOlCoachPort     , GENERIC_VALUE_INTEGER);

// coach-related parameters
addSetting( "say_coach_cnt_max" , &iSayCoachCntMax  , GENERIC_VALUE_INTEGER);
addSetting( "say_coach_msg_size", &iSayCoachMsgSize , GENERIC_VALUE_INTEGER);
addSetting( "clang_win_size"    , &iClangWinSize    , GENERIC_VALUE_INTEGER);
addSetting( "clang_define_win"  , &iClangDefineWin  , GENERIC_VALUE_INTEGER);
addSetting( "clang_meta_win"    , &iClangMetaWin    , GENERIC_VALUE_INTEGER);
addSetting( "clang_advice_win"  , &iClangAdviceWin  , GENERIC_VALUE_INTEGER);
addSetting( "clang_info_win"    , &iClangInfoWin    , GENERIC_VALUE_INTEGER);
addSetting( "clang_mess_delay"  , &iClangMessDelay  , GENERIC_VALUE_INTEGER);
addSetting("clang_mess_per_cycle",&iClangMessPerCycle,GENERIC_VALUE_INTEGER);
addSetting( "send_vi_step"      , &iSendViStep      , GENERIC_VALUE_INTEGER);

// time-related parameters
addSetting( "simulator_step"    , &iSimulatorStep   , GENERIC_VALUE_INTEGER);
addSetting( "send_step"         , &iSendStep        , GENERIC_VALUE_INTEGER);
addSetting( "recv_step"         , &iRecvStep        , GENERIC_VALUE_INTEGER);
addSetting( "half_time"         , &iHalfTime        , GENERIC_VALUE_INTEGER);
addSetting( "drop_ball_time"    , &iDropBallTime    , GENERIC_VALUE_INTEGER);

// speech-related parameters
addSetting( "say_msg_size"      , &iSayMsgSize      , GENERIC_VALUE_INTEGER);

// offside-related parameters
addSetting( "use_offside"       , &bUseOffside      , GENERIC_VALUE_BOOLEAN);
addSetting( "offside_active_area_size", &dOffsideActiveAreaSize,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "forbid_kick_off_offside" , &bForbidKickOffOffside ,
                                    GENERIC_VALUE_BOOLEAN);
addSetting( "offside_kick_margin",&dOffsideKickMargin,GENERIC_VALUE_DOUBLE );

// log-related parameters
addSetting( "verbose"           , &bVerbose         , GENERIC_VALUE_BOOLEAN);
addSetting( "record_version"    , &iRecordVersion   , GENERIC_VALUE_INTEGER);
addSetting( "record_log"        , &bRecordLog       , GENERIC_VALUE_BOOLEAN);
addSetting( "send_log"          , &bSendLog         , GENERIC_VALUE_BOOLEAN);
```

```
addSetting( "log_times"         , &bLogTimes        , GENERIC_VALUE_BOOLEAN);
addSetting( "log_file"          , &strLogFile       , GENERIC_VALUE_STRING );
addSetting( "synch_mode"        , &bSynchMode       , GENERIC_VALUE_BOOLEAN);
addSetting( "fullstate_l"       , &bFullStateL      , GENERIC_VALUE_BOOLEAN);
addSetting( "fullstate_r"       , &bFullStateR      , GENERIC_VALUE_BOOLEAN);

// heterogeneous player parameters from player.conf
addSetting( "player_types"      , &iPlayerTypes     , GENERIC_VALUE_INTEGER);
addSetting( "subs_max"          , &iSubsMax         , GENERIC_VALUE_INTEGER);
addSetting( "player_speed_max_delta_min"  , &dPlayerSpeedMaxDeltaMin  ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "player_speed_max_delta_max"  , &dPlayerSpeedMaxDeltaMax  ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "stamina_inc_max_delta_factor", &dStaminaIncMaxDeltaFactor,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "player_decay_delta_min"      , &dPlayerDecayDeltaMin     ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "player_decay_delta_max"      , &dPlayerDecayDeltaMax     ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "inertia_moment_delta_factor" , &dInertiaMomentDeltaFactor,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "dash_power_rate_delta_min"   , &dDashPowerRateDeltaMin   ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "dash_power_rate_delta_max"   , &dDashPowerRateDeltaMax   ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "player_size_delta_factor"    , &dPlayerSizeDeltaFactor   ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "kickable_margin_delta_min"   , &dKickableMarginDeltaMin  ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "kickable_margin_delta_max"   , &dKickableMarginDeltaMax  ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "kick_rand_delta_factor"      , &dKickRandDeltaFactor     ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "extra_stamina_delta_min"     , &dExtraStaminaDeltaMin    ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "extra_stamina_delta_max"     , &dExtraStaminaDeltaMax    ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "effort_max_delta_factor"     , &dEffortMaxDeltaFactor    ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "effort_min_delta_factor"     , &dEffortMinDeltaFactor    ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "new_dash_power_rate_delta_min", &dNewDashPowerRateDeltaMin   ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "new_dash_power_rate_delta_max", &dNewDashPowerRateDeltaMax   ,
                                    GENERIC_VALUE_DOUBLE );
addSetting( "new_stamina_inc_max_delta_factor", &dNewStaminaIncMaxDeltaFactor
```

```
                                        ,GENERIC_VALUE_DOUBLE  );

  // penalty parameters
  addSetting( "pen_dist_x"        , &dPenDistX        , GENERIC_VALUE_DOUBLE );

addSetting("pen_goalie_max_dist_x",&dPenMaxGoalieDistX,GENERIC_VALUE_DOUBLE);

addSetting("pen_allow_mult_kicks",&bPenAllowMultKicks,GENERIC_VALUE_BOOLEAN);

  // tackle parameters
  addSetting( "tackle_dist"       , &dTackleDist      , GENERIC_VALUE_DOUBLE );
  addSetting( "tackle_back_dist"  , &dTackleBackDist  , GENERIC_VALUE_DOUBLE );
  addSetting( "tackle_width"      , &dTackleWidth     , GENERIC_VALUE_DOUBLE );
  addSetting( "tackle_exponent"   , &dTackleExponent  , GENERIC_VALUE_DOUBLE );
  addSetting( "tackle_cycles"     , &iTackleCycles    , GENERIC_VALUE_INTEGER);
  addSetting( "tackle_power_rate" , &dTacklePowerRate , GENERIC_VALUE_DOUBLE );

  // important server parameters not in server.conf or player.conf - are now
  addSetting( "effort_max"        , &dEffortMax       , GENERIC_VALUE_DOUBLE );
  addSetting( "slow_down_factor"  , &iSlowDownFactor  , GENERIC_VALUE_INTEGER);
  addSetting( "visible_distance"  , &dVisibleDistance , GENERIC_VALUE_DOUBLE );
  addSetting( "extra_stamina"     , &dExtraStamina    , GENERIC_VALUE_DOUBLE );

  // parameters which depend on other values
  addSetting( "dMaximalKickDist"  , &dMaximalKickDist , GENERIC_VALUE_DOUBLE );
}

/*! This method is originally defined in the superclass GenericValues and is
    overridden in this subclass. It sets the variable denoted by the first
    argument to the value denoted by the second argument.
    \param strName a string representing the name of a variable for which the
    value must be (re)set
    \param strValue a string representing a value which must be assigned to the
    variable denoted by the first argument
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setValue( const char *strName, const char *strValue )
{
  // call to the superclass method
  bool bReturn = GenericValues::setValue( strName, strValue );
  // compute values for parameters which depend on others (reason for override)
  dMaximalKickDist = ( dKickableMargin + dPlayerSize + dBallSize );

  return ( bReturn );
}

/*! This method is originally defined in the superclass GenericValues and is
```

```
    overridden in this subclass. It reads the values from a server
    configuration file and assigns them to the proper variables in this class.
    \param strFileName a string representing the name of a configuration file
    \param strSeparator a string representing the separator between the name of
    a variable and its value
    \return a boolean indicating whether the values were read correctly */
bool ServerSettings::readValues( const char *strFileName,
                                 const char *strSeparator )
{
  // call to the superclass method
  bool bReturn = GenericValues::readValues( strFileName, strSeparator );
  // compute values for parameters which depend on others (reason for override)
  dMaximalKickDist = ( dKickableMargin + dPlayerSize + dBallSize );

  return ( bReturn );
}

/*! Set method for the 'dGoalWidth' member variable.
    \param d a double value representing a new width of the goal
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setGoalWidth( double d )
{
  dGoalWidth = d;
  return ( true );
}

/*! Get method for the 'dGoalWidth' member variable.
    \return the width of the goal */
double ServerSettings::getGoalWidth( ) const
{
  return ( dGoalWidth );
}

/*! Set method for the 'dPlayerSize' member variable.
    \param d a double value representing a new player size
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerSize( double d )
{
  dPlayerSize = d;
  // NOTE: also update parameters for which the value depends on this variable
  dMaximalKickDist = ( dKickableMargin + dPlayerSize + dBallSize );

  return ( true );
}

/*! Get method for the 'dPlayerSize' member variable.
```

```
    \return the size (=radius) of a player */
double ServerSettings::getPlayerSize( ) const
{
  return ( dPlayerSize );
}

/*! Set method for the 'dPlayerDecay' member variable.
    \param d a double value representing a new player speed decay per cycle
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerDecay( double d )
{
  dPlayerDecay = d;
  return ( true );
}

/*! Get method for the 'dPlayerDecay' member variable.
    \return the player speed decay per cycle */
double ServerSettings::getPlayerDecay( ) const
{
  return ( dPlayerDecay );
}

/*! Set method for the 'dPlayerRand' member variable.
    \param d a double value representing a new random error in player movement
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerRand( double d )
{
  dPlayerRand = d;
  return ( true );
}

/*! Get method for the 'dPlayerRand' member variable.
    \return the random error in player movement */
double ServerSettings::getPlayerRand( ) const
{
  return ( dPlayerRand );
}

/*! Set method for the 'dPlayerWeight' member variable.
    \param d a double value representing a new weight of a player (for wind)
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerWeight( double d )
{
  dPlayerWeight = d;
  return ( true );
}
```

```
/*! Get method for the 'dPlayerWeight' member variable.
    \return the weight of a player (for wind) */
double ServerSettings::getPlayerWeight( ) const
{
  return ( dPlayerWeight );
}

/*! Set method for the 'dPlayerSpeedMax' member variable.
    \param d a double value representing a new maximum speed of a player
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerSpeedMax( double d )
{
  dPlayerSpeedMax = d;
  return ( true );
}

/*! Get method for the 'dPlayerSpeedMax' member variable.
    \return the maximum speed of a player */
double ServerSettings::getPlayerSpeedMax( ) const
{
  return ( dPlayerSpeedMax );
}

/*! Set method for the 'dPlayerAccelMax' member variable.
    \param d a double value representing a new maximum acceleration of a player
    per cycle
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerAccelMax( double d )
{
  dPlayerAccelMax = d;
  return ( true );
}

/*! Get method for the 'dPlayerAccelMax' member variable.
    \return the maximum acceleration of a player per cycle */
double ServerSettings::getPlayerAccelMax( ) const
{
  return ( dPlayerAccelMax );
}

/*! Set method for the 'dStaminaMax' member variable.
    \param d a double value representing a new maximum stamina of a player
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setStaminaMax( double d )
{
```

```
  dStaminaMax = d;
  return ( true );
}

/*! Get method for the 'dStaminaMax' member variable.
    \return the maximum stamina of a player */
double ServerSettings::getStaminaMax( ) const
{
  return ( dStaminaMax );
}

/*! Set method for the 'dStaminaIncMax' member variable.
    \param d a double value representing a new maximum stamina increase of a
    player per cycle
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setStaminaIncMax( double d )
{
  dStaminaIncMax = d;
  return ( true );
}

/*! Get method for the 'dStaminaIncMax' member variable.
    \return the maximum stamina increase of a player per cycle */
double ServerSettings::getStaminaIncMax( ) const
{
  return ( dStaminaIncMax );
}

/*! Set method for the 'dRecoverDecThr' member variable.
    \param d a double value representing a new percentage of stamina_max below
    which player recovery decreases
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setRecoverDecThr( double d )
{
  dRecoverDecThr = d;
  return ( true );
}

/*! Get method for the 'dRecoverDecThr' member variable.
    \return percentage of stamina_max below which player recovery decreases */
double ServerSettings::getRecoverDecThr( ) const
{
  return ( dRecoverDecThr );
}

/*! Set method for the 'dRecoverDec' member variable.
```

```
    \param d a double value representing a new decrement step per cycle for
    player recovery
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setRecoverDec( double d )
{
  dRecoverDec = d;
  return ( true );
}

/*! Get method for the 'dRecoverDec' member variable.
    \return the decrement step per cycle for player recovery */
double ServerSettings::getRecoverDec( ) const
{
  return ( dRecoverDec );
}

/*! Set method for the 'dRecoverMin' member variable.
    \param d a double value representing a new minimum player recovery
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setRecoverMin( double d )
{
  dRecoverMin = d;
  return ( true );
}

/*! Get method for the 'dRecoverMin' member variable.
    \return the minimum player recovery */
double ServerSettings::getRecoverMin( ) const
{
  return ( dRecoverMin );
}

/*! Set method for the 'dEffortDecThr' member variable.
    \param d a double value representing a new percentage of stamina_max below
    which player effort capacity decreases
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortDecThr( double d )
{
  dEffortDecThr = d;
  return ( true );
}

/*! Get method for the 'dEffortDecThr' member variable.
    \return the percentage of stamina_max below which player effort capacity
    decreases */
double ServerSettings::getEffortDecThr( ) const
```

```
{
  return ( dEffortDecThr );
}

/*! Set method for the 'dEffortDec' member variable.
    \param d a double value representing a new decrement step per cycle for
    player effort capacity
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortDec( double d )
{
  dEffortDec = d;
  return ( true );
}

/*! Get method for the 'dEffortDec' member variable.
    \return the decrement step per cycle for player effort capacity */
double ServerSettings::getEffortDec( ) const
{
  return ( dEffortDec );
}

/*! Set method for the 'dEffortIncThr' member variable.
    \param d a double value representing a new percentage of stamina_max above
    which player effort capacity increases
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortIncThr( double d )
{
  dEffortIncThr = d;
  return ( true );
}

/*! Get method for the 'dEffortIncThr' member variable.
    \return the percentage of stamina_max above which player effort capacity
    increases */
double ServerSettings::getEffortIncThr( ) const
{
  return ( dEffortIncThr );
}

/*! Set method for the 'dEffortInc' member variable.
    \param d a double value representing a new increment step per cycle for
    player effort capacity
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortInc( double d )
{
  dEffortInc = d;
```

```
  return ( true );
}

/*! Get method for the 'dEffortInc' member variable.
    \return the increment step per cycle for player effort capacity */
double ServerSettings::getEffortInc( ) const
{
  return ( dEffortInc );
}

/*! Set method for the 'dEffortMin' member variable.
    \param d a double value representing a new minimum value for player effort
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortMin( double d )
{
  dEffortMin = d;
  return ( true );
}

/*! Get method for the 'dEffortMin' member variable.
    \return the minimum value for player effort */
double ServerSettings::getEffortMin( ) const
{
  return ( dEffortMin );
}

/*! Set method for the 'iHearMax' member variable.

    \param i an integer value representing a new maximum hearing
    capacity of a player (a player can hear iHearInc messages in
    iHearDecay simulation cycles)

    \return a boolean indicating whether the update was successful */
bool ServerSettings::setHearMax( int i )
{
  iHearMax = i;
  return ( true );
}

/*! Get method for the 'iHearMax' member variable.

    \return the maximum hearing capacity of a player (a player can
    hear iHearInc messages in iHearDecay simulation cycles) */
int ServerSettings::getHearMax( ) const
{
  return ( iHearMax );
```

```
}

/*! Set method for the 'iHearInc' member variable.
    \param i an integer value representing a new minimum hearing capacity of a
    player, i.e. the number of messages a player can hear in iHearDecay
    simulation cycles
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setHearInc( int i )
{
  iHearInc = i;
  return ( true );
}

/*! Get method for the 'iHearInc' member variable.
    \return the minimum hearing capacity of a player, i.e. the number of
    messages a player can hear in iHearDecay simulation cycles */
int ServerSettings::getHearInc( ) const
{
  return ( iHearInc );
}

/*! Set method for the 'iHearDecay' member variable.
    \param i an integer value representing a new decay rate of player hearing
    capacity, i.e. minimum number of cycles for iHearInc messages
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setHearDecay( int i )
{
  iHearDecay = i;
  return ( true );
}

/*! Get method for the 'iHearDecay' member variable.
    \return the decay rate of player hearing capacity, i.e. minimum number of
    cycles for iHearInc messages */
int ServerSettings::getHearDecay( ) const
{
  return ( iHearDecay );
}

/*! Set method for the 'dInertiaMoment' member variable.
    \param d a double value representing a new inertia moment of a player
    (affects actual turn angle depending on speed)
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setInertiaMoment( double d )
{
  dInertiaMoment = d;
```

```
  return ( true );
}

/*! Get method for the 'dInertiaMoment' member variable.
    \return the inertia moment of a player (affects actual turn angle depending
    on speed) */
double ServerSettings::getInertiaMoment( ) const
{
  return ( dInertiaMoment );
}

/*! Set method for the 'iSenseBodyStep' member variable.
    \param i an integer value representing a new length of the interval (in ms)
    between sense_body information messages
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSenseBodyStep( int i )
{
  iSenseBodyStep = i;
  return ( true );
}

/*! Get method for the 'iSenseBodyStep' member variable.
    \return the length of the interval (in ms) between sense_body information
    messages */
int ServerSettings::getSenseBodyStep( ) const
{
  // NOTE: do not take slow down factor into account for send intervals
  // already done by server
  return iSenseBodyStep ; // * iSlowDownFactor );
}

/*! Set method for the 'dCatchableAreaL' member variable.
    \param d a double value representing a new length of the area around the
    goalkeeper in which he can catch the ball
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setCatchableAreaL( double d )
{
  dCatchableAreaL = d;
  return ( true );
}

/*! Get method for the 'dCatchableAreaL' member variable.
    \return the length of the area around the goalkeeper in which he can catch
    the ball */
double ServerSettings::getCatchableAreaL( ) const
{
```

```
  return ( dCatchableAreaL );
}

/*! Set method for the 'dCatchableAreaW' member variable.
    \param d a double value representing a new width of the area around the
    goalkeeper in which he can catch the ball
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setCatchableAreaW( double d )
{
  dCatchableAreaW = d;
  return ( true );
}

/*! Get method for the 'dCatchableAreaW' member variable.
    \return the width of the area around the goalkeeper in which he can catch
    the ball */
double ServerSettings::getCatchableAreaW( ) const
{
  return ( dCatchableAreaW );
}

/*! Set method for the 'dCatchProbability' member variable.
    \param d a double value representing a new probability for a goalkeeper to
    catch the ball
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setCatchProbability( double d )
{
  dCatchProbability = d;
  return ( true );
}

/*! Get method for the 'dCatchProbability' member variable.
    \return the probability for a goalkeeper to catch the ball */
double ServerSettings::getCatchProbability( ) const
{
  return ( dCatchProbability );
}

/*! Set method for the 'iCatchBanCycle' member variable.
    \param i an integer value representing a new number of cycles after a catch
    in which the goalkeeper cannot catch again
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setCatchBanCycle( int i )
{
  iCatchBanCycle = i;
  return ( true );
```

```
}

/*! Get method for the 'iCatchBanCycle' member variable.
    \return the number of cycles after a catch in which the goalkeeper cannot
    catch again */
int ServerSettings::getCatchBanCycle( ) const
{
  return ( iCatchBanCycle );
}

/*! Set method for the 'iGoalieMaxMoves' member variable.
    \param i an integer value representing a new maximum number of 'move'
    actions allowed for a goalkeeper after a catch
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setGoalieMaxMoves( int i )
{
  iGoalieMaxMoves = i;
  return ( true );
}

/*! Get method for the 'iGoalieMaxMoves' member variable.
    \return the maximum number of 'move' actions allowed for a goalkeeper after
    a catch */
int ServerSettings::getGoalieMaxMoves( ) const
{
  return ( iGoalieMaxMoves );
}

/*! Set method for the 'dBallSize' member variable.
    \param d a double value representing a new ball size
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setBallSize( double d )
{
  dBallSize = d;
  // NOTE: also update parameters for which the value depends on this variable
  dMaximalKickDist = ( dKickableMargin + dPlayerSize + dBallSize );

  return ( true );
}

/*! Get method for the 'dBallSize' member variable.
    \return the size (=radius) of the ball */
double ServerSettings::getBallSize( ) const
{
  return ( dBallSize );
}
```

```
/*! Set method for the 'dBallDecay' member variable.
    \param d a double value representing a new ball speed decay per cycle
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setBallDecay( double d )
{
  dBallDecay = d;
  return ( true );
}

/*! Get method for the 'dBallDecay' member variable.
    \return the ball speed decay per cycle */
double ServerSettings::getBallDecay( ) const
{
  return ( dBallDecay );
}

/*! Set method for the 'dBallRand' member variable.

    \param d a double value representing a new random error in the
    ball movement

    \return a boolean indicating whether the update was successful */
bool ServerSettings::setBallRand( double d )
{
  dBallRand = d;
  return ( true );
}

/*! Get method for the 'dBallRand' member variable.
    \return the random error in the ball movement */
double ServerSettings::getBallRand( ) const
{
  return ( dBallRand );
}

/*! Set method for the 'dBallWeight' member variable.
    \param d a double value representing a new weight of the ball (for wind)
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setBallWeight( double d )
{
  dBallWeight = d;
  return ( true );
}

/*! Get method for the 'dBallWeight' member variable.
```

```
    \return the weight of the ball (for wind) */
double ServerSettings::getBallWeight( ) const
{
  return ( dBallWeight );
}

/*! Set method for the 'dBallSpeedMax' member variable.
    \param d a double value representing a new maximum speed of the ball
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setBallSpeedMax( double d )
{
  dBallSpeedMax = d;
  return ( true );
}

/*! Get method for the 'dBallSpeedMax' member variable.
    \return the maximum speed of the ball */
double ServerSettings::getBallSpeedMax( ) const
{
  return ( dBallSpeedMax );
}

/*! Set method for the 'dBallAccelMax' member variable.
    \param d a double value representing a new maximum acceleration of the ball
    per cycle
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setBallAccelMax( double d )
{
  dBallAccelMax = d;
  return ( true );
}

/*! Get method for the 'dBallAccelMax' member variable.
    \return the maximum acceleration of the ball per cycle */
double ServerSettings::getBallAccelMax( ) const
{
  return ( dBallAccelMax );
}

/*! Set method for the 'dWindForce' member variable.
    \param d a double value representing a new force of the wind
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setWindForce( double d )
{
  dWindForce = d;
  return ( true );
```

```
}

/*! Get method for the 'dWindForce' member variable.
    \return the force of the wind */
double ServerSettings::getWindForce( ) const
{
  return ( dWindForce );
}

/*! Set method for the 'dWindDir' member variable.
    \param d a double value representing a new direction of the wind
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setWindDir( double d )
{
  dWindDir = d;
  return ( true );
}

/*! Get method for the 'dWindDir' member variable.
    \return the direction of the wind */
double ServerSettings::getWindDir( ) const
{
  return ( dWindDir );
}

/*! Set method for the 'dWindRand' member variable.
    \param d a double value representing a new random error in wind direction
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setWindRand( double d )
{
  dWindRand = d;
  return ( true );
}

/*! Get method for the 'dWindRand' member variable.
    \return the random error in wind direction */
double ServerSettings::getWindRand( ) const
{
  return ( dWindRand );
}

/*! Set method for the 'bWindRandom' member variable.
    \param b a boolean indicating whether wind force and direction are random
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setWindRandom( bool b )
{
```

```
  bWindRandom = b;
  return ( true );
}

/*! Get method for the 'bWindRandom' member variable.
    \return boolean indicating whether wind force and direction are random */
bool ServerSettings::getWindRandom( ) const
{
  return bWindRandom;
}

/*! Set method for the 'dKickableMargin' member variable.
    \param d a double value representing a new margin around a player in which
    the ball is kickable (kickable area thus equals kickable_margin + ball_size
    + player_size)
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setKickableMargin( double d )
{
  dKickableMargin = d;
  // NOTE: also update parameters for which the value depends on this variable
  dMaximalKickDist = ( dKickableMargin + dPlayerSize + dBallSize );

  return ( true );
}

/*! Get method for the 'dKickableMargin' member variable.
    \return the margin around a player in which the ball is kickable (kickable
    area thus equals kickable_margin + ball_size + player_size) */
double ServerSettings::getKickableMargin( ) const
{
  return ( dKickableMargin );
}

/*! Set method for the 'dCkickMargin' member variable.
    \param d a double value representing a new corner kick margin, i.e. a new
    minimum distance to the ball for offending players when a corner kick is
    taken
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setCkickMargin( double d )
{
  dCkickMargin = d;
  return ( true );
}

/*! Get method for the 'dCkickMargin' member variable.
    \return the corner kick margin, i.e. the minimum distance to the ball for
```

```
    offending players when a corner kick is taken */
double ServerSettings::getCkickMargin( ) const
{
  return ( dCkickMargin );
}

/*! Set method for the 'dDashPowerRate' member variable.
    \param d a double value representing a new rate by which the 'Power'
    argument in a 'dash' command is multiplied (thus determining the amount of
    displacement of the player as a result of the 'dash')
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setDashPowerRate( double d )
{
  dDashPowerRate = d;
  return ( true );
}

/*! Get method for the 'dDashPowerRate' member variable.
    \return the rate by which the 'Power' argument in a 'dash' command is
    multiplied (thus determining the amount of displacement of the player as a
    result of the 'dash') */
double ServerSettings::getDashPowerRate( ) const
{
  return ( dDashPowerRate );
}

/*! Set method for the 'dKickPowerRate' member variable.
    \param d a double value representing a new rate by which the 'Power'
    argument in a 'kick' command is multiplied (thus determining the amount of
    displacement of the ball as a result of the 'kick')
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setKickPowerRate( double d )
{
  dKickPowerRate = d;
  return ( true );
}

/*! Get method for the 'dKickPowerRate' member variable.
    \return the rate by which the 'Power' argument in a 'kick' command is
    multiplied (thus determining the amount of displacement of the ball as a
    result of the 'kick') */
double ServerSettings::getKickPowerRate( ) const
{
  return ( dKickPowerRate );
}
```

```
/*! Set method for the 'dKickRand' member variable.
    \param d a double value representing a new random error in kick direction
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setKickRand( double d )
{
  dKickRand = d;
  return ( true );
}

/*! Get method for the 'dKickRand' member variable.
    \return the random error in kick direction */
double ServerSettings::getKickRand( ) const
{
  return ( dKickRand );
}

/*! Set method for the 'dVisibleAngle' member variable.
    \param d a double value representing a new angle of the view cone of a
    player in the standard view mode
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setVisibleAngle( double d )
{
  dVisibleAngle = d;
  return ( true );
}

/*! Get method for the 'dVisibleAngle' member variable.
    \return the angle of the view cone of a player in the standard view mode */
double ServerSettings::getVisibleAngle( ) const
{
  return ( dVisibleAngle );
}

/*! Set method for the 'dAudioCutDist' member variable.
    \param d a double value representing a new maximum distance over which a
    spoken message can be heard
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setAudioCutDist( double d )
{
  dAudioCutDist = d;
  return ( true );
}

/*! Get method for the 'dAudioCutDist' member variable.
    \return the maximum distance over which a spoken message can be heard */
double ServerSettings::getAudioCutDist( ) const
```

```
{
  return ( dAudioCutDist );
}

/*! Set method for the 'dQuantizeStep' member variable.
    \param d a double value representing a new quantization step for the
    distance of moving objects
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setQuantizeStep( double d )
{
  dQuantizeStep = d;
  return ( true );
}

/*! Get method for the 'dQuantizeStep' member variable.
    \return the quantization step for the distance of moving objects */
double ServerSettings::getQuantizeStep( ) const
{
  return ( dQuantizeStep );
}

/*! Set method for the 'dQuantizeStepL' member variable.
    \param d a double value representing a new quantization step for the
    distance of landmarks
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setQuantizeStepL( double d )
{
  dQuantizeStepL = d;
  return ( true );
}

/*! Get method for the 'dQuantizeStepL' member variable.
    \return the quantization step for the distance of landmarks */
double ServerSettings::getQuantizeStepL( ) const
{
  return ( dQuantizeStepL );
}

/*! Set method for the 'iMaxPower' member variable.
    \param i an integer value representing a new maximum power for a dash/kick
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMaxPower( int i )
{
  iMaxPower = i;
  return ( true );
}
```

```
/*! Get method for the 'iMaxPower' member variable.
    \return the maximum power for a dash/kick */
int ServerSettings::getMaxPower( ) const
{
  return ( iMaxPower );
}

/*! Set method for the 'iMinPower' member variable.
    \param i an integer value representing a new minimum power for a dash/kick
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMinPower( int i )
{
  iMinPower = i;
  return ( true );
}

/*! Get method for the 'iMinPower' member variable.
    \return the minimum power for a dash/kick */
int ServerSettings::getMinPower( ) const
{
  return ( iMinPower );
}

/*! Set method for the 'iMaxMoment' member variable.
    \param i an integer value representing a new maximum angle for a turn/kick
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMaxMoment( int i )
{
  iMaxMoment = i;
  return ( true );
}

/*! Get method for the 'iMaxMoment' member variable.
    \return the maximum angle for a turn/kick */
int ServerSettings::getMaxMoment( ) const
{
  return ( iMaxMoment );
}

/*! Set method for the 'iMinMoment' member variable.
    \param i an integer value representing a new minimum angle for a turn/kick
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMinMoment( int i )
{
  iMinMoment = i;
```

```
  return ( true );
}

/*! Get method for the 'iMinMoment' member variable.
    \return the minimum angle for a turn/kick */
int ServerSettings::getMinMoment( ) const
{
  return ( iMinMoment );
}

/*! Set method for the 'iMaxNeckMoment' member variable.
    \param i an integer value representing a new maximum angle for a turnneck
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMaxNeckMoment( int i )
{
  iMaxNeckMoment = i;
  return ( true );
}

/*! Get method for the 'iMaxNeckMoment' member variable.
    \return the maximum angle for a turnneck */
int ServerSettings::getMaxNeckMoment( ) const
{
  return ( iMaxNeckMoment );
}

/*! Set method for the 'iMinNeckMoment' member variable.
    \param i an integer value representing a new minimum angle for a turnneck
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMinNeckMoment( int i )
{
  iMinNeckMoment = i;
  return ( true );
}

/*! Get method for the 'iMinNeckMoment' member variable.
    \return the minimum angle for a turnneck */
int ServerSettings::getMinNeckMoment( ) const
{
  return ( iMinNeckMoment );
}

/*! Set method for the 'iMaxNeckAng' member variable.

    \param i an integer value representing a new maximum neck angle
    rel. to body
```

```
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMaxNeckAng( int i )
{
  iMaxNeckAng = i;
  return ( true );
}

/*! Get method for the 'iMaxNeckAng' member variable.
    \return the maximum neck angle relative to body */
int ServerSettings::getMaxNeckAng( ) const
{
  return ( iMaxNeckAng );
}

/*! Set method for the 'iMinNeckAng' member variable.

    \param i an integer value representing a new minimum neck angle
    rel. to body

    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMinNeckAng( int i )
{
  iMinNeckAng = i;
  return ( true );
}

/*! Get method for the 'iMinNeckAng' member variable.
    \return the minimum neck angle relative to body */
int ServerSettings::getMinNeckAng( ) const
{
  return ( iMinNeckAng );
}

/*! Set method for the 'iPort' member variable.
    \param i an integer value representing a new port number for a player
    connection
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPort( int i )
{
  iPort = i;
  return ( true );
}

/*! Get method for the 'iPort' member variable.
    \return the port number for a player connection */
```

```
int ServerSettings::getPort( ) const
{
  return ( iPort );
}

/*! Set method for the 'iCoachPort' member variable.
    \param i an integer value representing a new port number for a coach
    connection
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setCoachPort( int i )
{
  iCoachPort = i;
  return ( true );
}

/*! Get method for the 'iCoachPort' member variable.
    \return the port number for a coach connection */
int ServerSettings::getCoachPort( ) const
{
  return ( iCoachPort );
}

/*! Set method for the 'iOlCoachPort' member variable.
    \param i an integer value representing a new port number for an online
           coach connection
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setOlCoachPort( int i )
{
  iOlCoachPort = i;
  return ( true );
}

/*! Get method for the 'iOlCoachPort' member variable.
    \return the port number for an online coach connection */
int ServerSettings::getOlCoachPort( ) const
{
  return ( iOlCoachPort );
}

/*! Set method for the 'iSayCoachCntMax' member variable.
    \param i an integer value representing a new maximum number of coach
    messages possible
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSayCoachCntMax( int i )
{
  iSayCoachCntMax = i;
```

```
  return ( true );
}

/*! Get method for the 'iSayCoachCntMax' member variable.
    \return the maximum number of coach messages possible */
int ServerSettings::getSayCoachCntMax( ) const
{
  return ( iSayCoachCntMax );
}

/*! Set method for the 'iSayCoachMsgSize' member variable.
    \param i an integer value representing a new maximum size of coach messages
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSayCoachMsgSize( int i )
{
  iSayCoachMsgSize = i;
  return ( true );
}

/*! Get method for the 'iSayCoachMsgSize' member variable.
    \return the maximum size of coach messages */
int ServerSettings::getSayCoachMsgSize( ) const
{
  return ( iSayCoachMsgSize );
}

/*! Set method for the 'iClangWinSize' member variable.
    \param i an integer value representing a new time window which controls how
    many coach messages can be sent
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setClangWinSize( int i )
{
  iClangWinSize = i;
  return ( true );
}

/*! Get method for the 'iClangWinSize' member variable.
    \return time window which controls how many coach messages can be sent */
int ServerSettings::getClangWinSize( ) const
{
  return ( iClangWinSize );
}

/*! Set method for the 'iClangDefineWin' member variable.
    \param i an integer value representing a new number of define messages by
    the coach per time window
```

```
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setClangDefineWin( int i )
{
  iClangDefineWin = i;
  return ( true );
}

/*! Get method for the 'iClangDefineWin' member variable.
    \return the number of define messages by the coach per time window */
int ServerSettings::getClangDefineWin( ) const
{
  return ( iClangDefineWin );
}

/*! Set method for the 'iClangMetaWin' member variable.
    \param i an integer value representing a new number of meta messages by the
    coach per time window
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setClangMetaWin( int i )
{
  iClangMetaWin = i;
  return ( true );
}

/*! Get method for the 'iClangMetaWin' member variable.
    \return the number of meta messages by the coach per time window */
int ServerSettings::getClangMetaWin( ) const
{
  return ( iClangMetaWin );
}

/*! Set method for the 'iClangAdviceWin' member variable.
    \param i an integer value representing a new number of advice messages by
    the coach per time window
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setClangAdviceWin( int i )
{
  iClangAdviceWin = i;
  return ( true );
}

/*! Get method for the 'iClangAdviceWin' member variable.
    \return the number of advice messages by the coach per time window */
int ServerSettings::getClangAdviceWin( ) const
{
  return ( iClangAdviceWin );
```

```
}

/*! Set method for the 'iClangInfoWin' member variable.
    \param i an integer value representing a new number of info messages by the
    coach per time window
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setClangInfoWin( int i )
{
  iClangInfoWin = i;
  return ( true );
}

/*! Get method for the 'iClangInfoWin' member variable.
    \return the number of info messages by the coach per time window */
int ServerSettings::getClangInfoWin( ) const
{
  return ( iClangInfoWin );
}

/*! Set method for the 'iClangMessDelay' member variable.
    \param i an integer value representing a new delay of coach messages, i.e.
    a new number of cycles between the send to a player and the receival of the
    message
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setClangMessDelay( int i )
{
  iClangMessDelay = i;
  return ( true );
}

/*! Get method for the 'iClangMessDelay' member variable.
    \return the delay of coach messages, i.e. the number of cycles between the
    send to a player and the receival of the message */
int ServerSettings::getClangMessDelay( ) const
{
  return ( iClangMessDelay );
}

/*! Set method for the 'iClangMessPerCycle' member variable.
    \param i an integer value representing a new number of coach messages per
    cycle
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setClangMessPerCycle( int i )
{
  iClangMessPerCycle = i;
  return ( true );
```

```
}

/*! Get method for the 'iClangMessPerCycle' member variable.
    \return the number of coach messages per cycle */
int ServerSettings::getClangMessPerCycle( ) const
{
  return ( iClangMessPerCycle );
}

/*! Set method for the 'iSendViStep' member variable.
    \param i an integer value representing a new interval of the coach's look,
    i.e. a new length of the interval (in ms) between visual messages to the
    coach
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSendViStep( int i )
{
  iSendViStep = i;
  return ( true );
}

/*! Get method for the 'iSendViStep' member variable.
    \return the interval of the coach's look, i.e. the length of the interval
    (in ms) between visual messages to the coach */
int ServerSettings::getSendViStep( ) const
{
  // NOTE: do not take slow down factor into account for send intervals
  // already done by server
  return iSendViStep ; // * iSlowDownFactor );
}

/*! Set method for the 'iSimulatorStep' member variable.
    \param i an integer value representing a new length (in ms) of a simulator
    cycle
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSimulatorStep( int i )
{
  iSimulatorStep = i;
  return ( true );
}

/*! Get method for the 'iSimulatorStep' member variable.
    \return the length (in ms) of a simulator cycle */
int ServerSettings::getSimulatorStep( ) const
{
  // NOTE: do not take slow down factor into account for send intervals ->
  // already done in supplied server values
```

```
    return iSimulatorStep ; //  * iSlowDownFactor ;
}

/*! Set method for the 'iSendStep' member variable.
    \param i an integer value representing a new length of the interval (in ms)
    between visual messages to a player in the standard view mode
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSendStep( int i )
{
  iSendStep = i;
  return ( true );
}

/*! Get method for the 'iSendStep' member variable.
    \return the length of the interval (in ms) between visual messages to a
    player in the standard view mode */
int ServerSettings::getSendStep( ) const
{
  // NOTE: do not take slow down factor into account for send intervals,
  // already done by server
  return  iSendStep ; // * iSlowDownFactor );
}

/*! Set method for the 'iRecvStep' member variable.
    \param i an integer value representing a new length of the interval (in ms)
    for accepting commands from a player
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setRecvStep( int i )
{
  iRecvStep = i;
  return ( true );
}

/*! Get method for the 'iRecvStep' member variable.
    \return the length of the interval (in ms) for accepting commands from a
    player */
int ServerSettings::getRecvStep( ) const
{
  return ( iRecvStep );
}

/*! Set method for the 'iHalfTime' member variable.

    \param i an integer value representing a new length (in seconds)
    of a single game half
```

```
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setHalfTime( int i )
{
  iHalfTime = i;
  return ( true );
}

/*! Get method for the 'iHalfTime' member variable.
    \return the length (in seconds) of a single game half */
int ServerSettings::getHalfTime( ) const
{
  return ( iHalfTime );
}

/*! Set method for the 'iDropBallTime' member variable.
    \param i an integer value representing a new number of cycles to wait until
    dropping the ball automatically for free kicks, corner kicks, etc.
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setDropBallTime( int i )
{
  iDropBallTime = i;
  return ( true );
}

/*! Get method for the 'iDropBallTime' member variable.
    \return the number of cycles to wait until dropping the ball automatically
    for free kicks, corner kicks, etc. */
int ServerSettings::getDropBallTime( ) const
{
  return ( iDropBallTime );
}

/*! Set method for the 'iSayMsgSize' member variable.
    \param i an integer value representing a new maximum length (in bytes) of a
    spoken message
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSayMsgSize( int i )
{
  iSayMsgSize = i;
  return ( true );
}

/*! Get method for the 'iSayMsgSize' member variable.
    \return the maximum length (in bytes) of a spoken message */
int ServerSettings::getSayMsgSize( ) const
{
```

```
  return ( iSayMsgSize );
}

/*! Set method for the 'bUseOffside' member variable.
    \param b a boolean value indicating whether the offside rule should be
    applied or not
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setUseOffside( bool b )
{
  bUseOffside = b;
  return ( true );
}

/*! Get method for the 'bUseOffside' member variable.

    \return a boolean flag indicating whether the offside rule should
    be applied or not */
bool ServerSettings::getUseOffside( ) const
{
  return ( bUseOffside );
}

/*! Set method for the 'dOffsideActiveAreaSize' member variable.
    \param d a double value representing a new offside active area size, i.e.
    radius of circle around the ball in which player can be offside
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setOffsideActiveAreaSize( double d )
{
  dOffsideActiveAreaSize = d;
  return ( true );
}

/*! Get method for the 'dOffsideActiveAreaSize' member variable.
    \return the offside active area size, i.e. radius of circle around the ball
    in which player can be offside */
double ServerSettings::getOffsideActiveAreaSize( ) const
{
  return ( dOffsideActiveAreaSize );
}

/*! Set method for the 'bForbidKickOffOffside' member variable.
    \param b a boolean value indicating whether a kick from offside position is
    allowed
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setForbidKickOffOffside( bool b )
{
```

```
  bForbidKickOffOffside = b;
  return ( true );
}

/*! Get method for the 'bForbidKickOffOffside' member variable.
    \return a boolean flag indicating whether a kick from offside position is
    allowed */
bool ServerSettings::getForbidKickOffOffside( ) const
{
  return ( bForbidKickOffOffside );
}

/*! Set method for the 'dOffsideKickMargin' member variable.
    \param d a double value representing a new offside kick margin, i.e. a new
    minimum distance to the ball for offending players when a free kick for
    offside is taken
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setOffsideKickMargin( double d )
{
  dOffsideKickMargin = d;
  return ( true );
}

/*! Get method for the 'dOffsideKickMargin' member variable.
    \return the offside kick margin, i.e. the minimum distance to the ball for
    offending players when a free kick for offside is taken */
double ServerSettings::getOffsideKickMargin( ) const
{
  return ( dOffsideKickMargin );
}

/*! Set method for the 'bVerbose' member variable.
    \param b a boolean value indicating whether the verbose mode is active or
    not; in verbose mode the server sends extra error-information
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setVerbose( bool b )
{
  bVerbose = b;
  return ( true );
}

/*! Get method for the 'bVerbose' member variable.
    \return a boolean flag indicating whether the verbose mode is active or
    not; in verbose mode the server sends extra error-information */
bool ServerSettings::getVerbose( ) const
{
```

```
  return ( bVerbose );
}

/*! Set method for the 'iRecordVersion' member variable.
    \param i an integer value representing a new type of log record
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setRecordVersion( int i )
{
  iRecordVersion = i;
  return ( true );
}

/*! Get method for the 'iRecordVersion' member variable.
    \return the type of log record */
int ServerSettings::getRecordVersion( ) const
{
  return ( iRecordVersion );
}

/*! Set method for the 'bRecordLog' member variable.
    \param b a boolean value indicating whether a log record for a game should
    be created
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setRecordLog( bool b )
{
  bRecordLog = b;
  return ( true );
}

/*! Get method for the 'bRecordLog' member variable.
    \return a boolean flag indicating whether a log record for a game should
    be created */
bool ServerSettings::getRecordLog( ) const
{
  return ( bRecordLog );
}

/*! Set method for the 'bSendLog' member variable.
    \param b a boolean value indicating whether a send client command log for a
    game should be created
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSendLog( bool b )
{
  bSendLog = b;
  return ( true );
}
```

```
/*! Get method for the 'bSendLog' member variable.
    \return a boolean flag indicating whether a send client command log for a
    game should be created */
bool ServerSettings::getSendLog( ) const
{
  return ( bSendLog );
}

/*! Set method for the 'bLogTimes' member variable.
    \param b a boolean value indicating whether ms should be written between
    cycles in log file
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setLogTimes( bool b )
{
  bLogTimes = b;
  return ( true );
}

/*! Get method for the 'bLogTimes' member variable.
    \return a boolean flag indicating whether ms should be written between
    cycles in log file */
bool ServerSettings::getLogTimes( ) const
{
  return ( bLogTimes );
}

/*! Set method for the 'strLogFile' member variable.
    \param str a string representing a new server log to store all actions
    received
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setLogFile( char *str )
{
  strcpy( strLogFile, str );
  return ( true );
}

/*! Get method for the 'strLogFile' member variable.
    \return name of the server log in which all actions received are stored */
char* ServerSettings::getLogFile( )
{
  return ( strLogFile );
}

/*! Set method for the 'bSynchMode' member variable.
    \param b a boolean representing whether synchronization mode is on
```

```
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSynchMode( bool b )
{
  bSynchMode = b;
  return ( true );
}

/*! Get method for the 'bSynchMode' member variable.
    \return boolean indicating whether synchronization mode is used */
bool ServerSettings::getSynchMode( ) const
{
  return ( bSynchMode );
}

/*! Get method for the 'bFullStateR' member variable.
    \return boolean representing whether full state is on for the right team */
bool ServerSettings::getFullStateRight( ) const
{
  return bFullStateR;
}

/*! Set method for the 'bFullStateR' member variable.
    \param b a boolean representing whether full state is on for the right team
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setFullStateRight( bool b )
{
  bFullStateR = b;
  return true;
}

/*! Set method for the 'bFullStateL' member variable.
    \param b a boolean representing whether full state is on for the left team
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setFullStateLeft( bool b )
{
  bFullStateL = b;
  return true;
}

/*! Get method for the 'bFullStateL' member variable.
    \return boolean representing whether full state is on for the left team */
bool ServerSettings::getFullStateLeft( ) const
{
  return bFullStateL;
}
```

```
/*! Set method for the 'iPlayerTypes' member variable.
    \param i an integer value representing a new number of player types
    including the default player type
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerTypes( int i )
{
  iPlayerTypes = i;
  return ( true );
}

/*! Get method for the 'iPlayerTypes' member variable.
    \return the number of player types including the default player type */
int ServerSettings::getPlayerTypes( ) const
{
  return ( iPlayerTypes );
}

/*! Set method for the 'iSubsMax' member variable.

    \param i an integer value representing a new maximum number of
    substitutions allowed during a game (this value also indicates the
    maximum number of players allowed for each type)

    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSubsMax( int i )
{
  iSubsMax = i;
  return ( true );
}

/*! Get method for the 'iSubsMax' member variable.

    \return the maximum number of substitutions allowed during a game
    (this value also indicates the maximum number of players allowed
    for each type) */
int ServerSettings::getSubsMax( ) const
{
  return ( iSubsMax );
}

/*! Set method for the 'dPlayerSpeedMaxDeltaMin' member variable.
    \param d a double value representing a new minimum delta for adjusting
    player_speed_max
```

```
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerSpeedMaxDeltaMin( double d )
{
  dPlayerSpeedMaxDeltaMin = d;
  return ( true );
}

/*! Get method for the 'dPlayerSpeedMaxDeltaMin' member variable.
    \return the minimum delta for adjusting player_speed_max */
double ServerSettings::getPlayerSpeedMaxDeltaMin( ) const
{
  return ( dPlayerSpeedMaxDeltaMin );
}

/*! Set method for the 'dPlayerSpeedMaxDeltaMax' member variable.
    \param d a double value representing a new maximum delta for adjusting
    player_speed_max
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerSpeedMaxDeltaMax( double d )
{
  dPlayerSpeedMaxDeltaMax = d;
  return ( true );
}

/*! Get method for the 'dPlayerSpeedMaxDeltaMax' member variable.
    \return the maximum delta for adjusting player_speed_max */
double ServerSettings::getPlayerSpeedMaxDeltaMax( ) const
{
  return ( dPlayerSpeedMaxDeltaMax );
}

/*! Set method for the 'dStaminaIncMaxDeltaFactor' member variable.
    \param d a double value representing a new amount by which delta is
    multiplied for stamina_inc_max
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setStaminaIncMaxDeltaFactor( double d )
{
  dStaminaIncMaxDeltaFactor = d;
  return ( true );
}

/*! Get method for the 'dStaminaIncMaxDeltaFactor' member variable.
    \return the amount by which delta is multiplied for stamina_inc_max */
double ServerSettings::getStaminaIncMaxDeltaFactor( ) const
{
  return ( dStaminaIncMaxDeltaFactor );
```

```
}

/*! Set method for the 'dPlayerDecayDeltaMin' member variable.
    \param d a double value representing a new minimum delta for adjusting
    player_decay
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerDecayDeltaMin( double d )
{
  dPlayerDecayDeltaMin = d;
  return ( true );
}

/*! Get method for the 'dPlayerDecayDeltaMin' member variable.
    \return the minimum delta for adjusting player_decay */
double ServerSettings::getPlayerDecayDeltaMin( ) const
{
  return ( dPlayerDecayDeltaMin );
}

/*! Set method for the 'dPlayerDecayDeltaMax' member variable.
    \param d a double value representing a new maximum delta for adjusting
    player_decay
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerDecayDeltaMax( double d )
{
  dPlayerDecayDeltaMax = d;
  return ( true );
}

/*! Get method for the 'dPlayerDecayDeltaMax' member variable.
    \return the maximum delta for adjusting player_decay */
double ServerSettings::getPlayerDecayDeltaMax( ) const
{
  return ( dPlayerDecayDeltaMax );
}

/*! Set method for the 'dInertiaMomentDeltaFactor' member variable.
    \param d a double value representing a new amount by which delta is
    multiplied for inertia_moment
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setInertiaMomentDeltaFactor( double d )
{
  dInertiaMomentDeltaFactor = d;
  return ( true );
}
```

```
/*! Get method for the 'dInertiaMomentDeltaFactor' member variable.
    \return the amount by which delta is multiplied for inertia_moment */
double ServerSettings::getInertiaMomentDeltaFactor( ) const
{
  return ( dInertiaMomentDeltaFactor );
}

/*! Set method for the 'dDashPowerRateDeltaMin' member variable.
    \param d a double value representing a new minimum delta for adjusting
    dash_power_rate
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setDashPowerRateDeltaMin( double d )
{
  dDashPowerRateDeltaMin = d;
  return ( true );
}

/*! Get method for the 'dDashPowerRateDeltaMin' member variable.
    \return the minimum delta for adjusting dash_power_rate */
double ServerSettings::getDashPowerRateDeltaMin( ) const
{
  return ( dDashPowerRateDeltaMin );
}

/*! Set method for the 'dDashPowerRateDeltaMax' member variable.
    \param d a double value representing a new maximum delta for adjusting
    dash_power_rate
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setDashPowerRateDeltaMax( double d )
{
  dDashPowerRateDeltaMax = d;
  return ( true );
}

/*! Get method for the 'dDashPowerRateDeltaMax' member variable.
    \return the maximum delta for adjusting dash_power_rate */
double ServerSettings::getDashPowerRateDeltaMax( ) const
{
  return ( dDashPowerRateDeltaMax );
}

/*! Set method for the 'dPlayerSizeDeltaFactor' member variable.
    \param d a double value representing a new amount by which delta is
    multiplied for player_size
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setPlayerSizeDeltaFactor( double d )
```

```
{
 dPlayerSizeDeltaFactor = d;
 return ( true );
}

/*! Get method for the 'dPlayerSizeDeltaFactor' member variable.
    \return the amount by which delta is multiplied for player_size */
double ServerSettings::getPlayerSizeDeltaFactor( ) const
{
 return ( dPlayerSizeDeltaFactor );
}

/*! Set method for the 'dKickableMarginDeltaMin' member variable.
    \param d a double value representing a new minimum delta for adjusting
    kickable_margin
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setKickableMarginDeltaMin( double d )
{
 dKickableMarginDeltaMin = d;
 return ( true );
}

/*! Get method for the 'dKickableMarginDeltaMin' member variable.
    \return the minimum delta for adjusting kickable_margin */
double ServerSettings::getKickableMarginDeltaMin( ) const
{
 return ( dKickableMarginDeltaMin );
}

/*! Set method for the 'dKickableMarginDeltaMax' member variable.
    \param d a double value representing a new maximum delta for adjusting
    kickable_margin
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setKickableMarginDeltaMax( double d )
{
 dKickableMarginDeltaMax = d;
 return ( true );
}

/*! Get method for the 'dKickableMarginDeltaMax' member variable.
    \return the maximum delta for adjusting kickable_margin */
double ServerSettings::getKickableMarginDeltaMax( ) const
{
 return ( dKickableMarginDeltaMax );
}
```

```
/*! Set method for the 'dKickRandDeltaFactor' member variable.
    \param d a double value representing a new amount by which delta is
    multiplied for kick_rand
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setKickRandDeltaFactor( double d )
{
  dKickRandDeltaFactor = d;
  return ( true );
}

/*! Get method for the 'dKickRandDeltaFactor' member variable.
    \return the amount by which delta is multiplied for kick_rand */
double ServerSettings::getKickRandDeltaFactor( ) const
{
  return ( dKickRandDeltaFactor );
}

/*! Set method for the 'dExtraStaminaDeltaMin' member variable.
    \param d a double value representing a new minimum delta for adjusting
    extra_stamina
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setExtraStaminaDeltaMin( double d )
{
  dExtraStaminaDeltaMin = d;
  return ( true );
}

/*! Get method for the 'dExtraStaminaDeltaMin' member variable.
    \return the minimum delta for adjusting extra_stamina */
double ServerSettings::getExtraStaminaDeltaMin( ) const
{
  return ( dExtraStaminaDeltaMin );
}

/*! Set method for the 'dExtraStaminaDeltaMax' member variable.
    \param d a double value representing a new maximum delta for adjusting
    extra_stamina
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setExtraStaminaDeltaMax( double d )
{
  dExtraStaminaDeltaMax = d;
  return ( true );
}

/*! Get method for the 'dExtraStaminaDeltaMax' member variable.
    \return the maximum delta for adjusting extra_stamina */
```

```
double ServerSettings::getExtraStaminaDeltaMax( ) const
{
  return ( dExtraStaminaDeltaMax );
}

/*! Set method for the 'dEffortMaxDeltaFactor' member variable.
    \param d a double value representing a new amount by which delta is
    multiplied for effort_max
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortMaxDeltaFactor( double d )
{
  dEffortMaxDeltaFactor = d;
  return ( true );
}

/*! Get method for the 'dEffortMaxDeltaFactor' member variable.
    \return the amount by which delta is multiplied for effort_max */
double ServerSettings::getEffortMaxDeltaFactor( ) const
{
  return ( dEffortMaxDeltaFactor );
}

/*! Set method for the 'dEffortMinDeltaFactor' member variable.
    \param d a double value representing a new amount by which delta is
    multiplied for effort_min
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortMinDeltaFactor( double d )
{
  dEffortMinDeltaFactor = d;
  return ( true );
}

/*! Get method for the 'dEffortMinDeltaFactor' member variable.
    \return the amount by which delta is multiplied for effort_min */
double ServerSettings::getEffortMinDeltaFactor( ) const
{
  return ( dEffortMinDeltaFactor );
}

/*! Set method for the 'dDashPowerRateDeltaMin' member variable.
    \param d a double value representing a new minimum delta for adjusting
    dash_power_rate
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setNewDashPowerRateDeltaMin( double d )
{
  dNewDashPowerRateDeltaMin = d;
```

```
  return ( true );
}

/*! Get method for the 'dNewDashPowerRateDeltaMin' member variable.
    \return the minimum delta for adjusting dash_power_rate */
double ServerSettings::getNewDashPowerRateDeltaMin( ) const
{
  return ( dNewDashPowerRateDeltaMin );
}

/*! Set method for the 'dNewDashPowerRateDeltaMax' member variable.
    \param d a double value representing a new maximum delta for adjusting
    dash_power_rate
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setNewDashPowerRateDeltaMax( double d )
{
  dNewDashPowerRateDeltaMax = d;
  return ( true );
}

/*! Get method for the 'dNewDashPowerRateDeltaMax' member variable.
    \return the maximum delta for adjusting dash_power_rate */
double ServerSettings::getNewDashPowerRateDeltaMax( ) const
{
  return ( dNewDashPowerRateDeltaMax );
}

/*! Set method for the 'dNewStaminaIncMaxDeltaFactor' member variable.
    \param d a double value representing a new amount by which delta is
    multiplied for stamina_inc_max
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setNewStaminaIncMaxDeltaFactor( double d )
{
  dNewStaminaIncMaxDeltaFactor = d;
  return ( true );
}

/*! Get method for the 'dNewStaminaIncMaxDeltaFactor' member variable.
    \return the amount by which delta is multiplied for stamina_inc_max */
double ServerSettings::getNewStaminaIncMaxDeltaFactor( ) const
{
  return ( dNewStaminaIncMaxDeltaFactor );
}

/*! Set method for the 'dPenDistX' variable that indicates where the ball will
    be positioned in front of the goalline in case of a penalty kick. */
```

```
bool ServerSettings::setPenDistX( double d )
{
  dPenDistX = d;
  return true;
}

/*! Get method for the 'dPenDistX' variable that indicates where the ball will
    be positioned in front of the goalline in case of a penalty kick. */
double ServerSettings::getPenDistX( ) const
{
  return dPenDistX;
}

/*! Set method for the 'dPenMaxGoalieDistX' variable that indicates
    how far the goalie is allowed to move outside its goal in case of a
    penalty kick. */
bool ServerSettings::setPenMaxGoalieDistX( double d )
{
  dPenMaxGoalieDistX = d;
  return true;
}

/*! Get method for the 'dPenMaxGoalieDistX' variable that indicates
    how far the goalie is allowed to move outside its goal in case of a
    penalty kick. */
double ServerSettings::getPenMaxGoalieDistX( ) const
{
  return dPenMaxGoalieDistX;
}

/*! Set method for the 'bPenAllowMultKicks' variable that indicates whether the
    penalty kicker is allowed to kick the ball multiple times. */
bool ServerSettings::setPenAllowMultKicks( bool b )
{
  bPenAllowMultKicks = b;
  return true;
}

/*! Set method for the 'bPenAllowMultKicks' variable that indicates whether the
    penalty kicker is allowed to kick the ball multiple times. */
bool ServerSettings::getPenAllowMultKicks( ) const
{
  return bPenAllowMultKicks;
}

/*! Set method for the 'dTackleDist' variable that indicates the maximum x
```

```
    distance to the front of the agent in which a tackle can be performed. The
    probabiltiy of succeeding depends on this distance. */
bool ServerSettings::setTackleDist( double d )
{
  dTackleDist = d;
  return true;
}

/*! Get method for the 'dTackleDist' variable that indicates the maximum x
    distance to the front of the agent in which a tackle can be performed. The
    probabiltiy of succeeding depends on this distance. */
double ServerSettings::getTackleDist( ) const
{
  return dTackleDist;
}

/*! Set method for the 'dTackleBackDist' variable that indicates the maximum x
    distance to the back of the agent in which a tackle can be performed. The
    probabiltiy of succeeding depends on this distance. */
bool ServerSettings::setTackleBackDist( double d )
{
  dTackleBackDist = d;
  return true;
}

/*! Get method for the 'dTackleBackDist' variable that indicates the maximum x
    distance to the back of the agent in which a tackle can be performed. The
    probabiltiy of succeeding depends on this distance. */
double ServerSettings::getTackleBackDist( ) const
{
  return dTackleBackDist;
}

/*! Set method for the 'dTackleWidth' variable that indicates the maximum y
    distance to the agent in which a tackle can be performed. The
    probabiltiy of succeeding depends on this distance. */
bool ServerSettings::setTackleWidth( double d )
{
  dTackleWidth = d;
  return true;
}

/*! Get method for the 'dTackleWidth' variable that indicates the maximum y
    distance to the agent in which a tackle can be performed. The
    probabiltiy of succeeding depends on this distance. */
double ServerSettings::getTackleWidth( ) const
```

```
{
  return dTackleWidth;
}

/*! Set method for the 'dTackleExponent' variable that indicates the exponent
    used in order to calculate the probability that the tackle will succeed. */
bool ServerSettings::setTackleExponent( double d )
{
  dTackleExponent = d;
  return true;
}

/*! Get method for the 'dTackleExponent' variable that indicates the exponent
    used in order to calculate the probability that the tackle will succeed. */
double ServerSettings::getTackleExponent( ) const
{
  return dTackleExponent;
}

/*! Set method for the 'dTackleCycles' variable that indicates how many cycles
    a player is immobile after performing a tackle. */
bool ServerSettings::setTackleCycles( int i )
{
  iTackleCycles = i;
  return true;
}

/*! Get method for the 'dTackleCycles' variable that indicates how many cycles
    a player is immobile after performing a tackle. */
int ServerSettings::getTackleCycles( ) const
{
  return iTackleCycles;
}

/*! Set method for the 'dTacklePowerRate' variable that indicates with which
    power the ball is accelerated after a tackle. */
bool ServerSettings::setTacklePowerRate( double d )
{
  dTacklePowerRate = d;
  return true;
}

/*! Get method for the 'dTacklePowerRate' variable that indicates with which
    power the ball is accelerated after a tackle. */
double ServerSettings::getTacklePowerRate( ) const
{
```

```
  return dTacklePowerRate;
}

/*! Set method for the 'dEffortMax' member variable.
    \param d a double value representing a new maximum player effort capacity
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setEffortMax( double d )
{
  dEffortMax = d;
  return ( true );
}

/*! Get method for the 'dEffortMax' member variable.
    \return the maximum player effort capacity */
double ServerSettings::getEffortMax(  ) const
{
  return ( dEffortMax );
}

/*! Set method for the 'iSlowDownFactor' member variable.
    \param i an integer value representing a new factor to slow down the
    simulator and send intervals
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setSlowDownFactor( int i )
{
  iSlowDownFactor = i;
  return ( true );
}

/*! Get method for the 'iSlowDownFactor' member variable.
    \return the factor to slow down the simulator and send intervals */
int ServerSettings::getSlowDownFactor( ) const
{
  return ( iSlowDownFactor );
}

/*! Set method for the 'dVisibleDistance' member variable.

    \param d a double value representing a new distance within which
    objects are always 'visible' (even when not in view cone)

    \return a boolean indicating whether the update was successful */
bool ServerSettings::setVisibleDistance( double d )
{
  dVisibleDistance = d;
  return ( true );
```

```
}

/*! Get method for the 'dVisibleDistance' member variable.
    \return the distance within which objects are always 'visible' (even when
    not in view cone) */
double ServerSettings::getVisibleDistance( ) const
{
  return ( dVisibleDistance );
}

/*! Set method for the 'dExtraStamina' member variable.
    \param d a double value representing a new amount of extra stamina that can
    be added to the maximum stamina of a player. This value depends on the type
    of heterogeneous player.
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setExtraStamina( double d )
{
  dExtraStamina = d;
  return ( true );
}

/*! Get method for the 'dExtraStamina' member variable.
    \return the amount of extra stamina that can be added to the maximum
    stamina of a player. This value depends on the type of heterogeneous
    player. */
double ServerSettings::getExtraStamina( ) const
{
  return ( dExtraStamina );
}

/*! Set method for the 'dMaximalKickDist' member variable.
    \param d a double value representing a new maximum distance from a player
    for which the ball is still kickable
    \return a boolean indicating whether the update was successful */
bool ServerSettings::setMaximalKickDist( double d )
{
  dMaximalKickDist = d;
  return ( true );
}

/*! Get method for the 'dMaximalKickDist' member variable.
    \return the maximum distance from a player for which the ball is still
    kickable */
double ServerSettings::getMaximalKickDist( ) const
{
  return ( dMaximalKickDist );
```

```
}

/*************************************************************************
/
/*******************   CLASS HETEROPLAYERSETTINGS
*****************************/
/*************************************************************************
/

/*! This method prints all the values of this heterogeneous player type to
    the supplied output.
    \param os output stream to which values are printed. */
void HeteroPlayerSettings::show( ostream &os )
{
                                                        os.setf( ios::left );
  os << dPlayerSpeedMax << " ";
  os.width( 5 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
                                                        os << dStaminaIncMax << " "
;
  os.width( 6 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
                                                        os << dPlayerDecay  << " ";
  os.width( 5 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
  os << dInertiaMoment  << " " ;
  os.width( 8 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
                                                        os << dDashPowerRate << " "
;
  os.width( 3 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
                                                        os << dPlayerSize     << " " ;
  os.width( 6 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
                                                        os << dKickableMargin << " "
;
  os.width( 6 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
```

```
                                                        os << dKickRand      << " " ;
  os.width( 5 );
                                                        os.precision( 3 );
                                                        os.setf( ios::left );
                                                        os << dExtraStamina    << " " ;
  os.width( 6 );
                                                        os.precision( 4 );
                                                        os.setf( ios::left );
                                                        os << dEffortMin     << " " ;
  os.width( 6 );
                                                        os.setf( ios::left );
                                                        os.precision( 4 );
                                                        os << dEffortMax      << endl ;
/*
  os <<       dPlayerSpeedMax << " " << dStaminaIncMax << " " << dPlayerDecay
     << " " << dInertiaMoment  << " " << dDashPowerRate << " " << dPlayerSize
     << " " << dKickableMargin << " " << dKickRand      << " " << dExtraStamina
     << " " << dEffortMax      << " " << dEffortMin     << endl;
*/
}


/********************* TESTING PURPOSES
*************************************/
/*

int main( void )
{
  ServerSettings settings;
//  settings.show( cout );
  settings.readValues( "server.conf", ":" );
//  printf( "------------\n" );
  settings.show( cout );
  cout << "player_size: " << settings.getPlayerSize( ) << endl;
}

*/
```

BasicPlayer.h

```
#ifndef _BASICPLAYER_
#define _BASICPLAYER_

#include "ActHandler.h"

extern Logger Log; /*!< This is a reference to Logger to write log info to*/

/*! This class defines the skills that can be used by an agent. No
   functionality is available that chooses when to execute which skill, this
   is done in the Player class. The WorldModel is used to determine the way
   in which the skills are performed. */
class BasicPlayer
{
protected:
  ActHandler      *ACT; /*!< ActHandler to which commands can be sent        */
  WorldModel      *WM;  /*!< WorldModel that contains information of world   */
  ServerSettings  *SS;  /*!< All parameters used by the server               */
  PlayerSettings  *PS;  /*!< All parameters used for the player              */

  /////////////////////////// LOW-LEVEL SKILLS ////////////////////////////////

  SoccerCommand   alignNeckWithBody      (                          );
  SoccerCommand   turnBodyToPoint        ( VecPosition   pos,
                                 int          iPos = 1          );
  SoccerCommand   turnBackToPoint        ( VecPosition   pos,
                                 int          iPos = 1          );
  SoccerCommand   turnNeckToPoint        ( VecPosition   pos,
                                 SoccerCommand com              );
  SoccerCommand   searchBall             (                          );
  SoccerCommand   dashToPoint            ( VecPosition   pos,
                                 int          iCycles = 1       );
  SoccerCommand   freezeBall             (                          );
  SoccerCommand   kickBallCloseToBody    ( AngDeg        ang,

double       dKickRatio = 0.16  );
  SoccerCommand   accelerateBallToVelocity( VecPosition   vel             );
  SoccerCommand   catchBall              (                          );
  SoccerCommand   communicate            ( char         *str             );
  SoccerCommand   teleportToPos          ( VecPosition   pos             );
  SoccerCommand   listenTo               ( ObjectT       obj             );
  SoccerCommand   tackle                 (                          );

  /////////////////////////// INTERMEDIATE SKILLS ////////////////////////////

  SoccerCommand   turnBodyToObject       ( ObjectT       o               );
  SoccerCommand   turnNeckToObject       ( ObjectT       o,
```

```
                    SoccerCommand com               );
SoccerCommand   directTowards       ( VecPosition   posTo,
                    AngDeg      angWhenToTurn,
                    VecPosition   *pos = NULL,
                    VecPosition   *vel = NULL,
                    AngDeg      *angBody  = NULL   );
SoccerCommand   moveToPos           ( VecPosition   posTo,
                    AngDeg      angWhenToTurn,
                    double      dDistDashBack = 0.0,
                    bool        bMoveBack = false,
                    int         iCycles = 1       );
SoccerCommand   collideWithBall     (                           );
SoccerCommand   interceptClose      (                           );
SoccerCommand   interceptCloseGoalie    (                       );
SoccerCommand   kickTo              ( VecPosition   posTarget,
                    double      dEndSpeed         );
SoccerCommand   turnWithBallTo      ( AngDeg        ang,
                    AngDeg      angKickThr,
                    double      dFreezeThr        );
SoccerCommand   moveToPosAlongLine  ( VecPosition   pos,
                    AngDeg      ang,
                    double      dDistThr,
                    int         iSign,
                    AngDeg      angThr,
                    AngDeg      angCorr           );


////////////////////////// HIGH-LEVEL SKILLS ////////////////////////////////

SoccerCommand   intercept           ( bool          isGoalie          );
SoccerCommand   dribble             ( AngDeg        ang,
                    DribbleT    d                 );
SoccerCommand   directPass          ( VecPosition   pos,
                    PassT       passType          );
SoccerCommand   leadingPass         ( ObjectT       o,
                    double      dDist,
                    DirectionT  dir = DIR_NORTH   );
SoccerCommand   throughPass         ( ObjectT       o,
                    VecPosition   posEnd,
                    AngDeg      *angMax = NULL    );
SoccerCommand   outplayOpponent     ( ObjectT       o,
                    VecPosition   pos,
                    VecPosition   *posTo = NULL     );
SoccerCommand   clearBall           ( ClearBallT    type,
                    SideT       s = SIDE_ILLEGAL,
                    AngDeg      *angMax = NULL    );
```

```
  SoccerCommand   mark                ( ObjectT       o,
                                double        dDist,
                                MarkT         mark              );
  SoccerCommand   defendGoalLine          ( double        dDist             );
  SoccerCommand   interceptScoringAttempt (                                 );
  SoccerCommand   holdBall            (                             );

  /////////////////////////// UTILITY METHODS ////////////////////////////////

  VecPosition     getThroughPassShootingPoint( ObjectT       objTeam,
                                VecPosition   posEnd,

AngDeg        *angMax         );
  VecPosition     getInterceptionPointBall(    int          *iCyclesBall,
                                bool          isGoalie        );
  VecPosition     getActiveInterceptionPointBall
                              ( int           *iCyclesBall,
                                bool          isGoalie          );
  VecPosition     getDribblePoint         ( DribbleT      dribble,
                                double        *dDist            );
  VecPosition     getShootPositionOnLine  ( VecPosition   p1,
                                VecPosition   p2,
                                AngDeg        *angLargest = NULL );
  double          getEndSpeedForPass      ( ObjectT       o,
                                VecPosition   posPass           );
  VecPosition     getMarkingPosition      ( ObjectT       o,
                                double        dDist,
                                MarkT         mark              );
} ;

#endif
```

BasicPlayer.cpp

```
#include "BasicPlayer.h"
#include "Parse.h"        // parseFirstInt

/********************** LOW-LEVEL SKILLS
************************************/

/*! This skill enables an agent to align his neck with his body. It returns a
    turn neck command that takes the angle of the agent's body relative to his
    neck as its only argument.
   \return SoccerCommand turn_neck command that aligns neck with body */
SoccerCommand BasicPlayer::alignNeckWithBody( )
{
  return SoccerCommand( CMD_TURNNECK, WM->getAgentBodyAngleRelToNeck( ) );
}

/*! This skill enables an agent to turn his body towards a given point. It
    receives a global position 'pos' on the field and returns a turn command
    that will turn the agent's body towards this point. To this end the agent's
    global position in the next cycle is predicted based on his current
    velocity. This is done to compensate for the fact that the remaining
    velocity will move the agent to another position in the next cycle. The
    global angle between the given position and the predicted position is then
    determined after which the agent's global body direction is subtracted from
    this angle in order to make it relative to the agent's body. Finally,
    the resulting angle is normalized and adjusted to compensate for the
    inertia moment and speed of the agent. If it is impossible to turn towards
    the given position in a single cycle then the agent turns as far as
    possible.
    \param pos position to which body should be turned
    \param iCycles denotes the number of cycles that are used to update the
            the agent position. The resulting position is compared with
            'pos' to determine the desired turning angle.
    \return SoccerCommand turn command to turn body to the desired point */
SoccerCommand BasicPlayer::turnBodyToPoint( VecPosition pos, int iCycles )
{
  VecPosition posGlobal = WM->predictAgentPos(iCycles, 0);
  AngDeg angTurn        = (pos - posGlobal).getDirection();
  angTurn              -= WM->getAgentGlobalBodyAngle();
  angTurn               = VecPosition::normalizeAngle( angTurn );
  angTurn               = WM->getAngleForTurn( angTurn, WM->getAgentSpeed(),
                                WM->getAgentObjectType() );

  return SoccerCommand( CMD_TURN, angTurn );
}

/*! This skill enables an agent to turn his back towards a given point 'pos'.
```

```
    The only difference between this skill and turnBodyToPoint is that the
    angle between the given position and the predicted position of the agent
    in the next cycle is now made relative to the back of the agent by
    subtracting the agent's global back direction. This skill can for example
    be used by the goalkeeper in case he wants to move back to his goal while
    keeping sight of the rest of the field.
    \param pos position to which the agent's back should be turned
    \param iCycles denotes the number of cycles that are used to update the
            the agent position. The resulting position is compared with
            'pos' to determine the desired turning angle.
    \return SoccerCommand command to turn agent's back to the desired point */
SoccerCommand BasicPlayer::turnBackToPoint( VecPosition pos, int iCycles )
{
  VecPosition posGlobal = WM->predictAgentPos(iCycles, 0);
  AngDeg angTurn        = (pos - posGlobal).getDirection();
  angTurn              -= (WM->getAgentGlobalBodyAngle() + 180);
  angTurn               = VecPosition::normalizeAngle( angTurn );
  angTurn               = WM->getAngleForTurn( angTurn, WM->getAgentSpeed(),
                                WM->getAgentObjectType() );

  return SoccerCommand( CMD_TURN, angTurn );
}

/*! This skill enables an agent to turn his neck towards a given point. It
    receives a global position 'pos' on the field as well as a primary action
    command 'soc' that will be executed by the agent at the end of the current
    cycle and returns a turn neck command that will turn the agent's neck
    towards 'pos'. To this end the agent's global position and neck direction
    after executing the cmd command are predicted using methods from the world
    model. The global angle between the given position and the predicted
    position is then determined after which the predicted neck direction is
    subtracted from this angle in order to make it relative to the agent's
    neck. Finally, the resulting angle is normalized and directly passed as an
    argument to the turn neck command since the actual angle with which a
    player turns his neck is by definition equal to this argument.
    If the resulting turn angle causes the absolute angle between
    the agent's neck and body to exceed the maximum value, then the agent turns
    his neck as far as possible. Note that it is necessary to supply the
    selected primary command as an argument to this skill, since a turn neck
    command can be executed in the same cycle as a kick, dash, turn , move
    or catch command.
   \param pos position to which neck should be turned
   \param soc SoccerCommand that is executed in the same cycle
   \return SoccerCommand turn command to turn neck to the desired point */
SoccerCommand BasicPlayer::turnNeckToPoint(VecPosition pos, SoccerCommand soc)
{
```

```
  VecPosition posMe,   velMe;
  AngDeg      angBody, angNeck, angActual;
  Stamina     sta;

  // predict agent information after command 'soc' is performed
  // calculate the desired global angle of the neck
  // calculate the desired angle of the neck relative to the body
  WM->predictAgentStateAfterCommand(soc,&posMe,&velMe,&angBody,&angNeck,&sta);
  AngDeg angDesGlobNeck  = (pos - posMe).getDirection();
  AngDeg angNeckRelToBody= VecPosition::normalizeAngle(angDesGlobNeck-angBody);

  // calculate the current angle of the body relative to the neck
  // check if the desired neck angle relative to the body is possible:
  // if angle is smaller than the minimum or larger than the maximum neck angle
  //  turn neck to the minimum or maximum neck angle + the current neck angle
  // else calculate the desired angle relative to the body
  AngDeg angBodyRelToNeck = VecPosition::normalizeAngle(angBody-angNeck);
  if( angNeckRelToBody < SS->getMinNeckAng() )
    angActual = SS->getMinNeckAng() + angBodyRelToNeck;
  else if( angNeckRelToBody > SS->getMaxNeckAng() )
    angActual = SS->getMaxNeckAng() + angBodyRelToNeck;
  else
    angActual = angNeckRelToBody + angBodyRelToNeck;
  return SoccerCommand( CMD_TURNNECK, angActual );
}

/*! This skill enables an agent to search for the ball when he cannot see it.
    It returns a turn command that causes the agent to turn his body by an
    angle that equals the width of his current view cone (denoted by the
    ViewAngle attribute in the AgentObject class). In this way the agent will
    see an entirely different part of the field after the turn which maximizes
    the chance that he will see the ball in the next cycle. Note that the
    agent turns towards the direction in which the ball was last observed to
    avoid turning back and forth without ever seeing the ball. Furthermore
    the inertia moment of the agent is taken into account to compensate for
    the current speed of the agent.
    \return SoccerCommand that searches for the ball. */
SoccerCommand BasicPlayer::searchBall()
{
  static Time   timeLastSearch;
  static SoccerCommand soc;
  static int    iSign       = 1;
  VecPosition   posBall =WM->getBallPos();
  VecPosition   posAgent=WM->getAgentGlobalPosition();
  AngDeg        angBall =(posBall-WM->getAgentGlobalPosition()).getDirection();
  AngDeg        angBody =WM->getAgentGlobalBodyAngle();
```

```
  if( WM->getCurrentTime().getTime() == timeLastSearch.getTime()  )
    return soc;

  if( WM->getCurrentTime() - timeLastSearch > 3 )
    iSign = ( isAngInInterval( angBall, angBody,

VecPosition::normalizeAngle(angBody+180) ) )
       ? 1
       : -1  ;

  //  if( iSign == -1 )
  // angBall = VecPosition::normalizeAngle( angBall + 180 );

  soc = turnBodyToPoint( posAgent + VecPosition(1.0,

                                                                    VecPosition::normalizeAngle(
angBody+60*iSign), POLAR ) );
  Log.log( 556, "search ball: turn to %f s %d t(%d %d) %f", angBall, iSign,
          WM->getCurrentTime().getTime(), timeLastSearch.getTime(),
                                                                      soc.dAngle );
  timeLastSearch = WM->getCurrentTime();
  return soc;
}


/*! This method can be called to create a SoccerCommand that dashes to
    a point.  This skill enables an agent to dash to a given point. It
    receives a global position 'pos' as its only argument and returns
    a dash command that causes the agent to come as close to this
    point as possible. Since the agent can only move forwards or
    backwards, the closest point to the target position that he can
    reach by dashing is the orthogonal projection of 'pos' onto the
    line that extends into the direction of his body (forwards and
    backwards).  The power that must be supplied to the dash command
    is computed using the 'getPowerForDash' method which takes the
    position of 'pos' relative to the agent as input and 'iCycles'
    which denotes in how many cycles we want to reach that point.
    \param pos global position to which the agent wants to dash
    \param iCycles desired number of cycles to reach point 'pos'
    \return SoccerCommand dash command to move closer to 'pos' */
SoccerCommand BasicPlayer::dashToPoint( VecPosition pos, int iCycles )
{
  double dDashPower = WM->getPowerForDash(
                          pos - WM->getAgentGlobalPosition(),
                          WM->getAgentGlobalBodyAngle(),
```

```
                        WM->getAgentGlobalVelocity(),
                        WM->getAgentEffort(),
                        iCycles                      );
  return SoccerCommand( CMD_DASH, dDashPower );
}

/*! This skill enables an agent to freeze a moving ball, i.e. it returns a kick
    command that stops the ball dead at its current position. Since ball
    movement in the soccer server is implemented as a vector addition, the ball
    will stop in the next cycle when it is kicked in such a way that the
    resulting acceleration vector has the same length and opposite direction to
    the current ball velocity. The desired speed that should be given to the
    ball on the kick thus equals the current ball speed. Furthermore, the
    direction of the kick should equal the direction of the current ball
    velocity plus 180 degrees. Note that this direction must be made relative
    to the agent's global body angle before it can be passed as an argument to
    the kick command.
    \return SoccerCommand to freeze the ball. */
SoccerCommand BasicPlayer::freezeBall( )
{
  // determine power needed to kick the ball to compensate for current speed
  // get opposite direction (current direction + 180) relative to body
  // and make the kick command.
  VecPosition posAgentPred = WM->predictAgentPos( 1, 0 );

  double dPower = WM->getKickPowerForSpeed( WM->getBallSpeed() );
  if( dPower > SS->getMaxPower() )
  {
    Log.log( 552, "%d: freeze ball has to much power", WM->getCurrentCycle() );
    dPower = (double)SS->getMaxPower();
  }
  double dAngle = WM->getBallDirection() + 180 - WM->getAgentGlobalBodyAngle();
  dAngle        = VecPosition::normalizeAngle( dAngle );
  SoccerCommand soc( CMD_KICK, dPower, dAngle );
  VecPosition posBall, velBall;
  WM->predictBallInfoAfterCommand( soc, &posBall, &velBall );
  if( posBall.getDistanceTo( posAgentPred ) < 0.8 * SS->getMaximalKickDist() )
    return soc;
  Log.log( 101, "freeze ball will end up oustide -> accelerate" );
  posBall = WM->getBallPos();
  // kick ball to position inside to compensate when agent is moving
  VecPosition posTo   = posAgentPred +
    VecPosition( min( 0.7 * SS->getMaximalKickDist(),
                                                          posBall.getDistanceTo(
posAgentPred ) - 0.1 ),
```

```
                                                                    (posBall-
posAgentPred).getDirection(),
                                                                    POLAR );
  VecPosition velDes( posTo - posBall );
  return accelerateBallToVelocity( velDes );
}

/*! This skill enables an agent to kick the ball close to his body. It receives
    an angle 'ang' as its only argument and returns a kick command that causes
    the ball to move to a point at a relative angle of 'ang' degrees and at a
    close distance (kickable margin/6 to be precise) from the agent's body.
    To this end the ball has to be kicked from its current position to the
    desired point relative to the predicted position of the agent in the next
    cycle. In general, this skill will be used when the agent wants to kick the
    ball to a certain position on the field which cannot be reached with a
    single kick. Since the efficiency of a kick is highest when the ball is
    positioned just in front of the agent's body, calling this skill with
    'ang = 0' will have the effect that the agent can kick the ball with more
    power after it is executed.
    Note that this skill will only be executed if it is possible to actually
    reach the desired ball position with a single kick. If the required power
    does exceed the maximum then the ball is frozen at its current position
    using the freezeBall skill. In general, it will then always be possible to
    shoot the motionless ball to the desired point in the next cycle.
    \param 'ang' relative angle to body to which the ball should be kicked
    \param relative ratio to which the ball is kicked
    \return SoccerCommand to kick the ball close to the body */
SoccerCommand BasicPlayer::kickBallCloseToBody( AngDeg ang, double dKickRatio )
{
  AngDeg      angBody    = WM->getAgentGlobalBodyAngle();
  VecPosition posAgent   = WM->predictAgentPos( 1, 0 );
  double      dDist      = SS->getPlayerSize() +
                           SS->getBallSize()   +
                           SS->getKickableMargin()*dKickRatio;
  AngDeg      angGlobal  = VecPosition::normalizeAngle( angBody + ang );
  VecPosition posDesBall = posAgent   + VecPosition( dDist, angGlobal, POLAR );
  if( fabs( posDesBall.getY() ) > PITCH_WIDTH/2.0 ||
      fabs( posDesBall.getX() ) > PITCH_LENGTH/2.0 )
  {
    Line lineBody = Line::makeLineFromPositionAndAngle( posAgent, angGlobal );
    Line lineSide(0,0,0);
    if( fabs( posDesBall.getY() ) > PITCH_WIDTH/2.0 )
      lineSide = Line::makeLineFromPositionAndAngle(
                                                                    VecPosition( 0,
sign(posDesBall.getY() )* PITCH_WIDTH/2.0 ), 0 );
    else
```

```
     lineSide = Line::makeLineFromPositionAndAngle(
                                                              VecPosition( 0,
sign(posDesBall.getX() )* PITCH_LENGTH/2.0 ),  90 );
    VecPosition posIntersect = lineSide.getIntersection( lineBody );
    posDesBall = posAgent +
      VecPosition( posIntersect.getDistanceTo( posAgent ) - 0.2,
                                                                angGlobal, POLAR );
  }

  VecPosition vecDesired = posDesBall - WM->getBallPos();
  VecPosition vecShoot   = vecDesired - WM->getGlobalVelocity( OBJECT_BALL );
  double      dPower     = WM->getKickPowerForSpeed( vecShoot.getMagnitude() );
  AngDeg      angActual  = vecShoot.getDirection() - angBody;
              angActual  = VecPosition::normalizeAngle( angActual );

  if( dPower > SS->getMaxPower() && WM->getBallSpeed() > 0.1 )
  {
    Log.log( 500, "kickBallCloseToBody: cannot compensate ball speed, freeze");
    Log.log( 101, "kickBallCloseToBody: cannot compensate ball speed, freeze");
    return freezeBall();
  }
  else if( dPower > SS->getMaxPower() )
  {
    if( WM->isDeadBallUs() )
    {
      if( WM->getRelativeAngle( OBJECT_BALL ) > 25 )
      {
                                                        Log.log( 101, "dead ball
situation, turn to ball" );
                                                        return turnBodyToObject(
OBJECT_BALL );
      }
    }
    else
    {
      Log.log( 101, "kickBallCloseToBody: ball has no speed, but far away" );
      dPower = 100;
    }
  }
  else
    Log.log( 101, "(kick %f %f), vecDesired (%f,%f) %f posDes(%f,%f)",
                                                        dPower,
angActual,vecDesired.getX(), vecDesired.getY(),ang,
                                                        posDesBall.getX(),
posDesBall.getY() );
  return SoccerCommand( CMD_KICK, dPower, angActual );
```

```
}

/*! This skill enables an agent to accelerate the ball in such a way that it
    gets a certain velocity after the kick. It receives the desired velocity
    'vecDes' as its only argument and returns a kick command that causes the
    ball to be accelerated to this velocity. If the power that must be supplied
    to the kick command to get the desired result does not exceed the maximum
    kick power then the desired velocity can be realized with a single kick.
    The kick direction should then be equal to the direction of the
    acceleration vector relative to the agent's global body angle. However,
    if the desired velocity is too great or if the current ball velocity is
    too high then the required acceleration cannot be realized with a single
    kick. In this case, the ball is kicked in such a way that the acceleration
    vector has the maximum possible length and a direction that aligns the
    resulting ball movement with 'vecDes'. This means that after the kick the
    ball will move in the same direction as 'vecDes' but at a lower speed. To
    this end the acceleration vector has to compensate for the current ball
    velocity in the `wrong' direction (y-component).
    \param velDes desired ball velocity
    \return SoccerCommand that accelerates the ball to 'vecDes' */
SoccerCommand BasicPlayer::accelerateBallToVelocity( VecPosition velDes )
{
  AngDeg      angBody  = WM->getAgentGlobalBodyAngle();
  VecPosition velBall  = WM->getGlobalVelocity( OBJECT_BALL );
  VecPosition accDes   = velDes - velBall;
  double      dPower;
  double      angActual;

  // if acceleration can be reached, create shooting vector
  if( accDes.getMagnitude() < SS->getBallAccelMax() )
  {
    dPower    = WM->getKickPowerForSpeed   ( accDes.getMagnitude() );
    angActual = VecPosition::normalizeAngle( accDes.getDirection() - angBody );
    if( dPower <= SS->getMaxPower() )
      return SoccerCommand( CMD_KICK, dPower, angActual );
  }

  // else determine vector that is in direction 'velDes' (magnitude is lower)
        dPower    = SS->getMaxPower();
  double dSpeed    = WM->getActualKickPowerRate() * dPower;
  double tmp       = velBall.rotate(-velDes.getDirection()).getY();
        angActual = velDes.getDirection() - asinDeg( tmp / dSpeed );
        angActual = VecPosition::normalizeAngle( angActual - angBody );

  return SoccerCommand( CMD_KICK, dPower, angActual );
}
```

```
/*! This skill enables an agent to catch the ball and can only be executed
    when the agent is a goalkeeper. It returns a catch command that takes the
    angle of the ball relative to the body of the agent as its only argument.
    The correct value for this argument is computed by determining the global
    direction between the current ball position and the agent's current
    position and by making this direction relative to the agent's global body
    angle.
    \return SoccerCommand to catch the ball */
SoccerCommand BasicPlayer::catchBall()
{
  // true means returned angle is relative to body instead of neck
  return SoccerCommand( CMD_CATCH, WM->getRelativeAngle( OBJECT_BALL, true ));
}


/*! This skill enables an agent to communicate with other players on the field.
    It receives a string message as its only argument and returns a say command
    that causes the message to be broadcast to all players within a certain
    distance from the speaker.
    \return SoccerCommand to say the specified string 'str' */
SoccerCommand BasicPlayer::communicate( char *str )
{
  return SoccerCommand( CMD_SAY, str );
}

/*! This method returns a 'move' command to teleport the agent directly to the
    specified global position.
    \param pos global position to which should be moved.
    \return SoccerCommand to move directly to 'pos'. */
SoccerCommand BasicPlayer::teleportToPos( VecPosition pos )
{
  return SoccerCommand( CMD_MOVE, pos.getX(), pos.getY() );

}

/*! This method returns a 'attentionto' command to listen to the specified
    object. In most occasions this is a teammate. */
SoccerCommand BasicPlayer::listenTo( ObjectT obj )
{
  if( !SoccerTypes::isKnownPlayer( obj ) )
    return SoccerCommand( CMD_ATTENTIONTO, -1.0, -1.0 );

  return SoccerCommand( CMD_ATTENTIONTO,
     1.0,                              // 1.0 denotes our team
     SoccerTypes::getIndex( obj ) + 1 );
```

```
}

/*! This method returns the command to tackle the ball. */
SoccerCommand BasicPlayer::tackle( )
{
  return SoccerCommand( CMD_TACKLE, 100.0 );
}

/********************** INTERMEDIATE LEVEL SKILLS
*****************************/

/*! This skill enables an agent to turn his body towards an object o which is
    supplied to it as an argument. To this end the object's global position
    o in the next cycle is predicted based on its current velocity.
    This predicted position is passed as an argument to the turnBodyToPoint
    skill which generates a turn command that causes the agent to turn his
    body towards the object.
    \param o object to which agent wants to turn
    \return SoccerCommand that turns to this object */
SoccerCommand BasicPlayer::turnBodyToObject( ObjectT o )
{
  return turnBodyToPoint( WM->predictPosAfterNrCycles(o, 1) );
}

/*! This skill enables an agent to turn his neck towards an object. It
    receives as arguments this object o as well as a primary action command
    'soc' that will be executed by the agent at the end of the current cycle.
    Turning the neck towards an object amounts to predicting the object's
    global position in the next cycle and passing this predicted position
    together with the 'soc' command as arguments to the turnNeckToPoint skill.
    This low-level skill will then generate a turn neck command that causes the
    agent to turn his neck towards the given object. Note that the 'soc'
    command is supplied as an argument for predicting the agent's global
    position and neck angle after executing the command. This is necessary
    because a turn neck command can be executed in the same cycle as a kick,
    dash, turn , move or catch command.
   \param o object to which the agent wants to turn his neck
   \param soc SoccerCommand that is performed in this cycle.
   \return SoccerCommand that turns the neck of the agent to this object */
SoccerCommand BasicPlayer::turnNeckToObject( ObjectT o, SoccerCommand soc )
{
  return turnNeckToPoint( WM->predictPosAfterNrCycles(o, 1), soc );
}

SoccerCommand BasicPlayer::directTowards( VecPosition posTurnTo,
   AngDeg angWhenToTurn, VecPosition *pos, VecPosition *vel, AngDeg *angBody  )
```

```
{
//  return turnBodyToPoint( posTurnTo );
  // copy values or initialize when not set
  VecPosition posAgent= (pos    ==NULL)?WM->getAgentGlobalPosition ():*pos;
  VecPosition velAgent= (vel    ==NULL)?WM->getAgentGlobalVelocity ():*vel;
  AngDeg  angBodyAgent= (angBody==NULL)?WM->getAgentGlobalBodyAngle():*angBody;

  // first predict what happens when the agents rolls out.
  VecPosition posPred    = WM->predictFinalAgentPos();
  AngDeg      angTo      = ( posTurnTo - posPred ).getDirection();
  AngDeg      ang        = VecPosition::normalizeAngle( angTo - angBodyAgent );
  AngDeg      angNeck = 0;

  int iTurn = 0;
  while( fabs( ang ) > angWhenToTurn && iTurn < 5 )
  {
   iTurn++;
   WM->predictStateAfterTurn(
        WM->getAngleForTurn( ang, velAgent.getMagnitude() ),
        &posAgent,
        &velAgent,
        &angBodyAgent,
        &angNeck );
   ang = VecPosition::normalizeAngle( angTo - angBodyAgent );
  }
  Log.log( 509, "direct towards: %d turns", iTurn );
  posAgent = (pos    ==NULL)?WM->getAgentGlobalPosition ():*pos;
  velAgent = (vel    ==NULL)?WM->getAgentGlobalVelocity ():*vel;
  angBodyAgent = (angBody==NULL)?WM->getAgentGlobalBodyAngle():*angBody;

  switch( iTurn )
  {
   case 0:   cerr << "direct towards: 0 turns" ;
           return SoccerCommand( CMD_ILLEGAL );
   case 1:
   case 2:   return turnBodyToPoint( posTurnTo, 2 );
   default:  return dashToPoint(
          (pos==NULL)?WM->getAgentGlobalPosition ():*pos   );  // stop
  }
}

/*! This skill enables an agent to move to a global position 'pos' on
    the field which is supplied to it as an argument. Since the agent
    can only move forwards or backwards into the direction of his
    body, the crucial decision in the execution of this skill is
    whether he should perform a turn or a dash. Turning has the
```

```
   advantage that in the next cycle the agent will be orientated
   correctly towards the point he wants to reach. However, it has the
   disadvantage that performing the turn will cost a cycle and will
   reduce the agent's velocity since no acceleration vector is added
   in that cycle.  Apart from the target position 'pos', this skill
   receives several additional arguments for determining whether a
   turn or dash should be performed in the current situation. If the
   target point is in front of the agent then a dash is performed
   when the relative angle to this point is smaller than a given
   angle 'angWhenToTurn'. However, if the target point is behind the
   agent then a dash is only performed if the distance to point is
   less than a given value 'dDistBack' and if the angle relative to
   the back direction of the agent is smaller than
   'angWhenToTurn'. In all other cases a turn is performed. Note that
   in the case of the goalkeeper it is sometimes desirable that he
   moves backwards towards his goal in order to keep sight of the
   rest of the field. To this end an additional boolean argument
   'bMoveBack' is supplied to this skill that indicates whether the
   agent should always move backwards to the target point. If this
   value equals true then the agent will turn his back towards the
   target point if the angle relative to his back direction is larger
   than 'angToTurn'. In all other cases he will perform a (backward)
   dash towards 'posTo' regardless of whether the distance to this
   point is larger than 'dDistBack'.

   \param posTo global target position to which the agent wants to move
   \param angWhenToTurn angle determining when turn command is returned
   \param dDistBack when posTo lies closer than this value to the back of
        the agent (and within angWhenToTurn) a backward dash is returned
   \param bMoveBack boolean determing whether to move backwards to 'posTo'
   \return SoccerCommand that determines next action to move to 'posTo' */
SoccerCommand BasicPlayer::moveToPos( VecPosition posTo, AngDeg angWhenToTurn,
                    double dDistBack, bool bMoveBack, int iCycles )
{
// previously we only turned relative to position in next cycle, now take
// angle relative to position when you're totally rolled out...
//  VecPosition posPred   = WM->predictAgentPos( 1, 0 );
  VecPosition posAgent  = WM->getAgentGlobalPosition();
  VecPosition posPred   = WM->predictFinalAgentPos();

  AngDeg      angBody   = WM->getAgentGlobalBodyAngle();
  AngDeg      angTo     = ( posTo - posPred ).getDirection();
              angTo     = VecPosition::normalizeAngle( angTo - angBody );
  AngDeg      angBackTo = VecPosition::normalizeAngle( angTo + 180 );

  double      dDist     = posAgent.getDistanceTo( posTo );
```

```
  Log.log( 509, "moveToPos (%f,%f): body %f to %f diff %f now %f when %f",
         posTo.getX(), posTo.getY(), angBody,
          ( posTo - posPred ).getDirection(), angTo,
          ( posTo -  WM->predictAgentPos( 1, 0 )).getDirection(),
          angWhenToTurn );
  if( bMoveBack )
  {
    if( fabs( angBackTo ) < angWhenToTurn )
      return dashToPoint( posTo, iCycles );
    else
      return turnBackToPoint( posTo );
  }
  else if(  fabs( angTo     ) < angWhenToTurn ||
          (fabs( angBackTo ) < angWhenToTurn && dDist < dDistBack ) )
    return dashToPoint( posTo, iCycles );
  else
    return directTowards( posTo, angWhenToTurn );
//return turnBodyToPoint( posTo );
}


/*! This method returns a command that can be used to collide with the ball
    on purpose. When this is not possible. CMD_ILLEGAL is returned. Colliding
    with the ball may be useful when the player is turned with his back to the
    opponent goal and is intercepting a moving ball, by colliding both the ball
    and the player will loose all their velocity. Now the player can turn at
    once to the desired direction. Otherwise he first has to freeze the ball,
    freeze his own speed and then turn around. */
SoccerCommand BasicPlayer::collideWithBall( )
{
  SoccerCommand soc( CMD_ILLEGAL );
  if( WM->getRelativeDistance( OBJECT_BALL ) >
      WM->getBallSpeed() + SS->getPlayerSpeedMax() )
    return soc;

  VecPosition posBallPred  = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );

  // first try turn
  soc                   = turnBodyToPoint( WM->getAgentGlobalPosition() +
                                           VecPosition( 1, 0, POLAR ) );
  VecPosition posAgentPred = WM->predictAgentPosAfterCommand( soc );
  if( posAgentPred.getDistanceTo( posBallPred ) <
      SS->getBallSize() + SS->getPlayerSize() )
  {
    Log.log( 511, "can collide with ball by turning" );
```

```
    return soc;
  }

  soc          = dashToPoint( posBallPred, 1 );
  posAgentPred = WM->predictAgentPosAfterCommand( soc );
  if( posAgentPred.getDistanceTo( posBallPred ) <
      SS->getBallSize() + SS->getPlayerSize() )
  {
    Log.log( 511, "can collide with ball by dashing %f", soc.dPower );
    return soc;
  }

  return SoccerCommand( CMD_ILLEGAL );
}
/*! This skill enables an agent to intercept a ball which is close to him. The
    objective is to move in such a way that the ball will come within the
    kickable distance from the agent in one or two cycles. To this end the
    prediction methods from the world model are used to predict the ball
    position in the next cycle and two cycles into the future. It is then
    determined whether it is possible to move the agent within kickable
    distance from one of these positions using all logical combinations of
    turn and dash commands. If it is not possible to intercept the ball within
    two cycles then this skill returns an illegal command to indicate that
    it cannot be performed. First it is determined whether the agent can
    intercept the ball in one cycle. To this end the position of the ball in
    the next cycle is predicted and a calculation is performed to decide
    whether a single dash can move the agent within the kickable distance from
    this position. In order to be able to kick the ball efficiently after
    intercepting it, it is important that the agent moves to a good position
    relative to the ball (i.e. the ball must be in front of him). At the same
    time the agent must make sure that he does not collide with the ball when
    trying to intercept it. Let l be a line that runs forwards and backwards
    from the predicted position of the agent in the next cycle into the
    direction of his body. This line thus denotes the possible movement
    direction of the agent. Note that we have to use the agent's predicted
    position in the next cycle since his current velocity must be taken into
    account. In addition, let c be a circle which is centered on the predicted
    ball position and which has a radius equal to the sum of the radius of the
    agent, the radius of the ball and a small buffer (kickable margin/6). It
    is now determined whether the agent can intercept the ball in the next
    cycle by looking at the number of intersection points between l and c. If
    l and c have exactly one point in common then this point is the desired
    interception point for the next cycle. However, if the number of
    intersection points equals two then the desired point is the one for
    which the absolute angle of the ball relative to that point is the
    smallest. This amounts to the intersection point which is closest to
```

```
   the agent when the ball lies in front of him and to the furthest one when
   the ball is behind his back. As a result, the desired interception point
   will always be such that the agent has the ball in front of him in the
   next cycle. Then a dash command is generated
   that will bring the agent as close as possible to the desired point.
   Next, the position of the agent after executing this command is predicted
   and if it turns out that this predicted position lies within the kickable
   distance from the ball then the dash is performed. However, if the
   predicted position is not close enough to the ball or if l and c have no
   points in common then it is assumed that the ball cannot be intercepted
   with a single dash. In these cases, two alternatives are explored to see
   if the ball can be intercepted in two cycles.
   The first alternative is to determine whether the agent can intercept the
   ball by performing a turn followed by a dash. To this end the global
   position of the ball is predicted two cycles into the future and a turn
   command is generated that will turn
   the agent towards this point. The agent's position after executing this
   command is then predicted after which a dash command is generated
   that will bring the agent as close as possible to
   the predicted ball position in two cycles. If it turns out that the
   predicted position of the agent after the dash lies within kickable
   distance from the ball then the first command (i.e. the turn) in the
   sequence of two is performed. Otherwise, a second alternative is tried to
   determine whether the agent can intercept the ball by performing two dash
   commands. To this end two dash commands are generated to get closer to the
   predicted ball position after two cycles.
   If the predicted position of the agent after these two dashes lies within
   kickable distance from the ball then the first dash is performed.
   Otherwise, an illegal command is returned to indicate that the skill
   cannot be performed. The close interception procedure is heavily based on
   a similar method introduced in CMU'99 by Peter Stone.
   \return command to intercept ball in two cycles, CMD_ILLEGAL otherwise */
SoccerCommand BasicPlayer::interceptClose( )
{
  FeatureT feature_type = FEATURE_INTERCEPT_CLOSE;
  if( WM->isFeatureRelevant( feature_type ) )
    return WM->getFeature( feature_type ).getCommand();

  SoccerCommand soc, socDash1, socFinal, socCollide, socTurn(CMD_ILLEGAL);
  double        dPower, dDist;
  AngDeg        ang,    ang2;
  VecPosition   s1,     s2;
  bool          bReady = false;

  // first determine whether the distance to the ball is not too large
  dDist = 3*SS->getPlayerSpeedMax()
```

```
      + (1.0 + SS->getBallDecay())*SS->getBallSpeedMax()
      + SS->getMaximalKickDist();
if( WM->getRelativeDistance( OBJECT_BALL ) > dDist )
{
 bReady = true;
 socFinal = SoccerCommand( CMD_ILLEGAL ); // do not quit, but log feature
}

socCollide = collideWithBall( );
// initialize all variables with information from the worldmodel.
VecPosition   posAgent = WM->getAgentGlobalPosition( );
VecPosition   posPred  = WM->predictAgentPos( 1, 0 ), posDash1;
VecPosition   posBall  = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
VecPosition   velMe    = WM->getAgentGlobalVelocity( );
Stamina       sta      = WM->getAgentStamina( );
AngDeg        angBody  = WM->getAgentGlobalBodyAngle( ), angTurn, angNeck=0;
double        dDesBody = 0.0;

// our desired heading after the intercept is 0 degrees, only when we
// are far up front we want to be headed toward the opponent goal
if( posAgent.getX() > PENALTY_X - 5.0 )
 dDesBody = (WM->getPosOpponentGoal()-posAgent).getDirection();

// get the distance to the closest opponent
double dDistOpp;
ObjectT objOpp = WM->getClosestInSetTo( OBJECT_SET_OPPONENTS,
  WM->getAgentObjectType(), &dDistOpp, PS->getPlayerConfThr() );
angTurn =VecPosition::normalizeAngle(dDesBody-WM->getAgentGlobalBodyAngle());

// check the distance to the ball when we do not dash (e.g., and only turn)
posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
VecPosition posPred1 = WM->predictAgentPos( 1, 0 );
double dDist1 = posPred1.getDistanceTo( posBall );
posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
VecPosition posPred2 = WM->predictAgentPos( 2, 0 );
double dDist2 = posPred2.getDistanceTo( posBall );
posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 3 );
VecPosition posPred3 = WM->predictAgentPos( 3, 0 );
double dDist3 = posPred3.getDistanceTo( posBall );
Log.log( 508, "dist 1: %f, 2: %f 3: %f, 0.6: %f", dDist1, dDist2, dDist3,
                          0.7*SS->getMaximalKickDist() );

AngDeg angThreshold = 25;
bool   bOppClose = ( objOpp != OBJECT_ILLEGAL && dDistOpp < 3.0 )  ;

// make a line from center of body in next cycle with direction of body
```

```
  // use next cycle since current velocity is always propogated to position in
  // next cycle.  Make a circle around the ball with a radius equal to the
  // sum of your own body, the ball size and a small buffer. Then calculate
  // the intersection between the line and this circle. These are the (two)
  // points that denote the possible agent locations close to the ball
  // From these two points we take the point where the body direction of the
  // agent makes the smallest angle with the ball (with backward
  // dashing we sometime have to dash "over" the ball to face it up front)
  posAgent  = WM->getAgentGlobalPosition( );
  posBall   = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
  angBody   = WM->getAgentGlobalBodyAngle();
  velMe     = WM->getAgentGlobalVelocity( );
  sta       = WM->getAgentStamina( );
  Line line = Line::makeLineFromPositionAndAngle(posPred1,angBody);
  dDist     = SS->getPlayerSize()+SS->getBallSize()+SS->getKickableMargin()/6;
  int  iSol = line.getCircleIntersectionPoints(
                                Circle(posBall,dDist), &s1, &s2);
  if (iSol > 0)                                 // if a solution
  {
   if (iSol == 2)                                // take the best one
   {
    ang = VecPosition::normalizeAngle((posBall - s1).getDirection() -angBody);
    ang2= VecPosition::normalizeAngle((posBall - s2).getDirection() -angBody);
//     if ( fabs(ang2) < 90)
    if( s2.getX() > s1.getX() ) // move as much forward as possible
      s1 = s2;                                  // and put it in s1
   }

   // try one dash
   // now we have the interception point we try to reach in one cycle. We
   // calculate the needed dash power from the current position to this point,
   // predict were we will stand if we execute this command and check whether
   // we are in the kickable distance
   dPower = WM->getPowerForDash(s1-posAgent, angBody, velMe,sta.getEffort() );
   posDash1 = WM->predictAgentPos( 1, (int)dPower);
   if ( posDash1.getDistanceTo( posBall ) < 0.95*SS->getMaximalKickDist() )
   {
    Log.log( 508, "dash 1x possible at s1" );
    socDash1 = SoccerCommand( CMD_DASH, dPower );
   }
   else
   {
    dPower=WM->getPowerForDash(s2-posAgent, angBody, velMe,sta.getEffort() );
    posDash1 = WM->predictAgentPos( 1, (int)dPower);
    if ( posDash1.getDistanceTo( posBall ) < 0.95*SS->getMaximalKickDist() )
    {
```

```
      Log.log( 508, "dash 1x possible at s2" );
      socDash1 = SoccerCommand( CMD_DASH, dPower );
     }
    }
  }

// try one dash by getting close to ball
// this handles situation where ball cannot be reached within distance
// SS->getKickableMargin()/6
if( socDash1.commandType == CMD_ILLEGAL )
{
  soc = dashToPoint( posBall );
  WM->predictAgentStateAfterCommand(soc,&posDash1,&velMe,
                                &angBody,&ang,&sta );
  if ( posDash1.getDistanceTo( posBall ) < 0.95*SS->getMaximalKickDist() )
  {
    Log.log( 508, "dash 1x possible (special)" );
    socDash1 = soc;
  }
}

if( bReady != true )
{
  if( bOppClose && ! socDash1.isIllegal() )
  {
    Log.log( 508, "do dash 1x, opponent close" );
    WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
    socFinal = socDash1;
  }
  else
  {
    soc = turnBodyToPoint( posPred1 + VecPosition(1,dDesBody, POLAR), 1 );
    WM->predictAgentStateAfterCommand(soc, &posPred, &velMe,
                                  &angBody, &ang, &sta);
    posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
    if( posPred.getDistanceTo( posBall ) < 0.8*SS->getMaximalKickDist() )
    {
      socTurn = soc; // we can do turn and end up ok, but can maybe improve
      ang     = VecPosition::normalizeAngle(dDesBody-angBody);
      if( fabs(ang) < angThreshold )
      {
        socFinal = soc;
        Log.log( 508, "turn 1x, dist %f, angle %f, opp %f",
           dDist1, angTurn, dDistOpp );
        WM->logCircle( 508, posPred1, SS->getMaximalKickDist(), true );
      }
```

```
}
if( socFinal.isIllegal() )
{
  ang     = VecPosition::normalizeAngle(dDesBody-angBody);
  WM->predictStateAfterTurn(
            WM->getAngleForTurn(ang,velMe.getMagnitude()),
            &posPred, &velMe, &angBody, &angNeck,
            WM->getAgentObjectType(),
            &sta              );
  posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
  if( posPred.getDistanceTo( posBall ) < 0.8*SS->getMaximalKickDist() )
  {
    socTurn = soc; // we can do turn and end up ok, but can maybe improve
    ang     = VecPosition::normalizeAngle(dDesBody-angBody);
    if( fabs(ang) < angThreshold )
    {
      Log.log( 508, "turn 2x, dist %f, angle %f, opp %f",
           dDist2, angTurn, dDistOpp );
      WM->logCircle( 508, posPred2, SS->getMaximalKickDist(), true );
      socFinal = soc;
    }
  }
}
if( socFinal.isIllegal() && ! socCollide.isIllegal() &&
    fabs( angTurn ) > angThreshold )
{
  Log.log( 508, "collide with ball on purpose" );
  posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
  WM->logCircle( 508, posBall, SS->getMaximalKickDist(), true );
  socFinal = socCollide;
}
if( socFinal.isIllegal() && fabs( angTurn ) > angThreshold )
{
  posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
  soc = dashToPoint( posAgent );
  WM->predictAgentStateAfterCommand(soc,
                              &posPred,&velMe,&angBody,&ang,&sta );
  if( posPred.getDistanceTo( posBall ) < 0.8*SS->getMaximalKickDist() )
  {
    Log.log( 508, "dash 1x (stop), turn 1x, dist %f, angle %f, opp %f",
     dDist2, angTurn, dDistOpp );
    WM->logCircle( 508, posPred, SS->getMaximalKickDist(), true );
    socFinal = soc;
  }
}
```

```
    if( socFinal.isIllegal()  && ! socTurn.isIllegal() )
    {
      Log.log( 508, "can do turn" );
      WM->logCircle( 508, posPred1, SS->getMaximalKickDist(), true );
      socFinal = socTurn;
    }
    if( socFinal.isIllegal()  && ! socDash1.isIllegal() )
    {
      Log.log( 508, "do dash 1x" );
      WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
      socFinal = socDash1;
    }
  }

}
// if there are no opponents, we are wrongly directed, and we will be closely
// to the ball, see whether we can first update our heading
else if( fabs( angTurn ) > angThreshold && !bOppClose &&
    dDist1 < 0.7*SS->getMaximalKickDist() )
{
  soc = turnBodyToPoint( posPred1 + VecPosition(1,dDesBody, POLAR), 1 );
  Log.log( 508, "turn 1x, dist %f, angle %f, opp %f",
    dDist1, angTurn, dDistOpp );
  WM->logCircle( 508, posPred1, SS->getMaximalKickDist(), true );
  socFinal = soc;
}
else if( // fabs( angTurn ) > angThreshold &&
    !bOppClose &&
    dDist2 < 0.7*SS->getMaximalKickDist() )
{
  soc = turnBodyToPoint( posPred2 + VecPosition(1,dDesBody, POLAR), 2 );
  Log.log( 508, "turn 2x, dist %f, angle %f, opp %f",
     dDist2, angTurn, dDistOpp );
  WM->logCircle( 508, posPred2, SS->getMaximalKickDist(), true );
  socFinal = soc;
}


else if( socCollide.commandType != CMD_ILLEGAL &&
        fabs( angTurn ) > angThreshold )
{
  Log.log( 508, "collide with ball on purpose" );
  WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
  socFinal = socCollide;
}
else if( socDash1.commandType != CMD_ILLEGAL )
```

```
  {
   Log.log( 508, "do dash 1x" );
   WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
   socFinal = socDash1;
  }


  // cannot intercept ball in three cycles
  WM->setFeature( feature_type,
          Feature( WM->getTimeLastSeeMessage(),
                WM->getTimeLastSenseMessage(),
                WM->getTimeLastHearMessage(),
                OBJECT_ILLEGAL,
                -1,
                socFinal ) );
  return socFinal;
}

/*! This skill enables a goalkeeper to intercept a ball which is close to him.
    The objective is to move in such a way that the ball will come within the
    catchable distance from the agent in one or two cycles. To this end the
    prediction methods from the world model are used to predict the ball
    position in the next cycle and two cycles into the future. It is then
    determined whether it is possible to move the agent within the
    catchable area from one of these positions using all logical combinations
    of turn and dash commands. If it is not possible to intercept the ball
    within two cycles then this skill returns an illegal command to indicate
    that it cannot be performed. First it is determined whether the goalkeeper
    can intercept the ball in one cycle. To this end the position of the ball
    in the next cycle is predicted and a calculation is performed to decide
    whether a single dash can move the agent within catchable distance from
    this position. If it turns out that this is the case, the corresponding
    dash is performed. However, if the predicted position is not close enough
    to the ball then it is assumed that the ball cannot be intercepted with a
    single dash. In these cases, two alternatives are explored to see if the
    ball can be intercepted in two cycles.
    The first alternative is to determine whether the agent can intercept the
    ball by performing two dash commands. To this end two dash commands are
    generated. If the predicted position of the agent after these two
    dashes lies within catchable distance from the ball then the first dash is
    performed. Otherwise, a second alternative is tried to determine whether
    the agent can intercept the ball by performing a turn
    followed by a dash. To this end the global position of the ball is
    predicted two cycles into the future and a turn command is generated
    that will turn the agent towards this point. The agent's position after
    executing this command is then predicted after which a dash command is
```

```
   generated that will bring the agent as close as possible to the predicted
   ball position. If it turns out that the predicted position of the agent
   after the dash lies within catchable distance from the ball then the first
   command (i.e. the turn) in the sequence of two is performed.
   Otherwise, an illegal command is returned to indicate that the
   skill cannot be performed. The close interception procedure is heavily
   based on the method used by CMU'99 by Peter Stone.
   \return command to intercept ball in two cycles, CMD_ILLEGAL otherwise */
SoccerCommand BasicPlayer::interceptCloseGoalie( )
{
 SoccerCommand soc;
 double       dPower, dDist;
 AngDeg       ang;
 VecPosition  posClosestToBall;

 // initialize all variables with information from worldmodel.
 VecPosition  posPred  = WM->predictAgentPos( 1, 0 );
 VecPosition  posBall  = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
 VecPosition  velMe    = WM->getAgentGlobalVelocity( );
 Stamina      sta      = WM->getAgentStamina( );
 AngDeg       angBody  = WM->getAgentGlobalBodyAngle( );
 Line         lineGoalie=Line::makeLineFromPositionAndAngle(posPred,angBody);

 // when it is theoretical possible
 // try one dash and check whether ball is in catchable area
 dDist =SS->getBallSpeedMax()+SS->getPlayerSpeedMax()+SS->getCatchableAreaL();
 if( WM->getRelativeDistance( OBJECT_BALL ) < dDist )
 {
  posClosestToBall = lineGoalie.getPointOnLineClosestTo( posBall );
  dPower           = WM->getPowerForDash(
                     posClosestToBall-posPred,
                     angBody,
                     velMe,
                     sta.getEffort()          );
  posPred          = WM->predictAgentPos( 1, (int)dPower);
  if ( posPred.getDistanceTo( posBall ) < SS->getCatchableAreaL() )
    return SoccerCommand( CMD_DASH, dPower );
 }

 // when it is theoretical possible
 // try two dashes and check whether ball is in catchable area
 // otherwise try first two  dashes and check whether ball is in catchable
 // area, thereafter for turn and dash.
 dDist = SS->getBallSpeedMax()*(1.0+SS->getBallDecay())
         + 2*SS->getPlayerSpeedMax()
         + SS->getCatchableAreaL();
```

```
  if( WM->getRelativeDistance( OBJECT_BALL ) < dDist )
  {
    // try two dashes
    // first predict the position in the next cycle when dash with full power
    // is performed. Then calculate the dash power to reach the point where the
    // ball will be in two cycles and predict the global position of the agent
    // after a dash with this power. If the position is in the catchable area
    // return a dash with full power.
    posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
    soc     = dashToPoint( posBall );
    WM->predictAgentStateAfterCommand(soc,&posPred,&velMe,&angBody,&ang,&sta );
    dPower=WM->getPowerForDash(posBall-posPred,angBody,velMe,sta.getEffort());
    WM->predictStateAfterDash( dPower, &posPred, &velMe, &sta, angBody,
                               WM->getAgentObjectType() );
    if( posPred.getDistanceTo(posBall) < SS->getCatchableAreaL() )
      return soc;

    // try one turn and a dash
    posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
    posPred = WM->predictAgentPos( 2, 0 );
    ang     = (posBall - posPred).getDirection();
    ang     = VecPosition::normalizeAngle( ang - angBody );
    if (fabs( ang ) > PS->getPlayerWhenToTurnAngle() ) // if we want to turn
    {
      soc = turnBodyToPoint( posBall, 2 );             // perform turn
      WM->predictAgentStateAfterCommand(soc,&posPred,&velMe,&angBody,&ang,&sta
);
      dPower=WM->getPowerForDash(posBall-posPred, angBody,
                                 velMe, sta.getEffort());
      WM->predictStateAfterDash( dPower, &posPred, &velMe, &sta, angBody);
      if( posPred.getDistanceTo(posBall) < SS->getCatchableAreaL() )
        return soc;
    }
  }

  // did not succeed
  return SoccerCommand( CMD_ILLEGAL );
}

/*! This skill enables an agent to kick the ball from its current
    position to a given position 'posTarget' in such a way that it has
    a remaining speed equal to 'dEndSpeed' when it reaches this
    position. However, it is possible that the ball cannot reach this
    velocity with a single kick either because the needed magnitude of
    the generated velocity vector exceeds the maximum speed of the
    ball or due to the fact that the current ball speed in combination
```

```
   with the position of the ball relative to the agent make it
   impossible to achieve the required acceleration. If the magnitude
   of needed velocity vector is larger than ball speed max it is
   certain that even in the optimal situation (i.e. if the ball lies
   directly in front of the agent and has zero velocity) the agent
   will not be able to kick the ball to the target position at the
   desired speed. In this case the expected ball movement is computed
   after executing a kick with maximum power into the direction of
   'posTarget'.  If the magnitude of the resulting movement vector is
   larger than a given percentage ('getPlayerWhenToKick' defined in
   the PlayerSettings) of the maximum ball speed then this kick is
   actually performed despite the fact that it cannot produce the
   wanted effect. Otherwise, the agent shoots the ball close to his
   body and directly in front of him using the kickBallCloseToBody
   skill. In this way he will be able to kick the ball with more
   power in the next cycle. However, if the magnitude of the desired
   velocity vector is smaller than ball speed max it is possible to
   reach the target point at the desired speed in the optimal
   situation.  If the power that must be supplied to the kick command
   to achieve this acceleration is less than or equal to the maximum
   power the accelerateBallToVelocity skill is used to perform the
   desired kick.  Otherwise, the agent uses the kickBallCloseToBody
   skill to put the ball in a better kicking position for the next
   cycle.

   \param posTarget target position where the ball should be shot to
   \param endSpeed desired speed ball should have in target position
   \return SoccerCommand that produces kick  */
SoccerCommand BasicPlayer::kickTo( VecPosition posTarget, double dEndSpeed )
{
  VecPosition posBall  = WM->getBallPos();
  VecPosition velBall  = WM->getGlobalVelocity(OBJECT_BALL);
  VecPosition posTraj  = posTarget - posBall;
  VecPosition posAgent = WM->getAgentGlobalPosition();
  VecPosition velDes   = VecPosition(
            WM->getKickSpeedToTravel( posTraj.getMagnitude(), dEndSpeed ),
            posTraj.getDirection(),
            POLAR                                        );
  double      dPower;
  AngDeg      angActual;

  if( WM->predictAgentPos(1, 0 ).getDistanceTo( posBall + velDes ) <
     SS->getBallSize() + SS->getPlayerSize() )
  {
    Line line = Line::makeLineFromTwoPoints( posBall, posBall + velDes );
    VecPosition posBodyProj = line.getPointOnLineClosestTo( posAgent );
```

```
  double dDist = posBall.getDistanceTo( posBodyProj );
  if( velDes.getMagnitude() < dDist )
    dDist -=  SS->getBallSize() + SS->getPlayerSize();
  else
    dDist +=  SS->getBallSize() + SS->getPlayerSize();
  Log.log( 101, "kick results in collision, change velDes from (%f,%f)",
                                                                    velDes.getX(),
velDes.getY() );
  velDes.setVecPosition( dDist, velDes.getDirection(), POLAR );
 }

 Log.log( 101, "ball (%f,%f), agent (%f,%f), to (%f,%f) ang %f %f" ,
       WM->getBallPos().getX(), WM->getBallPos().getY(),
       WM->getAgentGlobalPosition().getX(),
       WM->getAgentGlobalPosition().getY(),
       posTarget.getX(), posTarget.getY(),
       WM->getAgentGlobalBodyAngle(),
       WM->getAgentGlobalNeckAngle() );
 Log.log( 101, "relpos body (%f,%f), vel. ball:(%f,%f) dist: %f (%f,%f,%f)" ,
       WM->getRelativeDistance( OBJECT_BALL ),
       WM->getRelativeAngle( OBJECT_BALL, true ),
        velBall.getX(), velBall.getY(),
        SS->getMaximalKickDist(), SS->getPlayerSize(), SS->getBallSize(),
         SS->getKickableMargin() );


 double   dDistOpp;
 ObjectT  objOpp    = WM->getClosestInSetTo( OBJECT_SET_OPPONENTS,
                                  OBJECT_BALL, &dDistOpp );

 if( velDes.getMagnitude() > SS->getBallSpeedMax() ) // can never reach point
 {
        dPower    = SS->getMaxPower();
  double   dSpeed    = WM->getActualKickPowerRate() * dPower;
  double   tmp       = velBall.rotate(-velDes.getDirection()).getY();
        angActual  = velDes.getDirection() - asinDeg( tmp / dSpeed );
  double   dSpeedPred = (WM->getGlobalVelocity(OBJECT_BALL)+
                    VecPosition(dSpeed,angActual, POLAR )
                   ).getMagnitude();

  // but ball acceleration in right direction is very high
  if( dSpeedPred > PS->getPlayerWhenToKick()*SS->getBallAccelMax() )
  {
   Log.log( 101, "pos (%f,%f) too far, but can acc ball good to %f k=%f,%f",
         velDes.getX(), velDes.getY(), dSpeedPred, dSpeed, tmp );
   return accelerateBallToVelocity( velDes );    // shoot nevertheless
```

```
    }
    else if( WM->getActualKickPowerRate() >
           PS->getPlayerWhenToKick() * SS->getKickPowerRate() )
    {
     Log.log( 101, "point too far, freeze ball" ); // ball well-positioned
     return freezeBall();                          // freeze ball
    }
    else
    {
     Log.log( 101, "point too far, reposition ball (k_r = %f)",
            WM->getActualKickPowerRate()/(SS->getKickPowerRate()) );
     return kickBallCloseToBody( 0 );            // else position ball better
    }
   }
   else                                    // can reach point
   {
    VecPosition accBallDes = velDes - velBall;
    dPower = WM->getKickPowerForSpeed(accBallDes.getMagnitude());
    if( dPower <= 1.05*SS->getMaxPower() || // with current ball speed
                                                                    (dDistOpp < 2.0 && dPower
<= 1.30*SS->getMaxPower() ) )
    {                           // 1.05 since cannot get ball fully perfect
     Log.log( 101, "point good and can reach point %f", dPower );
     return accelerateBallToVelocity( velDes );  // perform shooting action
    }
    else
    {
     Log.log( 101, "point good, but reposition ball since need %f",dPower );
     SoccerCommand soc = kickBallCloseToBody( 0 );
     VecPosition   posPredBall;
     WM->predictBallInfoAfterCommand( soc, &posPredBall );
     dDistOpp = posPredBall.getDistanceTo( WM->getGlobalPosition( objOpp ) );
     return soc;            // position ball better
    }
   }
}


/*! This skill enables an agent to turn towards a global angle while
    keeping the ball in front of him. It is used, for example, when a
    defender has intercepted the ball in his defensive area and faces
    his own goal. In this situation the defender usually wants to pass
    the ball up the field into an area that is currently not visible
    to him and to this end he will first use this skill to turn with
    the ball towards the opponent's goal. Turning with the ball
    requires a sequence of commands to be performed. The ball first
```

```
   has to be kicked to a desired position relative to the agent, then
   it has to be stopped dead at that position and finally the agent
   must turn towards the ball again. Each time when this skill is
   called it has to be determined which part of the sequence still
   has to be executed. This is done as follows.  If the absolute
   difference between the desired angle and the global angle of the
   ball relative to the position of the agent is larger than the
   value 'angKickThr' then the kickBallCloseToBody skill is used to
   kick the ball to a position close to the agent and at the desired
   angle.  Otherwise, it is checked whether the ball still has speed
   from the previous action. If the remaining ball speed exceeds the
   given value 'dFreezeThr' then the ball is stopped dead at its
   current position using the freezeBall skill. In all other cases
   the agent turns his body towards the specified angle 'ang'.

   \param ang global direction in which ball and player should be faced
   \param angKickThr when ball angle difference is larger than this value
                ball is repositioned
   \param dFreezeThr when ball lies correct, but has speed higher than this
                value, ball is frozen.
   \return Soccercommand to turn with the ball to global angle 'ang'.
  */

/*
SoccerCommand BasicPlayer::turnWithBallTo( AngDeg ang, AngDeg angKickThr,
                                double dFreezeThr )
{
  // if opponent is close
  // if ball is located more than 'angKickThr' degrees from ang
  //  kick ball to point right in front of player in direction ang
  // else if ball has still speed higher than 'dFreezeThr'
  /// freezeBall
  // else
  //  turn to direction 'ang'
  VecPosition posAgent = WM->getAgentGlobalPosition();
  VecPosition posBall  = WM->getBallPos();
  AngDeg      angBody  = WM->getAgentGlobalBodyAngle();
  AngDeg      angDiff  = (posBall-posAgent).getDirection() - ang;
           angDiff  = VecPosition::normalizeAngle( angDiff );
  double      dDist;
  ObjectT     objOpp   = WM->getClosestInSetTo( OBJECT_SET_OPPONENTS,
                      WM->getAgentObjectType(), &dDist );
  VecPosition posOpp   = WM->getGlobalPosition( objOpp );

  if( objOpp != OBJECT_ILLEGAL && dDist < 2.5 )
  {
```

```
    if( posBall.getDistanceTo( posOpp ) < posBall.getDistanceTo( posAgent ) )
    {
      ang = (posOpp - posAgent).getDirection() + 180;
      return kickBallCloseToBody( VecPosition::normalizeAngle( ang-angBody ));
    }
  }
  else if( fabs( angDiff ) > angKickThr )
  {
    Log.log( 101, "turnWithBall: kick ball close %f", ang );
    return kickBallCloseToBody( VecPosition::normalizeAngle( ang - angBody ) );
  }

  // hier niet altijd freezen -> kan dan niet meer goed liggen omdat je zelf
  // doorschiet.
  if( WM->getBallSpeed() > dFreezeThr )
  {
    Log.log( 101, "turnWithBall: freeze ball" );
    return freezeBall();
  }

  ACT->putCommandInQueue( alignNeckWithBody() );
  return turnBodyToPoint( posAgent + VecPosition(1.0, ang, POLAR ) );
}
*/
SoccerCommand BasicPlayer::turnWithBallTo( AngDeg ang, AngDeg, double )
{
  // if opponent is closer to the ball than I am
  //  kick ball away from his direction
  // if the ball will be in my kick_range in the next cycle
  //  turn to direction ang
  // if ball will be in kick range if frozen
  //   freeze_ball
  // else
  //   kickBallCloseToBody(ang)
  VecPosition   posAgent= WM->getAgentGlobalPosition();
  VecPosition   posBall = WM->getBallPos();
  AngDeg        angBody = WM->getAgentGlobalBodyAngle();
  double        dDist;
  ObjectT       objOpp  = WM->getClosestInSetTo( OBJECT_SET_OPPONENTS,
                      WM->getAgentObjectType(), &dDist );
  VecPosition   posOpp  = WM->getGlobalPosition( objOpp );
  SoccerCommand soc     = turnBodyToPoint(posAgent+VecPosition(1.0,ang,POLAR));

  if( objOpp != OBJECT_ILLEGAL && dDist < 2.5 )
  {
    AngDeg angBall = (posBall-posAgent).getDirection();
```

```
  AngDeg angOpp  = (posOpp -posAgent).getDirection();
  if( fabs( VecPosition::normalizeAngle( angBall - angOpp ) ) < 90  )
  {
    ang = angOpp + 180;
    Log.log( 101, "turnWithBall: kick ball away from opp at ang %f", angOpp);
    return kickBallCloseToBody( VecPosition::normalizeAngle( ang-angBody ));
  }
}

VecPosition posAgentPred = WM->predictAgentPosAfterCommand( soc );
VecPosition posBallPred  = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
if( posAgentPred.getDistanceTo( posBallPred ) < 0.8*SS->getMaximalKickDist()
 ||
 ( posAgentPred.getDistanceTo(posBallPred) < 0.9*SS->getMaximalKickDist()
   && WM->getBallSpeed() < 0.1 ))
{
  Log.log( 101, "turnWithBall: turn since ball will be kickable in t+1" );
  return soc;
}

posAgentPred = WM->predictPosAfterNrCycles( WM->getAgentObjectType(), 1, 0 );

// if ball will not be in kickable distance in next turn, we have to freeze
// the ball. Do this only when current ball position (same as frozen ball
// position) lies within kickable distance of predicted agent position in
// next cycle. When ball lies at edge of the kickable distance, do not
// freeze, since when vel.is not completely correct can move outside area
// but first we try to collide with the ball, the ball will then lie still
// and we can turn in one cycle to the desired angle in the next cycle
SoccerCommand socCollide = collideWithBall( );
if( ! socCollide.isIllegal( ) )
{
  Log.log( 101, "turnWithBall: collide with ball" );
  return socCollide;
}
else if( posAgentPred.getDistanceTo(posBall) < 0.8*SS->getMaximalKickDist()
        &&
        WM->getBallSpeed() > 0.1  )
{
  Log.log( 101, "turnWithBall: freeze ball" );
  return freezeBall();
}
else
{
  Log.log( 101, "turnWithBall: kick ball close to desired turnangle %f",ang);
  return kickBallCloseToBody( VecPosition::normalizeAngle( ang - angBody ) );
```

```
  }
}
```

```
/*! This skill enables an agent to move along a line to a given
    position 'pos' on this line. It is used, for example, by the
    goalkeeper who often wants to stay on a line in front of his goal
    and move to different positions on this line depending on where
    the ball is located. Furthermore, it can also used by defenders
    for marking an opponent player by moving along a line between this
    player and the ball. The idea is that the agent must try to move
    as fast as possible to the desired point 'pos' along the line
    thereby keeping the number of turns to a minimum to avoid wasting
    cycles. Apart from the target position 'pos', this skill receives
    several additional arguments for determining whether the agent
    should turn or dash in the current situation. Since the agent can
    only move forwards or backwards into the direction of his body, it
    is important that he tries to keep the orientation of his body
    aligned with the direction of the line in order to be able to move
    quickly to the target point. A given angle 'ang' denotes the
    desired body angle (global) of the agent in the point 'pos'. The
    line l can thus be defined as going through 'pos' and having
    global direction 'ang'. Due to the noise that is added to the
    movement of the agent, the orientation of his body will never be
    exactly equal to 'ang' and as a result the agent' s position will
    start to deviate from the line. Each time when this skill is
    called, the agent's desired orientation is therefore slightly
    adjusted depending on his position with respect to l. If the
    distance d between the agent's current position and the line l is
    smaller than the given value 'dDistThr' then 'ang' remains
    unchanged. However, if d exceeds 'dDistThr' then 'ang' is adjusted
    in such a way that the agent will move closer to l in subsequent
    cycles. This is done by either increasing or decreasing the
    desired orientation 'ang' by 'angCorr' degrees depending on which
    side of the line the agent is located and on a prediction of the
    agent's movement in the forthcoming cycles. This prediction is
    represented by a given value 'iSign' which equals 1 if the agent
    is expected to move in the same direction as 'ang' and -1 if he
    will move in the opposite direction.  Adjusting 'ang' in this way
    has the effect that in subsequent cycles the agent will move
    closer to the line again if this is necessary. The final decision
    whether to turn or dash is now made by comparing the agent's
    current body angle to the desired orientation. If the difference
    between these two angles is larger than 'angThr' degrees then the
```

```
    agent uses the turnBodyToPoint skill to turn in the desired
    direction.  Otherwise, the dashToPoint skill is called to move
    towards the target position.

    \param pos global position to which the agent wants to move
    \param ang desired global body angle of agent in position 'pos'
    \param dDistThr threshold value that defines if desired angle is adjusted
    \param iSign indication whether agent predicts that he will move
    in the same direction as 'ang' in the subsequent cycles (iSign=1)
    or in the opposite direction (iSign=-1)
    \param angThr threshold value that specifies when agent will perform a turn
    command
    \param angCorr correction term with which angle is adjusted if necessary
    \return SoccerCommand that will move agent along line defined by position
    'pos' and angle 'ang'. */
SoccerCommand BasicPlayer::moveToPosAlongLine( VecPosition pos, AngDeg ang,
    double dDistThr, int iSign, AngDeg angThr, AngDeg angCorr )
{
  Line       l       = Line::makeLineFromPositionAndAngle( pos, ang );
  VecPosition posBall  = WM->getBallPos();
  VecPosition posAgent = WM->getAgentGlobalPosition();
  AngDeg      angBody  = WM->getAgentGlobalBodyAngle();
  VecPosition posProj  = l.getPointOnLineClosestTo( posAgent );
  double      dDist    = posAgent.getDistanceTo( posProj );
  double      dDiff    = pos.getDistanceTo     ( posProj );

  // if deviated too much from line, compensate
  if( dDist > dDistThr )
  {
    // check on which side of line agent is located
    VecPosition posOrg(0,0);
    Line        m           = Line::makeLineFromTwoPoints( posOrg, posAgent );
    VecPosition posIntersect = l.getIntersection( m );
    int         iSide;
    if( posAgent.getDistanceTo(posOrg) < posIntersect.getDistanceTo( posOrg ) )
      iSide = +1;
    else
      iSide = -1;

    // adjust desired turning angle to move back to line in coming cycles
    ang = ang + iSign * iSide * angCorr;
  }

  Log.log( 553, "y difference to defend point %f", dDiff );
  // if current body angle differs much from desired turning angle, turn body
  if( fabs( VecPosition::normalizeAngle( ang - angBody ) ) > angThr )
```

```
  {
    Log.log( 553, "angle differs too much body = %f, des = %f", angBody, ang );
    return turnBodyToPoint( posAgent + VecPosition( 1.0, ang, POLAR ) );
  }
  else if(  posBall.getDistanceTo( OBJECT_BALL ) < 60 && dDiff > 0.6 )
    return dashToPoint( pos );  // move later when ball is far from pen. area.
  else if( posBall.getDistanceTo( OBJECT_BALL ) < 30  && dDiff > 0.3 )
    return dashToPoint( pos );  // move earlier when is ball near pen. area.
  else
    return SoccerCommand( CMD_ILLEGAL );
}


/********************** HIGH LEVEL SKILLS
************************************/

/*! When the ball-interception skill is called, it is first determined whether
    it is possible for the agent to intercept the ball within two cycles using
    the intermediate player skill closeIntercept (for the goalkeeper
    closeIntereptGoalie). If it turns out that the ball cannot be intercepted
    within  two cycles then the agent uses an iterative scheme to compute the
    optimal interception point. This is done using the method
    'getInterceptionPointBall'.
    \param isGoalie indicates whether the current player is a goalkeeper or not
    \return SoccerCommand to intercept the ball. */
SoccerCommand BasicPlayer::intercept( bool isGoalie )
{
  SoccerCommand soc = (isGoalie)? interceptCloseGoalie():interceptClose(),soc2;
  VecPosition   pos = WM->getAgentGlobalPosition();

  if( soc.commandType != CMD_ILLEGAL
    && isGoalie )
  {
    Log.log( 502, "intercept in two cycles" );
    return soc;
  }
  Log.log( 608, "start intercept, obj %d", WM->getAgentObjectType()  );
  soc2 = WM->predictCommandToInterceptBall( WM->getAgentObjectType(), soc );
  if( soc2.isIllegal() )
    return turnBodyToPoint( pos + VecPosition( 1.0, 0, POLAR ) );
  return soc2;
}

/*! This skill enables an agent to dribble with the ball, i.e. to move with the
    ball while keeping it within a certain distance. This amounts to repeatedly
    kicking  the ball at a certain speed into a desired direction and then
```

```
   intercepting it again.  Two arguments, the angle 'ang' and type 'dribbleT',
   are supplied to this skill which respectively denote the global direction
   towards which the agent wants to dribble and the kind of dribble that must
   be performed. We distinguish three kinds of dribbling:
   - DRIBBLE FAST: a fast dribble action in which the agent kicks the ball
           relatively far ahead of him.
   - DRIBBLE SLOW: a slower dribble action in which the agent keeps the ball
           closer than on a fast dribble.
   - DRIBBLE WITH BALL: a safe dribble action in which the agent keeps the
           ball very close to his body.

   It is important to realize that this skill is only called when the
   ball is located within the agent's kickable range. This means that
   it is only responsible for the kicking part of the overall
   dribbling behavior, i.e.  it only causes the ball to be kicked a
   certain distance ahead into the desired direction 'ang'. If the
   absolute angle between 'ang' and the agent's body direction is
   larger than DribbleTurnAngle (which currently has a value of 30
   degrees) then the agent uses the turnWithBallTo skill to turn with
   the ball towards the global angle 'ang'. Otherwise, he uses the
   kickTo skill to kick the ball into the desired direction towards a
   point that lies a certain distance ahead depending on the type of
   dribble. After the kick, the ball will move out of the agent's
   kickable range and as a result the agent will try to intercept it
   using the intercept skill. The dribbling skill can then be called
   again once the agent has succeeded in intercepting the ball.  This
   sequence of kicking and intercepting will repeat itself until the
   agent decides to perform another skill. Note that during the
   dribble the power of a kick depends on the distance that the ball
   should travel and on the speed that it should have when it reaches
   the target point.  In our current implementation this speed equals
   0.5 (=DribbleKickEndSpeed) for any type of dribble. Experiments
   have shown that lower end speed values cause the agent to
   intercept the ball before it reaches the target point which slows
   the dribble down significantly.

   \param ang global direction in which should be dribbled
   \param dribbleT type of dribble that should be performed
   \return SoccerCommand to dribble in direction 'ang' */
SoccerCommand BasicPlayer::dribble( AngDeg ang, DribbleT dribbleT )
{
  double       dLength;
  AngDeg       angBody  = WM->getAgentGlobalBodyAngle();
  VecPosition  posAgent = WM->getAgentGlobalPosition();
  SoccerCommand soc;
```

```
  // if not turned into correct direction, turn with the ball to that angle
  AngDeg angDiff = VecPosition::normalizeAngle( ang - angBody );
  if( fabs( angDiff ) > PS->getDribbleAngThr() )
    return turnWithBallTo( ang, PS->getTurnWithBallAngThr(),
                           PS->getTurnWithBallFreezeThr() );

  switch( dribbleT )
  {
    case DRIBBLE_WITHBALL:
      dLength = 4.0;
      break;
    case DRIBBLE_SLOW:
      dLength = 5.0;
      break;
    case DRIBBLE_FAST:
      dLength = 10.0;
      break;
    default:
      dLength = 0.0;
      break;
  }

  // determine shooting point, relative to agent since moving in that dir.
  VecPosition posDribble = posAgent + VecPosition( dLength, angBody, POLAR );

  // adjust when point lies outside side of field
  // to a point that lies at distance 2.0 from the side of the field
  if( fabs( posDribble.getY() ) > PITCH_WIDTH/2.0  - 3.0 )
    posDribble.setY( (PITCH_WIDTH/2.0  - 3.0)*sign(posDribble.getY()) );
  if( fabs( posDribble.getX() ) > PITCH_LENGTH/2.0 - 3.0 )
    posDribble.setX( (PITCH_LENGTH/2.0  - 3.0)*sign(posDribble.getX()) );

  soc = kickTo( posDribble, 0.5 );

  // if ball is kickable but already heading in the right direction, so only
  // small correction term is necessary, start intercepting
  SoccerCommand soc2      = intercept( false );
  if( soc.dPower < 7  && WM->isCollisionAfterCommand( soc2 ) == false )
  {
    Log.log( 560, "wanted to dribble, but only small kick %f", soc.dPower );
    return soc2;
  }
  return soc;
}

/*! This skill enables an agent to pass the ball directly to another player. It
```

```
    receives two arguments, 'pos' and 'passType', which respectively denote the
    position (of usually a teammate) to which the agent wants to pass the ball
    and the kind of pass (either normal or fast) that should be given. This
    skill uses the kickTo skill to pass the ball to the specified position
    with a certain desired end speed depending on the type of pass.
    \param pos position of object to which a direct pass should be given
    \param passType kind of pass (either PASS_NORMAL or PASS_FAST )
    \return SoccerCommand to perform a direct pass to object 'o' */
SoccerCommand BasicPlayer::directPass( VecPosition pos, PassT passType)
{
 if( passType == PASS_NORMAL )
   return kickTo( pos, PS->getPassEndSpeed() );
 else if( passType == PASS_FAST )
   return kickTo( pos, PS->getFastPassEndSpeed() );
 else
   return SoccerCommand( CMD_ILLEGAL );
}

/*! This skill enables an agent to give a leading pass to another player. A
    leading pass is a pass into open space that `leads' the receiver, i.e.
    instead of passing the ball di:rectly to another player it is kicked just
    ahead of him. In this way the receiver is able to intercept the ball while
    moving in a forward direction and this will speed up the attack. This skill
    receives two arguments, an object o (usually a teammate) and dist, which
    respectively denote the intended receiver of the leading pass and the
    `leading distance' ahead of the receiver. It uses the kickTo skill to pass
    the ball to a point that lies dist in front of the current position of o.
    Here `in front of' means in positive x-direction, i.e. at a global angle of
    0 degrees. Note that the desired end speed for a leading pass is always
    equal to PassEndSpeed (currently 1.4) since the leading aspect of the pass
    might cause the receiver to miss the ball when its speed is higher.
    \param o object to which a leading pass should be given
    \param dDist distance in front of o to which is passed
    \return SoccerCommand to perform a leading pass to object 'o' */
SoccerCommand BasicPlayer::leadingPass( ObjectT o, double dDist, DirectionT dir)
{
 VecPosition posShoot = WM->getGlobalPosition( o );
 if( dir != DIR_ILLEGAL && dir != DIR_CENTER )
   posShoot+=VecPosition(dDist,SoccerTypes::getAngleFromDirection(dir),POLAR);
 return kickTo( posShoot, PS->getPassEndSpeed() );
}

/*! This skill enables an agent to give a more advanced type of pass
    called a through pass. With a through pass the ball is not passed
    directly to another player or just ahead of him, but it is kicked
    into open space between the opponent defenders and the opponent
```

```
   goalkeeper in such a way that a teammate (usually an attacker)
   will be able to reach the ball before an opponent does. If a
   through pass is executed successfully it often causes a
   disorganization of the opponent's defense which will enable an
   attacker to get the ball close to the enemy goal. This skill takes
   an object o (usually a teammate) as an argument which denotes the
   intended receiver of the through pass. The position p on the field
   to which the ball should be kicked is determined by drawing a line
   l from the object's current position to a given point 'pos' also
   supplied as an argument) and by computing the safest trajectory
   for the ball to a point on this line. To this end the widest angle
   between opponents from the current ball position to a point p on l
   is calculated. After this, the speed that the ball should have
   when it reaches this point is determined based on the distance
   from the current ball position to p and on the number of cycles n
   that o will need to reach p. If it turns out that the required end
   speed falls outside the range [MinPassEndSpeed .. MaxPassEndSpeed]
   it is set to the closest boundary of this range. The kickTo skill
   is then used to kick the ball to the desired point p at the
   required speed.

   \param o Object to which through pass should be given
   \param posEnd position that together with ball position defines line
          segment on which shooting point should be determined.
   \param *angMax will be filled with the largest angle between the opponents
   \return SoccerCommand to give a throughPass to 'o' */
SoccerCommand BasicPlayer::throughPass( ObjectT o, VecPosition posEnd,
                                        AngDeg *angMax )
{
  VecPosition posShoot = getThroughPassShootingPoint( o, posEnd, angMax );
  double      dEnd     = getEndSpeedForPass( o, posShoot );

  return kickTo( posShoot, dEnd );
}

/*! This skill enables an agent to outplay an opponent. It is used, for
   example, when an attacker wants to get past an enemy defender. This is
   done by passing the ball to open space behind the defender in such a
   way that the attacker can beat the defender to the ball. Note that the
   attacker has an advantage in this situation, since he knows to which
   point he is passing the ball and is already turned in the right
   direction, whereas the defender is not. As result, the attacker has a
   headstart over the defender when trying to intercept the ball. Since a
   player can move faster without the ball, the main objective is to kick
   the ball as far as possible past the opponent while still being able to
   reach it before the opponent does. This skill receives two arguments,
```

'pos' and 'o', which respectively denote the point to which the agent wants to move with the ball and the object (usually an opponent) that the agent wants to outplay in doing so. First it is determined if it is possible to outplay the opponent o in the current situation. Let l be the line segment that runs from the agent's current position to the given point 'pos'. The best point to kick the ball to will be the furthest point from on this line that the agent can reach before the opponent does. We use a simple geometric calculation to find the point s on l which has equal distance to the agent and to the opponent. Let o' be the perpendicular projection of the opponent's position onto l and let d1, d2 and d3 respectively denote the distance between the agent poisition and o', the distance between o' and o and the distance between o' and s. To determine s we need to compute the unknown value of d3 using the values for d1 and d2 which can be derived from the world model. Since the distance from the agent position to s will be equal to the distance from o to s. Using this value for d3 we can compute the coordinates of the shooting point s. However, in some situations it is not likely that shooting the ball to this point will eventually result in outplaying the given opponent o on the way to 'pos'. We therefore use the values for d1, d2 and d3 to determine whether it is possible to outplay o in the current situation, and if so, what the best shooting point will be. The following situations are distinguished:

- d1 + d3 > OutplayMinDist.

If this condition holds, the opponent is located at a relatively large distance from the agent which makes an attempt to outplay him likely to be successful. First it is checked whether the agent's body is turned suffiiently towards the point 'pos'. If this is not the case then the turnWithBallTo skill is used to turn with the ball in the right direction. Otherwise, the kickTo skill is used to kick the ball to a point on the line l where the agent will be able to intercept it first. Note that in general the agent will be able to reach the point s before the opponent despite the fact that both players need to travel the same distance to this point. This is because the agent has already turned his body more or less towards 'pos' (and thus towards s), whereas the opponent probably has not. However, the actual point to which the ball is kicked is chosen slightly closer to the agent than the point s in order to be absolutely sure that he can intercept the ball before the opponent does. This safety margin is represented by the parameter OutplayBuffer which has a value of 2.5 in our current implementation. For this skill the desired end speed when the ball reaches z equals OutplayKickEndSpeed (=0.5). Note that the value for this parameter cannot be chosen too low, since this will cause the agent to intercept the ball before it reaches the target point

- d1 + d3 < OutplayMinDist and d1 < d2.

If this condition holds, the opponent is located close to the agent which makes it diÂ±cult to outplay him. However, if the agent is already

```
   turned in the right direction (i.e. towards 'pos') then d1 < d2 implies
   that the distance between the opponent and the line l (denoting the
   agent's desired movement trajectory) is large enough for the agent to
   outplay this opponent when the ball is kicked hard in the direction of
   'pos' (i.e. further ahead than s). This is because the agent can start
   dashing after the ball immediately, whereas the opponent still has to
   turn in the right direction. As a result, the agent will have dashed
   past the opponent by the time the latter has turned correctly and this
   puts him in a good position to intercept the ball before the opponent.
   Therefore it is checked first whether the agent's body is sufficiently
   turned towards 'pos' and if this is not so then the turnWithBallTo skill
   is used toturn with the ball in the right direction. Otherwise, the
   kickTo skill is used to kick the ball past the opponent. In this case the
   point to which the ball is kicked either lies OutplayMaxDist (=20) metres
   ahead of the agent's current position into the direction of 'pos' or
   equals 'pos' when the distance to 'pos' is smaller than this value. In
   all other cases (i.e. d1 +d3 < OutplayMinDist and d1 > d2) this
   skill returns an illegal command to indicate that it is not
   possible to outplay the opponent o on the way to the point 'pos'.
   \param o opponent object that should be outplayed
   \param pos position to which agent wants to move while outplaying 'o'
   \param *posTo position to which ball will be shot
   \return SoccerCommand to outplay object o on the way to 'pos',
     CMD_ILLEGAL when this is not possible. */
SoccerCommand BasicPlayer::outplayOpponent( ObjectT o, VecPosition pos,
                                VecPosition *posTo )
{
 // future: take more than one opponent into account

 VecPosition posAgent   = WM->getAgentGlobalPosition();
 AngDeg      angBody    = WM->getAgentGlobalBodyAngle();

 double dMaxDist = 12.0;
 if( posAgent.getX() > PENALTY_X - 6.0 )
  dMaxDist = 10.0;

 AngDeg      ang        = (pos - posAgent).getDirection();
 // if not heading in the desired direction, first turn with the ball
 AngDeg angTmp = VecPosition::normalizeAngle( ang - angBody ) ;
 if( fabs( angTmp ) > PS->getDribbleAngThr() )
  return turnWithBallTo( ang, PS->getTurnWithBallAngThr(),
                         PS->getTurnWithBallFreezeThr() );
 ang = WM->getAgentGlobalBodyAngle();
 Line        l          = Line::makeLineFromPositionAndAngle(posAgent,ang);
 VecPosition posObj     = WM->getGlobalPosition( o );
 VecPosition posProj    = l.getPointOnLineClosestTo( posObj );
```

```
double      dDistOpp   = posProj.getDistanceTo( posObj );
double      dDistAgent = posProj.getDistanceTo( posAgent );
VecPosition posShoot;

// we want to know when distance from ball to point p equals distance
// from opp to point p :
// d1 + d3 = sqrt(d2^2 + d3^2) > (d1+d3)^2 = d2^2 + d3^2 =>
// d1^2 + 2*d1*d3 = d2^2 -> d3 = (d2^2 - d1^2 ) / 2*d1
double dCalcDist;
if( o != OBJECT_ILLEGAL )
{
  dCalcDist = (dDistOpp*dDistOpp-dDistAgent*dDistAgent)/(2*dDistAgent);
  dCalcDist += dDistAgent;
}
else
  dCalcDist = dMaxDist;

Log.log( 552, "outplay opponent %d, calc: %f, opp: %f, agent:  %f",
   SoccerTypes::getIndex( o ) + 1, dCalcDist, dDistOpp, dDistAgent );
Log.log( 560, "outplay opponent %d, calc: %f, opp: %f, agent:  %f",
   SoccerTypes::getIndex( o ) + 1, dCalcDist, dDistOpp, dDistAgent );

if( dCalcDist > 7.0 ) // if point far away, kick there
{
  dCalcDist = min( posAgent.getDistanceTo( pos ), dCalcDist - 2.5 );
  dCalcDist = min( dMaxDist, dCalcDist );
  posShoot  = posAgent + VecPosition( dCalcDist, ang, POLAR );
}
else if( dDistAgent < dDistOpp - 0.3 ) // point close and well-positioned
{                                      // shoot far away and outplay opp
  dCalcDist = min( posAgent.getDistanceTo( pos ), dMaxDist );
  posShoot = posAgent + VecPosition( dCalcDist, ang, POLAR );
}
else                                   // opponent stands in line
  return SoccerCommand( CMD_ILLEGAL );

if( posShoot.getDistanceTo( WM->getAgentGlobalPosition() ) < 2.5 )
{
  Log.log( 552, "calculated point too close" );
  Log.log( 560, "calculated point too close (%f,%f)", posShoot.getX(),
    posShoot.getY() );
  return SoccerCommand( CMD_ILLEGAL );
}
else if( WM->getNrInSetInCone( OBJECT_SET_OPPONENTS,PS->getConeWidth(),
  posAgent, posShoot ) != 0 )
{
```

```
    Log.log( 552, "outplay: is opponent in cone" );
    Log.log( 560, "outplay: is opponent in cone (%f,%f)", posShoot.getX(),
     posShoot.getY() );
    return SoccerCommand( CMD_ILLEGAL );
   }
   else if( WM->getNrInSetInCircle( OBJECT_SET_OPPONENTS,
            Circle( posShoot, posShoot.getDistanceTo( posAgent ) ) ) > 1 )
   {
    Log.log( 552, "outplay: nr of opp in circle > 1" );
    Log.log( 560, "outplay: nr of opp in circle > 1" );
    return SoccerCommand( CMD_ILLEGAL );
   }
   else if( WM->getCurrentTime() - WM->getTimeGlobalAngles(o) < 3 )
   {
    double dDistOpp = posShoot.getDistanceTo( posObj ) ;
    if( fabs( VecPosition::normalizeAngle( (posShoot - posObj).getDirection() -
                                                                    WM-
>getGlobalBodyAngle(o) ) ) < 30 )
      dDistOpp -= 1.0;
    if( dDistOpp < posAgent.getDistanceTo( posShoot ) )
    {
     Log.log( 560, "outplay: opponent closer" );
     return SoccerCommand( CMD_ILLEGAL );
    }
   }



  if( posTo != NULL )
    *posTo = posShoot;
  return kickTo( posShoot, 0.5 );
}

/*! This skill enables an agent to clear the ball to a certain area on the
    field. It is useful, for example, when a defender cannot dribble or pass
    the ball to a teammate in a dangerous situation. Using this skill he can
    then kick the ball up the field away from the defensive zone. It is
    important to realize that this skill is only called when the agent has no
    alternative options in the current situation. Clearing the ball amounts to
    kicking it with maximum power into the widest angle between opponents in
    a certain area. The shooting direction is determined using the function
    which returns the direction of the bisector of this widest angle. The area
    on the field from which this angle is selected depends on the type of
    clear which is supplied as an argument to this skill. We distinguish
    three types of clearing:
    - CLEAR BALL DEFENSIVE: clearing the ball away from the defensive zone
```

```
      into a triangular area which is defined by the current ball position
      and the center line on the field.
    - CLEAR BALL OFFENSIVE: clearing the ball towards the offensive zone into
      a triangular area which is defined by the current ball position and the
      line segment that coincides with the front line of the opponent's
      penalty area and extends to the left and right side lines.
    - CLEAR BALL GOAL: clearing the ball into a triangular area in front of the
      opponent's goal which is defined by the current ball position and the
      line segment that runs from the center of the opponent's goal to the
      center of the front line of the penalty area.
    \param type type of the clear ball skill
    \param s if specified indicates the part of the field the clearBall should
          be directed to.
    \param angMax if specified (and not NULL) will be filled with the angle
          between the opponents in the direction in which will be shot
    \return SoccerCommand kick command to clear the ball */
SoccerCommand BasicPlayer::clearBall( ClearBallT type, SideT s,AngDeg *angMax )
{
  VecPosition posBall = WM->getBallPos();
  VecPosition posLeft, posRight;
  double      clearDist = PS->getClearBallDist();

  double      dPitchY = PITCH_WIDTH / 2.0;
  if( type == CLEAR_BALL_DEFENSIVE )
  {
    posLeft.setVecPosition ( 0, - dPitchY + clearDist );
    posRight.setVecPosition( 0, + dPitchY - clearDist );
  }
  else if( type == CLEAR_BALL_OFFENSIVE )
  {
    posLeft.setVecPosition ( PENALTY_X - clearDist, - dPitchY + clearDist );
    posRight.setVecPosition( PENALTY_X - clearDist, + dPitchY - clearDist );
  }
  else if( type == CLEAR_BALL_OFFENSIVE_SIDE )
  {
    posLeft.setVecPosition ( PENALTY_X - clearDist, - dPitchY + 8 );
    posRight.setVecPosition( PENALTY_X - clearDist, + dPitchY - 8 );
  }
  else if( type == CLEAR_BALL_GOAL && posBall.getY() > 0 )
  {
    posLeft.setVecPosition ( max(PENALTY_X - 2.0, posBall.getX()-10),  0.0 );
    posRight.setVecPosition( PITCH_LENGTH/2.0 - 5.0, 0.0 );
  }
  else if( type == CLEAR_BALL_GOAL && posBall.getY() < 0 )
  {
    posLeft.setVecPosition ( PITCH_LENGTH/2.0 - 5.0, 0.0 );
```

```
  posRight.setVecPosition( max(PENALTY_X - 2.0, posBall.getX()-10),  0.0 );
}
else
  return SoccerCommand( CMD_ILLEGAL );

if( type != CLEAR_BALL_GOAL && s == SIDE_RIGHT ) // take only right part of
{
  if( type == CLEAR_BALL_OFFENSIVE_SIDE )
    posLeft.setY( 15.0 );
  else
    posLeft.setY ( 0.0 );                    // field into account

}
else if( type != CLEAR_BALL_GOAL && s == SIDE_LEFT )
{
  if( type == CLEAR_BALL_OFFENSIVE_SIDE )
    posRight.setY( -15.0 );
  else
    posRight.setY( 0.0 );
}

// get angle of ball with left and right points
// get the largest angle between these two angles
AngDeg angLeft  = (posLeft  - posBall).getDirection();
AngDeg angRight = (posRight - posBall).getDirection();
double dDist;
if( type != CLEAR_BALL_GOAL )
  dDist = PS->getClearBallOppMaxDist();
else
  dDist = max( posBall.getDistanceTo(posLeft ),
               posBall.getDistanceTo(posRight) );
AngDeg ang      = WM->getDirectionOfWidestAngle( posBall, angLeft, angRight,
                                 angMax, dDist );

Line l1 = Line::makeLineFromPositionAndAngle( posBall, ang );
Line l2 = Line::makeLineFromTwoPoints( posLeft, posRight );
VecPosition posShoot = l1.getIntersection( l2 );
Log.log( 560, "angLeft %f, right %f, best %f point (%f,%f)",
angLeft, angRight, ang, posShoot.getX(), posShoot.getY() );
if( type == CLEAR_BALL_GOAL  )
  return kickTo( posShoot, SS->getBallSpeedMax() );
else if( type == CLEAR_BALL_OFFENSIVE )
  return kickTo( posShoot, 0.25 );
else if( type == CLEAR_BALL_OFFENSIVE_SIDE )
{
  return kickTo( posShoot, (posBall.getX()>20) ? 1.2 : 2.7);
```

```
  }
  else
    return kickTo( posShoot, 0.5 );
}

/*! This skill enables an agent to mark an opponent, i.e. to guard him
    one-on-one with the purpose to minimize his usefulness for the opponent
    team. It can be used, for example, to block the path from the ball to
    an opponent or from an opponent to the goal. In this way the agent can
    prevent this opponent from receiving a pass or from moving closer to the
    goal while also obstructing a possible shot. This skill amounts to
    calculating the desired marking position based on the given arguments
    and then moving to this position. It receives three arguments: an
    object o (usually an opponent) that the agent wants to mark, a distance
    'dDist' representing the desired distance between o and the marking
    position and a type indicator that denotes the type of marking that is
    required. We distinguish three types of marking:
    - MARK BALL: marking the opponent by standing at a distance 'dDist' away
                 from him on the line between him and the ball. This type of
                 marking will make it diÂ±cult for the opponent to receive a
                 pass.
    - MARK GOAL: marking the opponent by standing at a distance 'dDist' away
                 from him on the line between him and the center point of the
                 goal he attacks. This type of marking will make it difficult
                 for the opponent to score a goal.
    - MARK BISECTOR: marking the opponent by standing at a distance 'dDist'
                 away from him on the bisector of the ball-opponent-goal
                 angle. This type of marking enables the agent to intercept
                 both a direct and a leading pass to the opponent.

    After determining the marking position, the agent uses the moveToPos skill
    to move to this position. Note that the decision whether to turn or dash
    in the current situation depends on the angle of the marking position
    relative to the agent's body direction and on the distance to this position
    if this point lies behind the agent. In this case the moveToPos skill uses
    the threshold parameters MarkTurnAngle (=30) and MarkDistanceBack (=3) to
    make this decision. The values for these parameters are such that the
    condition which must hold for allowing a dash is fairly flexible. This is
    done because the marking position will be different in consecutive cycles
    due to the fact that the opponent and the ball move around from each
    cycle to the next. As a result, the agent will be able to actually
    progress towards a point that lies close to the marking position instead
    of constantly turning towards the newly calculated marking position in
    each cycle.
    \param o object that has to be marked
    \param dDist distance marking position is located from object position
```

```
    \param mark marking technique that should be used
    \return SoccerCommand to mark object 'o'. */
SoccerCommand BasicPlayer::mark( ObjectT o, double dDist, MarkT mark )
{
  VecPosition posMark  = getMarkingPosition( o, dDist, mark );
  VecPosition posAgent = WM->getAgentGlobalPosition();
  VecPosition posBall  = WM->getGlobalPosition( OBJECT_BALL );
//  AngDeg      angBody  = WM->getAgentGlobalBodyAngle();

  if( o == OBJECT_BALL )
  {
    if( posMark.getDistanceTo( posAgent ) < 1.5 )
      return turnBodyToObject( OBJECT_BALL );
    else
     return moveToPos( posMark, 30.0, 3.0, false );
  }

  if( posAgent.getDistanceTo( posMark ) < 2.0 )
  {
    AngDeg angOpp = (WM->getGlobalPosition( o ) - posAgent).getDirection();
    AngDeg angBall = (posBall - posAgent).getDirection();
    if( isAngInInterval( angBall, angOpp,
                         VecPosition::normalizeAngle( angOpp + 180 ) ) )
      angOpp += 80;
    else
      angOpp -= 80;
    angOpp  = VecPosition::normalizeAngle( angOpp );
    Log.log( 513, "mark: turn body to ang %f", angOpp );
    return turnBodyToPoint( posAgent + VecPosition( 1.0, angOpp, POLAR )  );
  }
  Log.log( 513, "move to marking position" );

  return moveToPos( posMark, 25, 3.0, false );
}


/*! This skill enables an agent (usually the goalkeeper) to defend his
    own goal line. To this end the agent moves to a position along a
    line l which runs parallel to the goal line at a small distance
    'dDist' (supplied as an argument) from the goal. The guard point
    to which the agent moves depends on the predicted position of the
    ball in the next cycle and is chosen in anticipation of a future
    shot on goal. This means that the guard point is selected in such
    a way that it will be most difficult for the opponent team to pass
    the goalkeeper. To find this point we need to look at the angle
    that the ball makes with respect to the left and right goal posts
```

```
   and we need to determine which point on l covers this angle in the
   most optimal way, i.e.  leaves an equal gap to the left and to the
   right of the goalkeeper. Let m be the line that goes through the
   center point of the goal line and through the predicted ball
   position in the next cycle. Since this line m bisects the
   above-mentioned angle, the optimal guard point on l can be
   computed by determining the intersection between l and m. Note
   that in our current implementation the goalkeeper always stays in
   front of the goal mouth to avoid leaving an open goal when the
   ball is passed to an opponent in the center of the penalty
   area. The computed guard point is therefore adjusted if it lies
   too far to the side . After computing the guard point the
   goalkeeper needs to move to this point while keeping sight of the
   ball.  If the distance between the current goalkeeper position and
   the line l is larger than DefendGoalLineMaxDist (which has a value
   of 3.0 in our current implementation) the moveToPos skill is used
   to move directly towards the guard point. This can happen, for
   example, when the goalkeeper has moved forward from his line to
   intercept the ball and now has to move back to his line
   again. Note that the fourth argument supplied to the moveToPos
   skill equals true in this case to indicate that the goalkeeper
   wants to turn his back to the guard point in order to keep the
   ball in sight while moving.  However, if the distance between the
   guard point and l is less than DefendGoalLineMaxDist then the
   moveToPosAlongLine skill is used to move along l to the guard
   point. This skill receives an argument `sign' representing a
   prediction of the agent's movement in the coming cycles.  This
   value is used to adjust the agent's body direction when necessary.
   In this case it can be expected that the goalkeeper will move
   along l in the same direction as the ball and `sign' is therefore
   determined by looking at the ball velocity in cycle t.

   \param dDist distance before goal the goalkeeper will move along
   \return SoccerCommand to defend the goal line. */
SoccerCommand BasicPlayer::defendGoalLine( double dDist )
{
  // determine defending point as intersection keeper line and line ball-goal
  VecPosition posBall    = WM->getBallPos();
  VecPosition posAgent   = WM->getAgentGlobalPosition();
  VecPosition posGoal    = WM->getPosOwnGoal( );
  VecPosition posCenter(sign(posGoal.getX())*(fabs(posGoal.getX())-dDist),0);
  Line        lineGoal   = Line::makeLineFromPositionAndAngle( posCenter, 90 );

  VecPosition posGoalLeft ( posGoal.getX(), -SS->getGoalWidth()/2.0 );
  VecPosition posGoalRight( posGoal.getX(),  SS->getGoalWidth()/2.0 );
  Line left    = Line::makeLineFromTwoPoints( posBall, posGoalLeft  );
```

```
Line right   = Line::makeLineFromTwoPoints( posBall, posGoalRight );
posGoalLeft  = left.getIntersection ( lineGoal );
posGoalRight = right.getIntersection( lineGoal );
double dDistLeft  = posGoalLeft.getDistanceTo( posBall );
double dDistRight = posGoalRight.getDistanceTo( posBall );
double dDistLine  = posGoalLeft.getDistanceTo( posGoalRight );
VecPosition posDefend  = posGoalLeft+
   VecPosition( 0, (dDistLeft/(dDistLeft+dDistRight))*dDistLine);

bool       bBallInPen = WM->isInOwnPenaltyArea( posBall );

// do not stand further to side than goalpost
if( fabs( posDefend.getY() ) > SS->getGoalWidth()/2.0 )
  posDefend.setY( sign(posDefend.getY())*SS->getGoalWidth()/2.0);

// if too far away from line, move directly towards it
double dDiff = ( bBallInPen == true ) ? 1.5 : 0.5;
if( posDefend.getX() + dDiff < posAgent.getX()  )
{
  Log.log( 553, "move backwards to guard point" );
  return moveToPos( posDefend, 30, -1.0, true ); // always backwards
}
else if( posDefend.getX() - dDiff > posAgent.getX() )
{
  Log.log( 553, "move forward to guard point" );
  return moveToPos( posDefend, 30, -1.0 );       // always forward
}

// desired body angle is in direction of the ball
// predicted movement direction in subsequent cycles is in moving dir. ball
AngDeg  angDes;
if( fabs( posBall.getY() - posDefend.getY() ) > 0.5 )
  angDes = sign( posBall.getY() - posDefend.getY() )*90.0;
else
  angDes = sign( WM->getAgentGlobalBodyAngle() )*90.0;
int     iSign     = sign( WM->getGlobalVelocity( OBJECT_BALL ).getY() );

// move to position along line: when ball in penalty area, never adjust body
// angle (value 3.0) and change trajectory when angle difference is > 7
// when ball is outside pen. area, adjust angle to move to line when more
// than 0.5 from desired line, adjust when angle diff > 2 with 12 degrees
if( bBallInPen )
{
  Log.log( 553, "move along line, with ball in penalty area x: %f ang %f",
             posDefend.getX(), angDes );
  return moveToPosAlongLine( posDefend, angDes, 3.0, iSign, 7.0, 12.0 );
```

```
  }
  else
  {
    Log.log( 553, "move along line, with ball out pen. area (%f,%f)(%f,%f) %f",
      WM->getAgentGlobalPosition().getX(),
      WM->getAgentGlobalPosition().getY(),
      posBall.getX(), posBall.getY(), WM->getConfidence( OBJECT_BALL) );
    Log.log( 553, "%s", WM->strLastSeeMessage );
    return moveToPosAlongLine( posDefend, angDes, 0.5, iSign, 2.0, 12.0 );
  }
}

/*! This method returns a command to intercept a ball that is currently
    heading towards the goal. The current trajectory of the ball is determined
    and the goalkeeper positions himself on a point on this trajectory just
    before the goal.
    \return SoccerCommand to intercept a ball heading towards the goal. */
SoccerCommand BasicPlayer::interceptScoringAttempt( )
{
  int iSide = 1;
  if( WM->isPenaltyUs() || WM->isPenaltyThem() )
    iSide = ( WM->getSide() == WM->getSidePenalty() ) ? 1 : -1;

  SoccerCommand soc;
  VecPosition   posAgent = WM->getAgentGlobalPosition();
  VecPosition   posBall  = WM->getBallPos();
  Line          lineBall = Line::makeLineFromPositionAndAngle( posBall,
                                WM->getBallDirection() );
  Line          lineKeeper = Line::makeLineFromPositionAndAngle( posAgent, 90);
  bool          bInterceptAtSide = false;

  // first create intersection point between ball trajectory and rectangle
  // that is located just in front of the goal mouth
  VecPosition   posIntersect = lineBall.getIntersection( lineKeeper );
  if( fabs( posIntersect.getY() ) > SS->getGoalWidth()/2.0  )
  {
    VecPosition posGoalPost( -iSide*PITCH_LENGTH/2.0,
                    sign(posBall.getY())*(SS->getGoalWidth()/2.0 + 0.5));
    Line l = Line::makeLineFromPositionAndAngle( posGoalPost, 0 );
    posIntersect = lineBall.getIntersection( l );

    // if intersection point does not cross rectangle between the agent and
    // the goalpost, we just move to edge of the rectangle
    if( fabs(posIntersect.getX()) > fabs(posGoalPost.getX())  ||
        fabs(posIntersect.getX()) < fabs(posAgent.getX()) )
      posIntersect.setVecPosition( posAgent.getX(), posGoalPost.getY() );
```

```
   else
     bInterceptAtSide = true; // interception point is at side of rectangle
}

// first try close intercept
soc = interceptCloseGoalie();
if( ! soc.isIllegal() )
{
  Log.log( 553, "close intercept" );
  return soc;
}

// if far away from goal, just intercept ball
if( PITCH_LENGTH/2.0 - fabs(posAgent.getX()) > 8.0 )
{
  Log.log( 553, "I am far away from keeper line: %d %d %f %f",
  WM->getSide(), WM->getSidePenalty(), -iSide*PITCH_LENGTH/2.0 ,
  posAgent.getX() );
  return intercept( true );
}
else if( fabs(posBall.getX()) > fabs(posAgent.getX()) &&
         fabs( posBall.getY() ) < SS->getGoalWidth()/2.0 + 2.0 )
{
  Log.log( 553, "ball heading and ball behind me" );
  return intercept( true );
}

// move to interception point
if( posIntersect.getDistanceTo( posAgent ) < 0.5 )
{
   Log.log( 553, "close to intersection point keeperline" );
   soc = turnBodyToObject( OBJECT_BALL );
}
else if( sign( (posIntersect - posAgent ).getDirection() ) ==
         sign( WM->getAgentGlobalBodyAngle()             ) ||
     bInterceptAtSide == true)
{
  Log.log( 553, "move forward to intersection point keeperline" );
  soc = moveToPos( posIntersect, 20, SS->getGoalWidth() );
}
else
{
  Log.log( 553, "move backward to intersection point keeperline" );
  soc = moveToPos( posIntersect, 20, SS->getGoalWidth(), true );
}
```

```
  return soc;
}

/*! This method returns a command to hold the ball close to your body. When no
    opponents are close the ball is kicked in front of the body of the agent.
    Otherwise it is kiced to that spot in the kickable distance which is
    hardest for the opponent to tackle. */
SoccerCommand BasicPlayer::holdBall( )
{
  double        dDist;
  VecPosition   posAgent = WM->getAgentGlobalPosition();
  ObjectT       objOpp   = WM->getClosestInSetTo( OBJECT_SET_OPPONENTS,

                                                  OBJECT_BALL, &dDist );
  VecPosition   posOpp   = WM->getGlobalPosition ( objOpp );
  AngDeg        angOpp   = WM->getGlobalBodyAngle( objOpp );
  AngDeg        ang      = 0.0;

  if( objOpp != OBJECT_ILLEGAL && dDist < 5 )
  {
    // get the angle between the object to the agent
    // check whether object is headed to the left or right of this line
    ang = ( posAgent - posOpp ).getDirection();
    int iSign = -sign(VecPosition::normalizeAngle( angOpp - ang ));
    ang +=  iSign*45 - WM->getAgentGlobalBodyAngle();
    ang = VecPosition::normalizeAngle( ang );
    Log.log( 512, "hold ball: opp close shoot to ang %f", ang );
  }
  else
    Log.log( 512, "hold ball: in direction body: %f", ang );

  if( WM->getBallPos().getDistanceTo( posAgent + VecPosition( 0.7, ang, POLAR ) )
      < 0.3 )
  {
    SoccerCommand soc          = turnBodyToPoint( WM->getPosOpponentGoal() );
    VecPosition   posBallPred = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
    VecPosition   posPred     = WM->predictAgentPosAfterCommand( soc );
    if( posPred.getDistanceTo( posBallPred ) < 0.85 * SS->getMaximalKickDist() )
    {
      Log.log( 512, "hold ball: turn body to goal, ball remains still" );
      return soc;
    }
  }

  return kickBallCloseToBody( ang, 0.7 ) ;
}
```

```
/********************** UTILITY METHODS
**************************************/

/*! This method returns the shooting point for a through pass to objTeam.
    This point lies on the line l that is created of the estimated position of
    the teammate after 3 cycles and 'posEnd'. Between these two points the
    maximal angle with respect to the opponents is calculated and the widest
    angle is calculated (this value is stored in angMax afterwards). The
    intersection between this widest angle and the line l is returned as the
    shooting point.
    \param objTeam teammate to give a throughpass to
    \param posEnd end point to which throughpasses are considered
    \param angMax will be filled with the maximal angle between the opponents
    \return position to give a through pass to. */
VecPosition BasicPlayer::getThroughPassShootingPoint( ObjectT objTeam,
     VecPosition posEnd, AngDeg  *angMax  )
{
  VecPosition posTeam = WM->getGlobalPosition( objTeam );
  return getShootPositionOnLine( posTeam, posEnd, angMax );
}

/*! This method uses an iterative scheme to compute the optimal
    interception point of the ball. A loop is executed in which the
    prediction methods are used to predict the position of the ball a
    number of cycles, say i, into the future and to predict the number
    of cycles, say n, that the agent will need to reach this
    position. This is repeated for increasing values of i until n < i
    in which case it is assumed thay the agent should be able to reach
    the predicted ball position before the ball does.  This point is
    chosen as the interception point and the moveToPos skill is used
    to move towards this point.

    \param iCyclesToBall will be filled with the nr of cycles it will take the
                         ball to reach the returned position.
    \param isGoalie indicates whether the current player is a goalkeeper or not
    \return position to intercept the ball. */
VecPosition BasicPlayer::getInterceptionPointBall( int *iMinCyclesBall,
                                        bool isGoalie )
{
  static Time timeLastMinCycles(-1,0);
  static Time timeLastIntercepted(-1,0);
  VecPosition posPred  = WM->getAgentGlobalPosition();
  VecPosition velMe    = WM->getAgentGlobalVelocity();
  double      dSpeed, dDistExtra;
  VecPosition posMe, posBall;
```

```
AngDeg      ang, angBody, angNeck;
Stamina     sta;
double      dMaxDist, dBestX = -100;
int         iCyclesBall, iCyclesOpp, iFirstBall = -100;//, iCyclesOppPoint;


if( (WM->getCurrentTime() - timeLastIntercepted) > 2 )
  timeLastMinCycles.updateTime( -1 );
timeLastIntercepted = WM->getCurrentTime();
*iMinCyclesBall = 100;

if( isGoalie )
  return getActiveInterceptionPointBall( iMinCyclesBall, isGoalie );

dMaxDist = (isGoalie) ? SS->getCatchableAreaL() : SS->getMaximalKickDist();

// predict the position of the agent when current velocity is propogated
dSpeed     = WM->getAgentSpeed();
dDistExtra = Geometry::getSumInfGeomSeries( dSpeed, SS->getPlayerDecay() );
posPred   += VecPosition( dDistExtra, velMe.getDirection(), POLAR );

ObjectT objOpp = WM->getFastestInSetTo(OBJECT_SET_OPPONENTS,
                                OBJECT_BALL, &iCyclesOpp);
Log.log( 501, "closest opponent can get to ball in %d cycles", iCyclesOpp );
if( WM->isBeforeGoal( WM->getAgentGlobalPosition() ) )
  iCyclesOpp = 20;
int i = 2; // otherwise should be done by interceptClose
// for each loop, check whether agent can reach ball in less cycles
// do not look further than cycles that fastest opponent can reach the ball
while( i <= PS->getPlayerWhenToIntercept() && i <= iCyclesOpp + 2 )
{
  posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, i );

  // if too far away, try next cycle
  if( posPred.getDistanceTo(posBall)/SS->getPlayerSpeedMax() > i + dMaxDist
      ||
      WM->isInField( posBall ) == false )
  {
    i++;
    continue;
  }

  // (re-)initialize all the needed variables
  // set ball prediction one further to get right in front of ball line
  posMe   = WM->getAgentGlobalPosition();
  velMe   = WM->getAgentGlobalVelocity();
```

```
angBody = WM->getAgentGlobalBodyAngle();
angNeck = WM->getAgentGlobalNeckAngle();
ang     = (posBall - posPred).getDirection();
ang     = VecPosition::normalizeAngle( ang - angBody );
sta     = WM->getAgentStamina();
int turn = 0;

// as long as not correctly headed for point, simulate a turn command
while (fabs(ang) > PS->getPlayerWhenToTurnAngle() && turn<5)
{
  turn++;
  WM->predictStateAfterTurn( WM->getAngleForTurn(ang,velMe.getMagnitude()),
              &posMe, &velMe, &angBody, &angNeck,WM->getAgentObjectType(),
              &sta             );
  ang      = (posBall - posPred).getDirection();
  ang      = VecPosition::normalizeAngle( ang - angBody );
}

if( turn > 1 )
  Log.log( 501, "nr of turns needed: %d", turn );
int iTurnTmp = turn;
// for cycles that are left over after turn(s), execute full power dash
for( ; turn < i; turn++ )
  WM->predictStateAfterDash(SS->getMaxPower(),&posMe,&velMe,&sta,angBody);

// if in kickable distance or passed ball, we can reach the ball!
if (posMe.getDistanceTo( posBall ) < dMaxDist  ||
   (posMe.getDistanceTo( posPred ) > posBall.getDistanceTo( posPred ) -
                              dMaxDist) )
{
  Log.log( 501, "can intercept in %d cycles %d turns, ball %f best %f",
      i, iTurnTmp, posBall.getX(), dBestX );

  if( *iMinCyclesBall == 100 ) // log first possible interception point
  {
    *iMinCyclesBall = i;
    iFirstBall = i;
  }

  iCyclesBall = i;
  if( WM->getCurrentTime() + iCyclesBall == timeLastMinCycles )
  {
    Log.log( 501, "choose old interception point %d", iCyclesBall );
    return posBall;
  }
  if( objOpp == OBJECT_ILLEGAL || isGoalie == true )
```

```
      {
       // if no opponent, move to first reachable point
       if( *iMinCyclesBall == 100 )
       {
         *iMinCyclesBall = iCyclesBall;
         i = PS->getPlayerWhenToIntercept() + 1; // and quit
       }
      }
      else if( WM->isBeforeGoal( WM->getAgentGlobalPosition() ) )
      {
       if( fabs(posBall.getY()) < fabs(dBestX ) &&
           iCyclesBall < iFirstBall + 3   &&
           (iCyclesOpp - iCyclesBall)  >= 2  )
       {
         dBestX = fabs(posBall.getY());
         *iMinCyclesBall = iCyclesBall;
         Log.log( 501, "update best cycles in goalarea to %d, opp %d",
                                                                    *iMinCyclesBall,
(iCyclesOpp - iCyclesBall) );
       }

      }
      else if( posBall.getX() >= dBestX &&
              fabs( posBall.getY() ) < 32.0 &&
                                                                    fabs( posBall.getX() )
< 50.0 &&
              ( posBall.getX() > - PITCH_LENGTH/4.0 ||
                iCyclesBall < iFirstBall + 3 ) &&
                                                                    (iCyclesOpp -
iCyclesBall)  >= 1 )
      {
       dBestX = posBall.getX();
       *iMinCyclesBall = iCyclesBall;
       Log.log( 501, "update best cycles to %d, opp diff %d", *iMinCyclesBall,
                                                                  iCyclesOpp - iCyclesBall
);
      }
     }
     i++;
   }

   if( i == 2 || ( i == iCyclesOpp + 3 && *iMinCyclesBall > iCyclesOpp ))
   {
     Log.log( 501, "move to interception point closest opp." );
     *iMinCyclesBall = iCyclesOpp;
   }
```

```
  posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, *iMinCyclesBall );
  Log.log( 501, "choose: %d", *iMinCyclesBall );
  timeLastMinCycles = WM->getCurrentTime() + *iMinCyclesBall;

  return posBall;
}


/*! This method intercepts the ball at the first possible position.
    \param iCyclesBall is the nr of cyles after the ball is intercepted
    \param isGoalie bool to indicate that a goalie has to intercept the ball
    \return intercept position */
VecPosition BasicPlayer::getActiveInterceptionPointBall( int *iCyclesBall,
                                         bool isGoalie )
{
  VecPosition posPred  = WM->getAgentGlobalPosition();
  VecPosition velMe    = WM->getAgentGlobalVelocity();
  double      dSpeed, dDistExtra;
  VecPosition posMe, posBall;
  AngDeg      ang, angBody, angNeck;
  Stamina     sta;
  double      dMaxDist;

  if( isGoalie &&
      !WM->isInOwnPenaltyArea(WM->predictPosAfterNrCycles( OBJECT_BALL, 45)))
  {
    *iCyclesBall = -1;
    return posBall;
  }

  dMaxDist = (isGoalie) ? SS->getCatchableAreaL() : SS->getMaximalKickDist();

  // predict the position of the agent when current velocity is propogated
  dSpeed     = WM->getAgentSpeed();
  dDistExtra = Geometry::getSumInfGeomSeries( dSpeed, SS->getPlayerDecay() );
  posPred   += VecPosition( dDistExtra, velMe.getDirection(), POLAR );

  // for each loop, check whether agent can reach ball in less cycles
  for ( int i = 0; i <= PS->getPlayerWhenToIntercept(); i++ )
  {
    // (re-)initialize all the needed variables
    // set ball prediction one further to get right in front of ball line
    velMe   = WM->getAgentGlobalVelocity();
    angBody = WM->getAgentGlobalBodyAngle();
    angNeck = WM->getAgentGlobalNeckAngle();
```

```
  posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, i + 1 );
  posMe   = WM->getAgentGlobalPosition();
  ang     = (posBall - posPred).getDirection();
  ang     = VecPosition::normalizeAngle( ang - angBody );
  sta     = WM->getAgentStamina();
  int turn = 0;

  // as long as not correctly headed for point, simulate a turn command
  while (fabs(ang) > PS->getPlayerWhenToTurnAngle() && turn<5)
  {
   turn++;
   WM->predictStateAfterTurn( WM->getAngleForTurn(ang,velMe.getMagnitude()),
              &posMe, &velMe, &angBody, &angNeck,WM->getAgentObjectType(),
              &sta            );
   ang      = (posBall - posPred).getDirection();
   ang      = VecPosition::normalizeAngle( ang - angBody );
  }

  if( turn > 1 )
  {
   Log.log( 502, "nr of turns needed: %d", turn );
  }

  // for cycles that are left over after turn(s), execute full power dash
  for( ; turn < i; turn++ )
   WM->predictStateAfterDash(SS->getMaxPower(),&posMe,&velMe,&sta,angBody);

  // if in kickable distance or passed ball, we can reach the ball!
  if (posMe.getDistanceTo( posBall ) < dMaxDist  ||
     (posMe.getDistanceTo( posPred ) > posBall.getDistanceTo( posPred ) +
                                dMaxDist) )
  {
   WM->logCircle( 501, posBall, 2, true );
   WM->logCircle( 501, posBall, 1, true );
   *iCyclesBall = i;
   Log.log( 501, "intercept ball in %d cycles", *iCyclesBall );
   return posBall;
  }
 }

 *iCyclesBall = -1;
 return posBall;
}

VecPosition BasicPlayer::getDribblePoint( DribbleT dribble, double *dDist )
{
```

```
  AngDeg       angBody = WM->getAgentGlobalBodyAngle() ;
  VecPosition  posAgent = WM->getAgentGlobalPosition() ;
  VecPosition  pos;
  double       dLength;

  switch( dribble )
  {
   case DRIBBLE_WITHBALL:
     dLength = 5.0;   // 6: max 4.58, average 2.66
     *dDist = 4;
     break;
   case DRIBBLE_SLOW:
     dLength = 8.0;   // 8: max 6.47, average 3.34
     *dDist = 3.8;
     break;         // 9: max 8.57, average 5.38
   case DRIBBLE_FAST:
     dLength = 14.0;  // 14; max: 15.93 average 11.71
     *dDist = 12;
     break;
   default:
     dLength = 0.0;
     break;
  }

  // get point to shoot ball to
  pos = posAgent + VecPosition( dLength, angBody, POLAR );

  // adjust when point lies outside side of field
  // to a point that lies at distance 2.0 from the side of the field
  if( ( fabs( pos.getY() ) > PITCH_WIDTH/2.0  - 3.0 && fabs(angBody) > 3) ||
      ( fabs( pos.getX() ) > PITCH_LENGTH/2.0 - 3.0 ) )
    pos = WM->getOuterPositionInField( WM->getAgentGlobalPosition(),
                              WM->getAgentGlobalBodyAngle(), 2.0, false );

    return pos;
}

/*! This method returns the point on a line segment with which the ball has
    the largest angle with the surrounding opponents. It uses the method
    getDirectionOfWidestAngle. The line is determined by the two position
    p1 and p2. The returned position lies on this line and makes the largest
    angle with the opponents. The actual angle between the opponents is
    returned by 'angLargest'.
    \param p1 first position of line segment
    \param p2 second position of line segment
    \param *angLargest will be filled with the largest angle with the opponents
```

```
   \return VecPosition position on line that has the largest angle with the
           opponents */
VecPosition BasicPlayer::getShootPositionOnLine( VecPosition p1,
                                VecPosition  p2, AngDeg *angLargest )
{
  VecPosition posBall  = WM->getBallPos();
  Line   line          = Line::makeLineFromTwoPoints( p1, p2 );
  double dRadius       = min( PS->getClearBallOppMaxDist(),
                         posBall.getDistanceTo( p2 )  );
  AngDeg angMin        = (p1 - posBall ).getDirection();
  AngDeg angMax        = (p2 - posBall ).getDirection();
  // not correct when line crosses -180 boundary, but will never happenk
  AngDeg angShoot      = WM->getDirectionOfWidestAngle(
                        posBall, min(angMin, angMax),
                           max(angMin,angMax), angLargest, dRadius );
  Line   line2         = Line::makeLineFromPositionAndAngle( posBall,
                                              angShoot );
  return line.getIntersection( line2 );
}

/*! This method returns the end speed for a pass. This end speed is
    determined using two parameters, 'o' and 'posPass' which
    respectively denote the object to which is passed and the position to
    which is passed. First it is determined how many server cycles 'o'
    needs to travel to position 'posPass'. Then the starting speed of the ball
    is determined when it wants to travel to 'posPass' in the same number of
    cycles. This starting speed is adjusted if the corresponding end speed
    of the ball in the passing point lies outside the range
    [MinPassEndSpeed .. MaxPassEndSpeed].
    \param o object to which the ball will be passed
    \param posPass desired passing point where o can intercept the ball
    \return end speed to give to the ball such that player can intercept
            the ball the best. */
double BasicPlayer::getEndSpeedForPass( ObjectT o, VecPosition posPass )
{
  // we want that the ball arrives at that point after length nr of cycles
  // where length is the nr of cycles it takes the player to get there.
  VecPosition posBall = WM->getBallPos();
  double      dDist   = posBall.getDistanceTo( posPass );
  double      dLength = WM->predictNrCyclesToPoint( o, posPass );
  double      dFirst  = WM->getFirstSpeedFromDist( dDist, dLength );
  if( dFirst > SS->getBallSpeedMax() )
    dFirst = SS->getBallSpeedMax();
  double dEnd         = WM->getEndSpeedFromFirstSpeed( dFirst, dLength );
  if( dEnd > PS->getPassEndSpeed() )
    dEnd = PS->getPassEndSpeed();
```

```
  else if( dEnd < 0.6 )
    dEnd = 0.6;
  else if( dLength > 10.0 )
    dEnd = 0.6;

  return dEnd;
}

/*! This method returns a global position on the field which denotes the
position
    to mark object 'o'. It receives three arguments: an
    object o (usually an opponent) that the agent wants to mark, a distance
    'dDist' representing the desired distance between o and the marking
    position and a type indicator that denotes the type of marking that is
    required. We distinguish three types of marking:
    - MARK BALL: marking the opponent by standing at a distance 'dDist' away
                 from him on the line between him and the ball. This type of
                 marking will make it diÂ±cult for the opponent to receive a
                 pass.
    - MARK GOAL: marking the opponent by standing at a distance 'dDist' away
                 from him on the line between him and the center point of the
                 goal he attacks. This type of marking will make it difficult
                 for the opponent to score a goal.
    - MARK BISECTOR: marking the opponent by standing at a distance 'dDist'
                 away from him on the bisector of the ball-opponent-goal
                 angle. This type of marking enables the agent to intercept
                 both a direct and a leading pass to the opponent.
    \param o object that has to be marked
    \param dDist distance marking position is located from object position
    \param mark marking technique that should be used
    \return position that is the marking position. */
VecPosition BasicPlayer::getMarkingPosition( ObjectT o, double dDist,
                                             MarkT mark)
{
  VecPosition pos      = WM->getGlobalPosition( o );
  // except on back line assume players is moving to goalline
  if( pos.getX() > - PITCH_LENGTH/2.0 + 4.0 )
    pos -= VecPosition( 1.0, 0.0 );

  return WM->getMarkingPosition( pos, dDist, mark );

}


/*
  // stop and then turn
```

```
// this is stamina intensive??
if( bReady == true )
  ;
else if( fabs( angTurn ) > angThreshold && !bOppClose &&
   WM->getAgentSpeed() > 0.1 )
{
 posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
 soc = dashToPoint( posAgent );
 WM->predictAgentStateAfterCommand(soc,&posPred,&velMe,&angBody,&ang,&sta );
 if( posPred.getDistanceTo( posBall ) < 0.8*SS->getMaximalKickDist() )
 {
  Log.log( 508, "dash 1x (stop), turn 1x, dist %f, angle %f, opp %f",
    dDist2, angTurn, dDistOpp );
  WM->logCircle( 508, posPred, SS->getMaximalKickDist(), true );
  socFinal = soc;
 }
}
// if there are no opponents, we are wrongly directed, and we will be closely
// to the ball, see whether we can first update our heading
else if( fabs( angTurn ) > angThreshold && !bOppClose &&
   dDist1 < 0.7*SS->getMaximalKickDist() )
{
 soc = turnBodyToPoint( posPred1 + VecPosition(1,dDesBody, POLAR), 1 );
 Log.log( 508, "turn 1x, dist %f, angle %f, opp %f",
  dDist1, angTurn, dDistOpp );
 WM->logCircle( 508, posPred1, SS->getMaximalKickDist(), true );
 socFinal = soc;
}
else if( // fabs( angTurn ) > angThreshold &&
   !bOppClose &&
   dDist2 < 0.7*SS->getMaximalKickDist() )
{
 soc = turnBodyToPoint( posPred2 + VecPosition(1,dDesBody, POLAR), 2 );
 Log.log( 508, "turn 2x, dist %f, angle %f, opp %f",
   dDist2, angTurn, dDistOpp );
 WM->logCircle( 508, posPred2, SS->getMaximalKickDist(), true );
 socFinal = soc;
}


else if( socCollide.commandType != CMD_ILLEGAL &&
       fabs( angTurn ) > angThreshold )
{
 Log.log( 508, "collide with ball on purpose" );
 WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
 socFinal = socCollide;
```

```
  }
  else if( socDash1.commandType != CMD_ILLEGAL )
  {
    Log.log( 508, "do dash 1x" );
    WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
    socFinal = socDash1;
  }
*/
```

Player.h

```
#ifndef PLAYER
#define PLAYER

#include "BasicPlayer.h"
#include "Formations.h"     // needed for Formations

#ifdef WIN32
  DWORD WINAPI stdin_callback( LPVOID v );
#else
  void* stdin_callback( void * v );
#endif


/*! This class is a superclass from BasicPlayer and contains a more
    sophisticated decision procedure to determine the next action. */
class Player:public BasicPlayer
{
  Formations    *formations;          /*!< all formation information      */
  bool          bContLoop;           /*!< is server is alive             */

  Time          m_timeLastSay;       /*!< last time communicated         */
  ObjectT       m_objMarkOpp;        /*!< last marked opponent           */
  ObjectT       m_objPassOption;     /*!< passing option in kick         */

  int           m_iPenaltyNr;        /*!< number of current penalty      */

  ActionT       m_actionPrev;        /*!< previous action of this agent  */

  SoccerCommand goalieMainLoop       (                              );
  SoccerCommand defenderMainLoop     (                              );
  SoccerCommand midfielderMainLoop   (                              );
  SoccerCommand attackerMainLoop     (                              );

  void          performPenalty       (                              );


  VecPosition   getDeadBallPosition  (                              );

  // methods associated with saying (defined in Player.C)
  bool          shallISaySomething   ( SoccerCommand  soc            );
  bool          amIAgentToSaySomething ( SoccerCommand  soc          );
  void          sayOppAttackerStatus ( char *         str            );
  void          sayBallStatus        ( char *         str            );
  void          makeBallInfo         ( VecPosition    posBall,
                                       VecPosition    velBall,
                                       int            iDiff,
```

```
                              char *        str             );

public:
  Player                       ( ActHandler     *a,
                               WorldModel     *wm,
                               ServerSettings *ss,
                               PlayerSettings *cs,
                               Formations     *fs,
                               char          *strTeamName,
                               double         dVersion,
                               int           iReconnect = -1   );

  void        mainLoop              (                          );


  // methods that deal with user input (from keyboard) to sent commands
  void        handleStdin           (                          );
  void        showStringCommands    ( ostream& out                 );
  bool        executeStringCommand  ( char *str                    );

  // simple teams (PlayerTeams.cpp)
  SoccerCommand deMeer5             (                          );
  SoccerCommand deMeer5_goalie          (                          );


};

#endif
```

Player.cpp

```
#include "Player.h"
#include "Parse.h"
#ifndef WIN32
  #include <sys/poll.h> // needed for 'poll'
#endif

extern Logger LogDraw;

/*!This is the constructor the Player class and calls the constructor of the
   superclass BasicPlayer.
   \param act ActHandler to which the actions can be sent
   \param wm WorldModel which information is used to determine action
   \param ss ServerSettings that contain parameters used by the server
   \param ps PlayerSettings that contain parameters important for the client
   \param strTeamName team name of this player
   \param dVersion version this basicplayer corresponds to
   \param iReconnect integer that defines player number (-1 when new player) */
Player::Player( ActHandler* act, WorldModel *wm, ServerSettings *ss,
      PlayerSettings *ps,
      Formations *fs, char* strTeamName, double dVersion, int iReconnect )

{
  char str[MAX_MSG];

  ACT          = act;
  WM           = wm;
  SS           = ss;
  PS           = ps;
  formations    = fs;
  bContLoop     = true;
  m_iPenaltyNr  = -1;
  WM->setTeamName( strTeamName );
  m_timeLastSay = -5;
  m_objMarkOpp  = OBJECT_ILLEGAL;
  m_actionPrev  = ACT_ILLEGAL;

  // wait longer as role number increases, to make sure players appear at the
  // field in the correct order
#ifdef WIN32
  Sleep( formations->getPlayerInFormation()*500 );
#else
  poll( 0, 0, formations->getPlayerInFormation()*500 );
#endif

  // create initialisation string
```

```
  if( iReconnect != -1 )
    sprintf( str, "(reconnect %s %d)", strTeamName, iReconnect );
  else if( formations->getPlayerType() == PT_GOALKEEPER )
    sprintf( str, "(init %s (version %f) (goalie))", strTeamName, dVersion );
  else
    sprintf( str, "(init %s (version %f))", strTeamName, dVersion );
  ACT->sendMessage( str );

}

/*! This is the main loop of the agent. This method calls the update methods
    of the world model after it is indicated that new information has arrived.
    After this, the correct main loop of the player type is called, which
    puts the best soccer command in the queue of the ActHandler. */
void Player::mainLoop( )
{
  char str[MAX_MSG];
  Timing timer;

  // wait for new information from the server
  // cannot say bContLoop=WM->wait... since bContLoop can be changed elsewhere
  if(  WM->waitForNewInformation() == false )
    bContLoop =  false;

  // and set the clang version
  sprintf( str, "(clang (ver 8 8))" );
  ACT->sendMessage( str );

  while( bContLoop )                        // as long as server alive
  {
    Log.logWithTime( 3, "  start update_all" );
//    Log.setHeader( WM->getCurrentCycle(), WM->getPlayerNumber() );
    Log.setHeader( WM->getCurrentCycle() );
    LogDraw.setHeader( WM->getCurrentCycle() );

    if( WM->updateAll( ) == true )
    {
      timer.restartTime();
      SoccerCommand soc;
      if( ( WM->isPenaltyUs( ) || WM->isPenaltyThem() ) )
        performPenalty();
      else if( WM->getPlayMode() == PM_FROZEN )
        ACT->putCommandInQueue( turnBodyToObject( OBJECT_BALL )  );
      else
      {
        switch( formations->getPlayerType( ) )       // determine right loop
```

```
    {
     case PT_GOALKEEPER:       soc = goalieMainLoop( );     break;
     case PT_DEFENDER_SWEEPER:
     case PT_DEFENDER_CENTRAL:
     case PT_DEFENDER_WING:    soc = defenderMainLoop( );   break;
     case PT_MIDFIELDER_CENTER:
     case PT_MIDFIELDER_WING:  soc = midfielderMainLoop( ); break;
     case PT_ATTACKER:
     case PT_ATTACKER_WING:    soc = attackerMainLoop( );   break;
     case PT_ILLEGAL:
     default: break;
    }

    if( shallISaySomething(soc) == true )          // shall I communicate
    {
     m_timeLastSay = WM->getCurrentTime();
     char strMsg[MAX_SAY_MSG];
     if( WM->getPlayerNumber() == 6 &&
         WM->getBallPos().getX() < - PENALTY_X + 4.0  )
       sayOppAttackerStatus( strMsg );
     else
       sayBallStatus( strMsg );
     if( strlen( strMsg ) != 0 )
       Log.log( 600, "send communication string: %s", strMsg );
     WM->setCommunicationString( strMsg );
    }
   }
   Log.logWithTime( 3, "  determined action; waiting for new info" );
   // directly after see message, will nog get better info, so send commands
   if( WM->getTimeLastSeeMessage() == WM->getCurrentTime() ||
      (SS->getSynchMode() == true && WM->getRecvThink() == true ))
   {
    Log.logWithTime( 3, "  send messages directly" );
    ACT->sendCommands( );
    Log.logWithTime( 3, "  sent messages directly" );
    if( SS->getSynchMode() == true  )
    {
     WM->processRecvThink( false );
     ACT->sendMessageDirect( "(done)" );
    }
   }

  }
  else
   Log.logWithTime( 3, "  HOLE no action determined; waiting for new info");
```

```
    if( WM->getCurrentCycle()%(SS->getHalfTime()*SS->getSimulatorStep()) != 0 )
    {
      if( LogDraw.isInLogLevel( 600 ) )
      {
        WM->logDrawInfo( 600 );
      }

      if( LogDraw.isInLogLevel( 601 ) )
        WM->logDrawBallInfo( 601 );

      if( LogDraw.isInLogLevel( 700 ) )
        WM->logCoordInfo( 700 );
    }

    Log.logWithTime( 604, "time for action: %f", timer.getElapsedTime()*1000 );

    // wait for new information from the server cannot say
    // bContLoop=WM->wait... since bContLoop can be changed elsewhere
    if(  WM->waitForNewInformation() == false )
       bContLoop =  false;
  }

  // shutdow, print hole and number of players seen statistics
  printf("Shutting down player %d\n", WM->getPlayerNumber() );
  printf("   Number of holes: %d (%f)\n", WM->iNrHoles,
                  ((double)WM->iNrHoles/WM->getCurrentCycle())*100 );
  printf("   Teammates seen: %d (%f)\n", WM->iNrTeammatesSeen,
                  ((double)WM->iNrTeammatesSeen/WM->getCurrentCycle()));
  printf("   Opponents seen: %d (%f)\n", WM->iNrOpponentsSeen,
                  ((double)WM->iNrOpponentsSeen/WM->getCurrentCycle()));

}


/*! This is the main decision loop for the goalkeeper. */
SoccerCommand Player::goalieMainLoop( )
{
  return deMeer5_goalie();
}

/*! This is the main decision loop for a defender. */
SoccerCommand Player::defenderMainLoop( )
{
  return deMeer5() ;
}
```

```
/*! This is the main decision loop for a midfielder. */
SoccerCommand Player::midfielderMainLoop( )
{
  return deMeer5() ;
}

/*! This is the main decision loop for an agent. */
SoccerCommand Player::attackerMainLoop( )
{
  return deMeer5() ;
}

/*! This method returns the position to move in case of a dead ball situation.
    A dead ball situation occurs when the team can have a free kick, kick in,
    etc. The agent will move to the position behind the ball and when he is
    there will move to the ball again. */
VecPosition Player::getDeadBallPosition(  )
{
  VecPosition pos, posBall = WM->getBallPos();
  VecPosition posAgent = WM->getAgentGlobalPosition();
  double dDist;

  // determine point to move to
  if( WM->isKickInUs()  )
    pos = posBall + VecPosition( -1.5, sign( posBall.getY() )*1.5 );
  else if( WM->isCornerKickUs( ) )
    pos = posBall + VecPosition( 1.5, sign( posBall.getY() ) * 1.5 );
  else if( WM->isFreeKickUs() || WM->isOffsideThem() || WM->isGoalKickUs() ||
           WM->isFreeKickFaultThem() || WM->isBackPassThem() )
    pos = posBall + VecPosition( -1.5, 0.0 );
  else
    return VecPosition( UnknownDoubleValue, UnknownDoubleValue );

  AngDeg      angBall = (posBall-posAgent).getDirection() ;
  ObjectT     obj = WM->getClosestInSetTo( OBJECT_SET_PLAYERS,
                                  WM->getAgentObjectType(), &dDist);
  VecPosition posPlayer = WM->getGlobalPosition( obj );

  // change point when heading towards other player or towards the ball
  if( fabs( angBall - (posPlayer-posAgent).getDirection() ) < 20 &&
      dDist < 6 )
    pos -= VecPosition( 5, 0 );
  if( fabs( angBall -  (pos-posAgent).getDirection()) < 20 )
  {
    angBall = VecPosition::normalizeAngle( angBall - 90 );
    pos = posBall + VecPosition( 1, angBall , POLAR );
```

```
  }
  return pos;
}

/*!This method listens for input from the keyboard and when it receives this
   input it converts this input to the associated action. See
   showStringCommands for the possible options. This method is used together
   with the SenseHandler class that sends an alarm to indicate that a new
   command can be sent. This conflicts with the method in this method that
   listens for the user input (fgets) on Linux systems (on Solaris this isn't a
   problem). The only known method is to use the flag SA_RESTART with this
   alarm function, but that does not seem to work under Linux. If each time
   the alarm is sent, this gets function unblocks, it will cause major
   performance problems. This function should not be called when a whole match
   is played! */
void Player::handleStdin( )
{
  char buf[MAX_MSG];

  while( bContLoop )
  {
#ifdef WIN32
    cin.getline( buf, MAX_MSG );
#else
    fgets( buf, MAX_MSG, stdin ); // does unblock with signal !!!!!
#endif
   printf( "after fgets: %s\n", buf );
   executeStringCommand( buf );
  }
}

/*!This method prints the possible commands that can be entered by the user.
   The whole name can be entered to perform the corresponding command, but
   normally only the first character is sufficient. This is indicated by
   putting brackets around the part of the command that is not needed.
   \param out output stream to which the possible commands are printed */
void Player::showStringCommands( ostream& out )
{
  out << "Basic commands:"                    << endl <<
         " a(ctions)"                         << endl <<
         " c(atch) direction"                 << endl <<
         " cs(lientsettings"                  << endl <<
         " d(ash) power [ times ]"            << endl <<
         " de(bug) nr_cycles"                 << endl <<
         " g(oto) x y"                        << endl <<
         " h(elp)"                            << endl <<
```

```
        " i(ntercept) x y"                          << endl <<
        " k(ick) power angle"                          << endl <<
        " ka x y endspeed "                          << endl <<
        " m(ove) x y"                               << endl <<
        " n(eck) angle"                             << endl <<
        " o(pponents in cone) width dist"                << endl <<
        " p(redict cycles to) x y"                     << endl <<
        " q(uit)"                                  << endl <<
        " s(ay) message"                              << endl <<
        " ss(erversettings)"                          << endl <<
        " t(urn) angle"                             << endl <<
        " v(iewmode) narrow | normal | wide low | high"  << endl <<
        " w(orldmodel)"                               << endl;
}

/*!This method executes the command that is entered by the user. For the
   possible command look at the method showStringCommands.
   \param str string that is entered by the user
   \return true when command could be executed, false otherwise */
bool Player::executeStringCommand( char *str)
{
  SoccerCommand socCommand;
  int          i;
  double       x, y;

  switch( str[0] )
  {
    case 'a':                           // actions
      WM->showQueuedCommands();
      break;
    case 'c':                           // catch dir or cs
      if( strlen(str) > 1 && str[1] == 's' )
      {
        PS->show( cout, ":" );
        break;
      }
      socCommand.makeCommand( CMD_CATCH, Parse::parseFirstInt( &str ) );
      break;
    case 'd':                            // dash
      socCommand.commandType = CMD_DASH;
      socCommand.dPower      = Parse::parseFirstDouble( &str );
      socCommand.iTimes      = Parse::parseFirstInt   ( &str );
      if( socCommand.iTimes == 0 ) socCommand.iTimes = 1;
      break;
    case 'h':                           // help
      showStringCommands( cout );
```

```
  return true;
 case 'k':                          // kick or ka (kick advanced)
  socCommand.commandType = CMD_KICK;
  if( str[1] == 'a' ) // advanced kick
  {
   double x = Parse::parseFirstDouble( &str );
   double y = Parse::parseFirstDouble( &str );
   double e = Parse::parseFirstDouble( &str );
   socCommand = kickTo( VecPosition( x, y), e );
  }
  else
  {
   socCommand.dPower = Parse::parseFirstDouble( &str );
   socCommand.dAngle = Parse::parseFirstDouble( &str );
  }
  break;
 case 'm':                           // move
  socCommand.commandType = CMD_MOVE;
  socCommand.dX         = Parse::parseFirstDouble( &str );
  socCommand.dY         = Parse::parseFirstDouble( &str );
  socCommand.dAngle     = Parse::parseFirstDouble( &str );
  break;
 case 'n':                          // turn_neck
  socCommand.commandType = CMD_TURNNECK;
  socCommand.dAngle     = Parse::parseFirstDouble( &str );
  break;
 case 'o':                          // count nr opp in cone
  x = Parse::parseFirstDouble( &str );
  y = Parse::parseFirstDouble( &str );
  i = WM->getNrInSetInCone( OBJECT_SET_OPPONENTS, x,
                    WM->getAgentGlobalPosition(),
                    WM->getAgentGlobalPosition() +
                    VecPosition( y,
                                 WM->getAgentGlobalNeckAngle(),
                                 POLAR ) );
  printf( "%d opponents\n", i );
  return true;
 case 'p':                          // predict cycles to point
  x = Parse::parseFirstDouble( &str );
  y = Parse::parseFirstDouble( &str );
  i = WM->predictNrCyclesToPoint( WM->getAgentObjectType(),
                          VecPosition( x, y ) );
  printf( "%d cycles\n", i );
  return true;
 case 'q':                          // quit
  bContLoop = false;
```

```
      return true;
    case 's':                       // ss (serversettings) or say
      if( strlen(str) > 1 && str[1] == 's' )
      {
        SS->show( cout, ":" );
        break;
      }
      socCommand.commandType = CMD_SAY;
      Parse::gotoFirstOccurenceOf( ' ', &str );
      Parse::gotoFirstNonSpace( &str );
      strcpy( socCommand.str, str);
      break;
    case 't':                       // turn
      socCommand.commandType = CMD_TURN;
      socCommand.dAngle      = Parse::parseFirstDouble( &str );
      break;
    case 'v':                       // change_view
      socCommand.commandType = CMD_CHANGEVIEW;
      Parse::gotoFirstOccurenceOf(' ', &str );
      Parse::gotoFirstNonSpace( &str );
      socCommand.va          = SoccerTypes::getViewAngleFromStr( str );
      Parse::gotoFirstOccurenceOf(' ', &str );
      Parse::gotoFirstNonSpace( &str );
      socCommand.vq          = SoccerTypes::getViewQualityFromStr( str );
      break;
    case 'w':                       // worldmodel
      WM->show();
      return true;
    default:                        // default: send entered string
      ACT->sendMessage( str );
      return true;
  }
  if( socCommand.commandType != CMD_ILLEGAL ) // when socCommand is set
    ACT->putCommandInQueue( socCommand );     // send it.

  return true;
}

/*!This method can be called in a separate thread (see pthread_create) since
   it returns a void pointer. It is assumed that this function receives a
   BasicPlayer class as argument. The only thing this function does
   is starting the method handleStdin() from the corresponding BasicPlayer
   class that listens to user input from the keyboard. This function is
   necessary since a method from a class cannot be an argument of
   pthread_create.
   \param v void pointer to a BasicPlayer class.
```

```
   \return should never return since function handleStdin has infinite loop*/
#ifdef WIN32
DWORD WINAPI stdin_callback( LPVOID v )
#else
void* stdin_callback( void * v )
#endif

{
  Log.log( 1, "Starting to listen for user input" );
  Player* p = (Player*)v;
  p->handleStdin();
  return NULL;
}

/********************** SAY
**************************************************/

/*!This method determines whether a player should say something.
   \return bool indicating whether the agent should say a message */
bool Player::shallISaySomething( SoccerCommand socPri )
{
  bool       bReturn;

  bReturn  = ((WM->getCurrentTime() - m_timeLastSay) >= SS->getHearDecay());
  bReturn  &= amIAgentToSaySomething( socPri );
  bReturn  &= (WM->getCurrentCycle() > 0 );

  return bReturn;
}

/*! This method returns a boolean indicating whether I should communicate my
    world model to the other agents.
    \return boolean indicating whether I should communicate my world model. */
bool Player::amIAgentToSaySomething( SoccerCommand socPri )
{
  double  dDist;
  ObjectT obj;

  // get the closest teammate to the ball, if we are not him, we do not
  // communicate since he will
  obj = WM->getClosestInSetTo( OBJECT_SET_TEAMMATES,OBJECT_BALL,&dDist);
  if( dDist < SS->getVisibleDistance() &&
      obj != WM->getAgentObjectType() )
    return false;

  // in the defense, player 6 keeps track of the opponent attacker
```

```
  if( WM->getBallPos().getX() < - PENALTY_X + 4.0 &&
      WM->getConfidence( OBJECT_BALL ) > 0.96 &&
      WM->getPlayerNumber() == 6 &&
      WM->getCurrentCycle() % 3 == 0 ) // once very 3 cycles is enough
  {
    Log.log( 600, "player 6 is going to communicate attacker info" );
    return true;
  }

  VecPosition posBallPred;
  WM->predictBallInfoAfterCommand( socPri, &posBallPred );
  VecPosition posAgentPred = WM->predictAgentPosAfterCommand( socPri );
  // in all other cases inform teammates of ball when you have good information
  if( ( WM->getTimeChangeInformation(OBJECT_BALL) == WM->getCurrentTime() &&
          WM->getRelativeDistance( OBJECT_BALL ) < 20.0 &&
          WM->getTimeLastSeen( OBJECT_BALL ) == WM->getCurrentTime() )
        ||
      (
       WM->getRelativeDistance( OBJECT_BALL ) < SS->getVisibleDistance() &&
       WM->getTimeLastSeen( OBJECT_BALL ) == WM->getCurrentTime()
       )
      ||
      ( // pass ball
       WM->getRelativeDistance( OBJECT_BALL ) < SS->getMaximalKickDist() &&
       posBallPred.getDistanceTo( posAgentPred ) > SS->getMaximalKickDist()
       )
      )
    return true;

  return false;
}

/*! This method encodes the opponent attacker status in a visual message.
    \return string in which the opponent attacker position is encoded.*/
void Player::sayOppAttackerStatus( char* strMsg )
{
  char    strTmp[MAX_SAY_MSG];

  // fill the first byte with some encoding to indicate the current cycle.
  // The second byte   is the last digit of the cycle added to 'a'.
  sprintf( strMsg, "%c", 'a' + WM->getCurrentCycle()%10   );

  // fill the second byte with information about the offside line.
  // Enter either a value between a-z or A-Z indicating. This gives 52 possible
  // values which correspond with meters on the field. So B means the offside
  // line lies at 27.0.
```

```
  int iOffside = (int)WM->getOffsideX();
  if( iOffside < 26 ) // 0..25
    sprintf( strTmp, "%c", 'a' + iOffside );
  else              // 26...
    sprintf( strTmp, "%c", 'A' + max(iOffside - 26, 25) );
  strcat( strMsg, strTmp );

  // find the closest opponent attacker to the penalty spot.
  double dDist ;
  ObjectT objOpp = WM->getClosestInSetTo( OBJECT_SET_OPPONENTS,
        VecPosition(- PITCH_LENGTH/2.0 + 11.0, 0 ), &dDist  ) ;

  if( objOpp == OBJECT_ILLEGAL || dDist >= 20.0 )
  {
    strncpy( strMsg, "", 0 );
    return;
  }

  VecPosition posOpp = WM->getGlobalPosition( objOpp );
  if( fabs( posOpp.getY() ) > 10 )
  {
    strncpy( strMsg, "", 0 );
    return;
  }

  // encode the position of this attacker in the visual message. The
  // player_number is the third byte, then comes the x position in 3 digits (it
  // is assumed this value is always negative), a space and finally the y
  // position in 2 digits. An example of opponent nr. 9 at position
  // (-40.3223,-3.332) is "j403 -33)
  sprintf( strTmp, "%c%d %d", 'a' + SoccerTypes::getIndex( objOpp ) ,
                        (int)(fabs(posOpp.getX())*10),
                        (int)(posOpp.getY()*10));
  strcat( strMsg, strTmp );

  return ;
}

/*! This method creates a string to communicate the ball status. When
    the player just kicks the ball, it is the new velocity of the
    ball, otherwise it is the current velocity.
    \param strMsg will be filled */
void Player::sayBallStatus( char * strMsg  )
{
  VecPosition posBall = WM->getGlobalPosition( OBJECT_BALL );
  VecPosition velBall = WM->getGlobalVelocity( OBJECT_BALL );
```

```
  int iDiff = 0;
  SoccerCommand soc = ACT->getPrimaryCommand();

  if( WM->getRelativeDistance( OBJECT_BALL ) < SS->getMaximalKickDist() )
  {
    // if kick and a pass
    if( soc.commandType == CMD_KICK )
    {
      WM->predictBallInfoAfterCommand( soc, &posBall, &velBall );
      VecPosition posAgent = WM->predictAgentPos( 1, 0 );
      if( posBall.getDistanceTo( posAgent ) > SS->getMaximalKickDist() + 0.2 )
        iDiff = 1;
    }

    if( iDiff == 0 )
    {
      posBall = WM->getGlobalPosition( OBJECT_BALL );
      velBall.setVecPosition( 0, 0 );
    }
  }
  Log.log( 600, "create comm. ball after: (%1.2f,%1.2f)(%1.2f,%1.2f) diff %d",
     posBall.getX(), posBall.getY(), velBall.getX(), velBall.getY(), iDiff);
  makeBallInfo( posBall, velBall, iDiff, strMsg );
}

/*! This method is used to create the communicate message for the
    status of the ball, that is its position and velocity is encoded.

    \param VecPosition posBall ball position
    \param VecPosition velBall ball velocity
    \param iDiff time difference corresponding to given ball information
    \strMsg string message in which the ball information is encoded. */
void Player::makeBallInfo( VecPosition posBall, VecPosition velBall, int iDiff,
                           char * strMsg  )
{
  char    strTmp[MAX_SAY_MSG];

  // fill the first byte with some encoding to indicate the next cycle.
  // The second byte is the last digit of the cycle added to 'a'.
  sprintf( strMsg, "%c", 'a' + (WM->getCurrentCycle()+iDiff)%10   );

  // fill the second byte with information about the offside line.
  // Enter either a value between a-z or A-Z indicating. This gives 52 possible
  // values which correspond with meters on the field. So B means the offside
  // line lies at 27.0.
  int iOffside = (int)( WM->getOffsideX( false ) - 1.0 );
```

```
  if( iOffside < 26 ) // 0..25
    sprintf( strTmp, "%c", 'a' + max( 0, iOffside) );
  else              // 26...
    sprintf( strTmp, "%c", 'A' + min(iOffside - 26, 25) );
  strcat( strMsg, strTmp );

  // First add al values to a positive interval, since then we don't need
  // another byte for the minus sign. then take one digit at a time
  double x = max(0,min( rint( posBall.getX() + 48.0), 99.0));
  sprintf( strTmp, "%c%c%c%c%c%c%c%c%c",
     '0' + ((int)( x                        ) % 100 ) / 10 ,
     '0' + ((int)( x                        ) % 100 ) % 10 ,
     '0' + ((int)( rint(posBall.getY() + 34.0)) % 100 ) / 10 ,
     '0' + ((int)( rint(posBall.getY() + 34.0)) % 100 ) % 10 ,
     '0' + ((int)(( velBall.getX() + 2.7) * 10 )) / 10 ,
     '0' + ((int)(( velBall.getX() + 2.7) * 10 )) % 10 ,
     '0' + ((int)(( velBall.getY() + 2.7) * 10 )) / 10 ,
     '0' + ((int)(( velBall.getY() + 2.7) * 10 )) % 10 );
  strcat( strMsg, strTmp );
  Log.log( 6560, "say (%d) %s\n", WM->getPlayerNumber() , strMsg );

  return ;
}

/*! This method is called when a penalty kick has to be taken (for both the
  goalkeeper as the player who has to take the penalty. */
void Player::performPenalty( )
{
  VecPosition   pos;
  int           iSide = ( WM->getSide() == WM->getSidePenalty() ) ? -1 : 1;
  VecPosition   posPenalty( iSide*(52.5 - SS->getPenDistX()), 0.0 );
  VecPosition   posAgent = WM->getAgentGlobalPosition();
  AngDeg        angBody  = WM->getAgentGlobalBodyAngle();

  SoccerCommand soc(CMD_ILLEGAL);
  static PlayModeT pmPrev = PM_ILLEGAL;

  // raise number of penalties by one when a penalty is taken
  if(
    ( WM->getSide() == SIDE_LEFT &&
      pmPrev != PM_PENALTY_SETUP_LEFT &&
      WM->getPlayMode() == PM_PENALTY_SETUP_LEFT )
    ||
    ( WM->getSide() == SIDE_RIGHT &&
      pmPrev != PM_PENALTY_SETUP_RIGHT &&
      WM->getPlayMode() == PM_PENALTY_SETUP_RIGHT ) )
```

```
 m_iPenaltyNr++;

// start with player 11 and go downwards with each new penalty
// if we take penalty
if( WM->isPenaltyUs() && WM->getPlayerNumber() == (11 - (m_iPenaltyNr % 11)))
{
  if( WM->getPlayMode() == PM_PENALTY_SETUP_LEFT ||
      WM->getPlayMode() == PM_PENALTY_SETUP_RIGHT )
  {
    pos = posPenalty - VecPosition( iSide*2.0, 0 );
    if( fabs( posAgent.getX() ) > fabs( pos.getX() ) )
      pos = posPenalty;
    if( pos.getDistanceTo( posAgent ) < 0.6 )
    {
      pos = posPenalty;
      if(  fabs( VecPosition::normalizeAngle(
              (pos-posAgent).getDirection() - angBody ) ) > 20 )
        soc = turnBodyToPoint( pos );
    }
    //  pos = WM->getAgentGlobalPosition();
  }
  else if( ( WM->getPlayMode() == PM_PENALTY_READY_LEFT ||
             WM->getPlayMode() == PM_PENALTY_READY_RIGHT ||
             WM->getPlayMode() == PM_PENALTY_TAKEN_LEFT ||
             WM->getPlayMode() == PM_PENALTY_TAKEN_RIGHT
             )
           && WM->isBallKickable() )
  {
    soc = kickTo(VecPosition(iSide*52.5,((drand48()<0.5)?1:-1)*5.9 ),2.7);
  }
  else
    pos = posPenalty;
}
else if( formations->getPlayerType() == PT_GOALKEEPER )
{
  if( WM->getAgentViewAngle() != VA_NARROW )
    ACT->putCommandInQueue(
                                                    SoccerCommand(
CMD_CHANGEVIEW, VA_NARROW, VQ_HIGH ));

  // is penalty them, stop it, otherwise go to outside field
  pos = posPenalty;
  if( WM->isPenaltyThem( ) )
  {
    pos = VecPosition( iSide*(52.5 - 2.0), 0.0 );
    if( SS->getPenAllowMultKicks() == false )
```

```
    {
                                                                    if( WM->getPlayMode() ==
PM_PENALTY_TAKEN_LEFT ||
                                                                       WM->getPlayMode() ==
PM_PENALTY_TAKEN_RIGHT )
                                                                    {
                                                                     if( WM->isBallCatchable( ) )
                                                                       soc = catchBall();
                                                                     else
                                                                       soc = intercept( true );
                                                                    }
    }
    else if( pos.getDistanceTo( posAgent ) < 1.0 )
      soc = turnBodyToPoint( VecPosition( 0,0) ) ;
    else
      soc = moveToPos( pos, 25 );
   }
   else
    pos.setVecPosition( iSide * ( PITCH_LENGTH/2.0 + 2 ) , 25 );
 }
 else
 {
  pos = VecPosition( 5.0,
                     VecPosition::normalizeAngle(
                      iSide*(50 + 20*WM->getPlayerNumber())),
                     POLAR );
 }


 if( soc.isIllegal() &&
    WM->getAgentGlobalPosition().getDistanceTo( pos ) < 0.8 )
 {
  soc = turnBodyToPoint( posPenalty  );
 }
 else if( soc.isIllegal() )
 {
  soc = moveToPos( pos, 10);
 }
 if( WM->getAgentStamina().getStamina() <
    SS->getRecoverDecThr()*SS->getStaminaMax() + 500 &&
  soc.commandType == CMD_DASH)
  soc.dPower = 0;

 ACT->putCommandInQueue( soc );
 ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
```

```
  pmPrev = WM->getPlayMode();
}
```

PlayerSettings.h

```
#ifndef _PlayerSettings_
#define _PlayerSettings_

#include "GenericValues.h"

/*****************************************************************************
/
/******************** CLASS PlayerSettings *********************************/
/*****************************************************************************
/

/*! This class contains all the settings that are important for the client
   (agent) to determine its actions. It contains mostly threshold values to
   determine whether a certain kind of actions should be taken or not.
   Furthermore this class contains all the standard set- and get methods for
   manipulating these values. Although it is normally not the case that these
   values are changed at runtime. The PlayerSettings class is a subclass of the
   GenericValues class and therefore each value in this class can be reached
   through the string name of the corresponding parameter. This may be helpful
   when the parameters are taken from a configuration file. */
class PlayerSettings : public GenericValues
{
 double dPlayerConfThr;        /*!< confidence threshold below which player
                               information is assumed illegal        */
 double dPlayerHighConfThr;    /*!< confidence threshold above which player
                               information is assumed very good      */
 double dBallConfThr;          /*!< confidence threshold below which ball
                               information is assumed illegal        */
 double dPlayerDistTolerance;  /*!< radius in which player has to be to be
                               mapped from unknown to known player   */
 double dPlayerWhenToTurnAngle; /*!< angle when to turn to ball when moving */
 double dPlayerWhenToKick;     /*!< percentage of kick power rate when kick
                               is performed                          */
 int   iPlayerWhenToIntercept; /*!< how many cycles to ball when intercept */
 double dClearBallDist;        /*!< distance before the penalty area to
                               clear the ball to                     */
 double dClearBallOppMaxDist;  /*!< radius in which opponent has to be to
                                be taken into account for clear ball  */
 double dClearBallToSideAngle;  /*!< minimum angle between opponents before
                               clear ball is taken into account */
 double dConeWidth;            /*!< Cone width (at distance 1) to check for
                               opponents when passing to player.     */
 double dPassEndSpeed;         /*!< end speed ball when passed to teammate */
 double dFastPassEndSpeed;     /*!< end speed ball when passed fast       */
 double dPassExtraX;           /*!< extra x value added to x coordinate of
                               player pos to which is passed         */
```

```
  double dFractionWaitNoSee;     /*!< % of simulation step that is waited
                                  before action is sent in case no
                                  see message arrives                  */
  double dFractionWaitSeeBegin;  /*!< % of simulation step that is waited
                                  before action is sent in case see message
                                  arrives in first half cycle          */
  double dFractionWaitSeeEnd;    /*!< % of simulation step that is waited
                                  before action is sent in case see message
                                  arrives in second half cycle         */
  double dMarkDistance;          /*!< This is the distance the agent marks
                                  an opponent.                         */
  double dStratAreaRadius;       /*!< This is the radius around the strategic
                                  position in which an optimal position
                                  is determined.                       */
  double dShootRiskProbability;  /*!< This is the probability for the ball to
                                  enter the goal that is used when scoring
                                  with risk                            */
  int    iCyclesCatchWait;       /*!< Cycles the coach waits after a catch has
                                  been performed before he shoots.     */
  int    iServerTimeOut;         /*!< Number of seconds before the soccer
                                  server is assumed dead.              */
  double dDribbleAngThr;         /*!< Threshold value for angle difference
                                  that indicates when the agent turns
                                  towards the dribble direction.       */
  double dTurnWithBallAngThr;    /*!< Threshold value for angle difference
                                  that indicates when the ball is kicked in
                                  turnWithBallTo skill.                */
  double dTurnWithBallFreezeThr; /*!< Threshold value for ball speed that
                                  indicates when the ball is frozen in
                                  turnWithBallTo skill.                */
  int    iInitialFormation;      /*!< Initial formation for the team.      */
  double dMaxYPercentage;        /*!< Maximum y percentage of the field width
                                  for the y position in a strategic
                                  position. */

public:
  PlayerSettings( );

  // all standard get and set methods

  double getPlayerConfThr       (          ) const;
  bool   setPlayerConfThr       ( double d );

  double getPlayerHighConfThr   (          ) const;
  bool   setPlayerHighConfThr   ( double d );
```

```
double getBallConfThr          (          ) const;
bool   setBallConfThr          ( double d );

double getPlayerDistTolerance  (          ) const;
bool   setPlayerDistTolerance  ( double d );

double getPlayerWhenToTurnAngle(          ) const;
bool   setPlayerWhenToTurnAngle( double d );

double getPlayerWhenToKick     (          ) const;
bool   setPlayerWhenToKick     ( double d );

int    getPlayerWhenToIntercept(          ) const;
bool   setPlayerWhenToIntercept( int i    );

double getClearBallDist        (          ) const;
bool   setClearBallDist        ( double d );

double getClearBallOppMaxDist  (          ) const;
bool   setClearBallOppMaxDist  ( double d );

double getClearBallToSideAngle (          ) const;
bool   setClearBallToSideAngle ( double d );

double getConeWidth            (          ) const;
bool   setConeWidth            ( double d );

double getPassEndSpeed         (          ) const;
bool   setPassEndSpeed         ( double d );

double getFastPassEndSpeed     (          ) const;
bool   setFastPassEndSpeed     ( double d );

double getPassExtraX           (          ) const;
bool   setPassExtraX           ( double d );

double getFractionWaitNoSee    (          ) const;
bool   setFractionWaitNoSee    ( double d );

double getFractionWaitSeeBegin (          ) const;
bool   setFractionWaitSeeBegin ( double d );

double getFractionWaitSeeEnd   (          ) const;
bool   setFractionWaitSeeEnd   ( double d );

double getMarkDistance         (          ) const;
```

```
  bool   setMarkDistance         ( double d );

  double getStratAreaRadius      (          ) const;
  bool   setStratAreaRadius      ( double d );

  double getShootRiskProbability (          ) const;
  bool   setShootRiskProbability ( double d );

  int    getCyclesCatchWait      (          ) const;
  bool   setCyclesCatchWait      ( int i    );

  int    getServerTimeOut        (          ) const;
  bool   setServerTimeOut        ( int i    );

  double getDribbleAngThr        (          ) const;
  bool   setDribbleAngThr        ( double d );

  double getTurnWithBallAngThr   (          ) const;
  bool   setTurnWithBallAngThr   ( double d );

  double getTurnWithBallFreezeThr(          ) const;
  bool   setTurnWithBallFreezeThr( double d );

  int    getInitialFormation     (          ) const;
  bool   setInitialFormation     ( int i    );

  double getMaxYPercentage       (          ) const;
  bool   setMaxYPercentage       ( double d );


};

#endif
```

PlayerSettings.cpp

```
#include "PlayerSettings.h"

/*****************************************************************************
/
/********************* CLASS PlayerSettings *********************************/
/*****************************************************************************
/

/*! This method initializes all client settings and adds these to the generic
    values class with the effect that they can referenced by their textual
    name. */
PlayerSettings::PlayerSettings( ) : GenericValues("PlayerSettings", 27)
{
  dPlayerConfThr         = 0.88; // threshold below player info is illegal
  dPlayerHighConfThr     = 0.92; // threshold above which player info is high
  dBallConfThr           = 0.90; // threshold below which ball info is illegal
  dPlayerDistTolerance   = 7.5;  // distance when unknownplayer is mapped
  dPlayerWhenToTurnAngle = 7.0;  // angle when to turn when moving
  dPlayerWhenToKick      = 0.85; // % of kick power rate when kick is performed
  iPlayerWhenToIntercept = 30;   // maximum number of interception cycles
  dClearBallDist         = 5.0;  // dist before penalty area to clear ball to
  dClearBallOppMaxDist   = 30.0; // radius in which opp in clear ball has to be
  dClearBallToSideAngle  = 17.0; // minimum angle for clear ball to side
  dConeWidth             = 0.5;  // cone width to check for opponents in pass
  dPassEndSpeed          = 1.2;  // end speed for ball when passed to teammate
  dFastPassEndSpeed      = 1.8;  // end speed for ball when passed fast
  dPassExtraX            = 0.0;  // extra x value added to player passing pos
  dFractionWaitNoSee     = 0.61;  // % of cycle to wait in cycle with no see
  dFractionWaitSeeBegin  = 0.70; // % to wait in cycle with see in begin
  dFractionWaitSeeEnd    = 0.85; // % to wait in cycle with see in 2nd  half
  dMarkDistance          = 5.0;  // mark distance to a player
  dStratAreaRadius       = 5.0;  // radius around strat pos to find optimal pos
  dShootRiskProbability  = 0.88; // prob. of scoring when shooting with risk
  iCyclesCatchWait       = 20;   // cycles to wait after a catch
  iServerTimeOut         = 9;    // seconds before server is assumed dead
  dDribbleAngThr         = 20.0; // angle thr. to turn to ball before dribbling
  dTurnWithBallAngThr    = 45.0; // angle threshold to kick ball
  dTurnWithBallFreezeThr = 0.2;  // ball speed threshold to freeze ball
  iInitialFormation      = 2;    // initial formation number
  dMaxYPercentage        = 0.8;  // max y in strat. pos (percentage of field)

  // add all the settings and link text string to variable
  addSetting( "player_conf_thr"     , &dPlayerConfThr,  GENERIC_VALUE_DOUBLE );
  addSetting( "player_high_conf_thr", &dPlayerHighConfThr,
                                      GENERIC_VALUE_DOUBLE );
  addSetting( "ball_conf_thr"      ,  &dBallConfThr,    GENERIC_VALUE_DOUBLE );
```

```
  addSetting( "player_dist_tolerance",&dPlayerDistTolerance,
                                      GENERIC_VALUE_DOUBLE );
  addSetting( "player_when_to_turn",  &dPlayerWhenToTurnAngle,
                                      GENERIC_VALUE_DOUBLE );
  addSetting( "player_when_to_kick", &dPlayerWhenToKick,GENERIC_VALUE_DOUBLE );
  addSetting( "player_when_to_intercept",&iPlayerWhenToIntercept,
                                      GENERIC_VALUE_INTEGER);
  addSetting( "clear_ball_dist"    ,  &dClearBallDist,  GENERIC_VALUE_DOUBLE );
  addSetting( "clear_ball_opp_max_dist", &dClearBallOppMaxDist,
                                      GENERIC_VALUE_DOUBLE );
  addSetting( "clear_ball_side_angle", &dClearBallToSideAngle,
                                      GENERIC_VALUE_DOUBLE );
  addSetting( "cone_width"         ,  &dConeWidth,      GENERIC_VALUE_DOUBLE );
  addSetting( "pass_end_speed"     ,  &dPassEndSpeed,   GENERIC_VALUE_DOUBLE );
  addSetting( "fast_pass_end_speed", &dFastPassEndSpeed,GENERIC_VALUE_DOUBLE );
  addSetting( "pass_extra_x"       ,  &dPassExtraX,     GENERIC_VALUE_DOUBLE );
  addSetting( "wait_no_see"        ,&dFractionWaitNoSee,GENERIC_VALUE_DOUBLE );
  addSetting( "wait_see_begin" , &dFractionWaitSeeBegin,GENERIC_VALUE_DOUBLE );
  addSetting( "wait_see_end"    , &dFractionWaitSeeEnd, GENERIC_VALUE_DOUBLE );
  addSetting( "mark_distance"      ,  &dMarkDistance,   GENERIC_VALUE_DOUBLE );
  addSetting( "strat_area_radius"  ,  &dStratAreaRadius,GENERIC_VALUE_DOUBLE );
  addSetting( "shoot_risk_prob", &dShootRiskProbability,GENERIC_VALUE_DOUBLE );
  addSetting( "cycles_catch_wait"  ,  &iCyclesCatchWait,GENERIC_VALUE_INTEGER);
  addSetting( "server_time_out"    ,  &iServerTimeOut,  GENERIC_VALUE_INTEGER);
  addSetting( "dribble_ang_thr"    ,  &dDribbleAngThr,  GENERIC_VALUE_DOUBLE );
  addSetting( "turn_with_ball_ang_thr" , &dTurnWithBallAngThr,
                                      GENERIC_VALUE_DOUBLE );
  addSetting( "turn_with_ball_freeze_thr" , &dTurnWithBallFreezeThr,
                                      GENERIC_VALUE_DOUBLE );
  addSetting( "initial_formation"  , &iInitialFormation,GENERIC_VALUE_INTEGER);
  addSetting( "max_y_percentage"   , &dMaxYPercentage,  GENERIC_VALUE_DOUBLE );

}

/*! This method returns the confidence threshold below which player information
    is assumed illegal
   \return player confidence threshold */
double PlayerSettings::getPlayerConfThr( ) const
{
  return dPlayerConfThr;
}

/*! This method sets the confidence threshold below which player information is
    assumed illegal
    \param d player confidence threshold in range [0..1]
    \return boolean indicating whether update was successful */
```

```
bool PlayerSettings::setPlayerConfThr( double d )
{
  dPlayerConfThr = d;
  return true;
}

/*! This method returns the confidence threshold above which player information
    is assumed very good.
    \return player high confidence threshold */
double PlayerSettings::getPlayerHighConfThr( ) const
{
  return dPlayerHighConfThr;
}

/*! This method sets the confidence threshold above which player information is
    assumed very good
    \param d player high confidence threshold in range [0..1]
    \return boolean indicating whether update was successful */
bool PlayerSettings::setPlayerHighConfThr( double d )
{
  dPlayerHighConfThr = d;
  return true;
}

/*! This method returns the confidence threshold below which ball information
    is assumed illegal.
    \return ball confidence threshold */
double PlayerSettings::getBallConfThr( ) const
{
  return dBallConfThr;
}

/*! This method sets the confidence threshold below which ball information is
    assumed illegal
    \param d ball confidence threshold in range [0..1]
    \return boolean indicating whether update was successful */
bool PlayerSettings::setBallConfThr( double d )
{
  dBallConfThr = d;
  return true;
}

/*! This method returns the radius in which a player has to be to be
    mapped from unknown to known player
    \return radius in which player is assumed same player. */
double PlayerSettings::getPlayerDistTolerance( ) const
```

```
{
  return dPlayerDistTolerance;
}

/*! This method sets the radius in which a player has to be to be
    mapped from unknown to known player
    \param d radius (>0) in which player is assumed same player
    \return boolean indicating whether update was successful */
bool PlayerSettings::setPlayerDistTolerance( double d )
{
  dPlayerDistTolerance = d;
  return true;
}

/*!This method returns the angle when a player determines to turn to a point
   first before moving towards it.
   \return global angle when player first moves before moving towards point*/
double PlayerSettings::getPlayerWhenToTurnAngle( ) const
{
  return dPlayerWhenToTurnAngle;
}

/*! This method sets the angle when a player determines to turn to a point
    first before moving towards it.
    \param d global angle when player turns in move (interval [0..360]).
    \return boolean indicating whether update was successful */
bool PlayerSettings::setPlayerWhenToTurnAngle( double d )
{
  dPlayerWhenToTurnAngle = d;
  return true;
}

/*! This method returns the percentage of the maximal acceleration in which
    case a kick should still be performed. This value
    is needed to determine whether the ball should be better positioned or
    should be kicked when the ball should be kicked hard.
    If it is possible to accelerate the ball with a higher
    percentage than the returned percentage the kick is performed, in all other
    cases the ball is positioned better.
    \return percentage of ball acceleration when kick should be performed */
double PlayerSettings::getPlayerWhenToKick( ) const
{
  return dPlayerWhenToKick;
}

/*! This method sets the percentage of the maximal acceleration that defines
```

```
   in which cases the ball is actually kicked or in which case it is
   positioned better when the ball should be given a very high velocity.
   \param d percentage in range [0..1]
   \return boolean indicating whether update was successful */
bool PlayerSettings::setPlayerWhenToKick( double d )
{
 dPlayerWhenToKick = d;
 return true;
}

/*! This method returns the maximal allowed number of cycles to intercept the
   ball. If it takes more cycles to intercept the ball, the ball is not
   intercepted.
   \return number of intercept cycles */
int PlayerSettings::getPlayerWhenToIntercept( ) const
{
 return iPlayerWhenToIntercept;
}

/*! This methods sets the maximal allowed number of cycles to intercept the
   ball.
   \param i new maximal allowed number of cycles (>0)
   \return boolean indicating whether update was successful */
bool PlayerSettings::setPlayerWhenToIntercept( int i )
{
 iPlayerWhenToIntercept = i;
 return true;
}

/*! This method returns the clear ball distance. When a clear ball is
   performed, the ball is aimed to a point just in front of the penalty area
   of the opponent. This method returns the distance before the penalty area
   to which the ball is aimed.
   \return clear ball distance before opponent penalty area */
double PlayerSettings::getClearBallDist( ) const
{
 return dClearBallDist;
}


/*! This method sets the clear ball distance.
   \param d new clear ball distance before opponent penalty area (>0).
   \return boolean indicating whether update was successful */
bool PlayerSettings::setClearBallDist( double d )
{
 dClearBallDist = d;
```

```
  return true;
}

/*! This method returns the distance in which opponents are taken into account
    when a clear ball is issued.
    \return maximum opponent distance for clear ball. */
double PlayerSettings::getClearBallOppMaxDist( ) const
{
  return dClearBallOppMaxDist;
}

/*! This method sets the distance in which opponents are taken into account
    when a clear ball is issued.
    \param d maximum opponent distance for clear ball (>0).
    \return boolean indicating whether update was successful */
bool PlayerSettings::setClearBallOppMaxDist( double d )
{
  dClearBallOppMaxDist = d;
  return true;
}

/*! This method returns the minimum needed angle for a clear ball to the side.
    \return minimum needed angle for clear ball to side */
double PlayerSettings::getClearBallToSideAngle( ) const
{
  return dClearBallToSideAngle;
}

/*! This method sets the minimum needed angle for a clear ball to the side.
    \param d minimum needed angle (>0) for clear ball to side
    \return boolean indicating whether update was successful */
bool PlayerSettings::setClearBallToSideAngle( double d )
{
  dClearBallToSideAngle = d;
  return true;
}

/*! This method returns the cone width that is used to check for opponents when
    passing to a player. A pass is only performed when no opponents are in the
    cone. The cone is specified as the width to one side
    after distance 1. So for a value of 0.5 the cone angle equals
    45 (22.5 to both sides).
    \return cone width in which no opponents are allowed when passing */
double PlayerSettings::getConeWidth( ) const
{
  return dConeWidth;
```

```
}

/*! This method sets the cone width in which no opponents are allowed when the
    ball is passed to a teammate. The cone width is specified as the width to
    one side after distance 1. So for a value of 0.5 the cone angle
    equals 45 (22.5 to both sides).
    \param d cone width in which no opponents are allowed when passing (>0)
    \return boolean indicating whether update was successful */
bool PlayerSettings::setConeWidth( double d )
{
  dConeWidth = d;
  return true;
}

/*! This method returns the desired end speed for the ball when a normal pass
    is performed.
   \return desired end speed for ball */
double PlayerSettings::getPassEndSpeed( ) const
{
  return dPassEndSpeed;
}

/*! This method sets the desired end speed for the ball when a normal pass is
    performed.
    \param d desired end speed for ball (>0)
    \return boolean indicating whether update was successful */
bool PlayerSettings::setPassEndSpeed( double d )
{
  dPassEndSpeed = d;
  return true;
}

/*! This method returns the desired end speed for the ball when a fast pass is
    performed.
   \return desired end speed for ball when it is passed fast */
double PlayerSettings::getFastPassEndSpeed( ) const
{
  return dFastPassEndSpeed;
}


/*! This method sets the desired end speed for the ball when a fast pass is
    performed.
    \param d desired end speed for passing ball fast (>0)
    \return boolean indicating whether update was successful */
bool PlayerSettings::setFastPassEndSpeed( double d )
```

```
{
 dFastPassEndSpeed = d;
 return true;
}

/*! This method returns the x value that can be added to the position of a
    teammate when a leading pass to this teammate is performed.
    \return x value added to teammate to which is passed */
double PlayerSettings::getPassExtraX( ) const
{
 return dPassExtraX;
}

/*! This method sets the x value that is added to the position of a teammate
    when a leading pass is performed.
    \param d x value added to teammate to which is passed
    \return boolean indicating whether update was successful */
bool PlayerSettings::setPassExtraX( double d )
{
 dPassExtraX = d;
 return true;
}

/*! This method returns the fraction of the simulation step that is waited
    before an action is sent to the server in case no see message will arrive
    in this cycle.
    \return fraction of simulation step that is waited when no see arrives */
double PlayerSettings::getFractionWaitNoSee( ) const
{
 return dFractionWaitNoSee;
}

/*! This method sets the fraction of the simulation step that is waited before
    an action is sent to the server in case no see message will arrive in this
    cycle.
    \param d fraction of simulation step that is waited when no see arrives
    (in range [0..1])
    \return boolean indicating whether update was successful */
bool PlayerSettings::setFractionWaitNoSee( double d )
{
 dFractionWaitNoSee = d;
 return true;
}

/*! This method returns the fraction of the simulation step that is waited
    before an action is sent to the server in case the see message will arrive
```

```
    in the first half of the cycle.
    \return fraction of simulation step that is waited when see arrives in
            first half of the cycle */
double PlayerSettings::getFractionWaitSeeBegin( ) const
{
  return dFractionWaitSeeBegin;
}

/*! This method sets the fraction of the simulation step that is waited before
    an action is sent to the server in case the see message will arrive in the
    first part of the cycle.
    \param d fraction of simulation step that is waited when see arrives in the
    first half of the cycle (in range [0..1])
    \return boolean indicating whether update was successful */
bool PlayerSettings::setFractionWaitSeeBegin( double d )
{
  dFractionWaitSeeBegin = d;
  return true;
}

/*! This method returns the fraction of the simulation step that is waited
    before an action is sent to the server in case the see message will arrive
    in the second half of the cycle.
    \return fraction of simulation step that is waited when see arrives in
            second half of the cycle */
double PlayerSettings::getFractionWaitSeeEnd( ) const
{
  return dFractionWaitSeeEnd;
}

/*! This method sets the fraction of the simulation step that is waited before
    an action is sent to the server in case the see message will arrive in the
    second part of the cycle.
    \param d fraction of simulation step that is waited when see arrives in the
    second half of the cycle (in range [0..1])
    \return boolean indicating whether update was successful */
bool PlayerSettings::setFractionWaitSeeEnd( double d )
{
  dFractionWaitSeeEnd = d;
  return true;
}

/*! This method returns the desired distance to an opponent which should be
    marked.
    \return desired marking distance to an opponent */
double PlayerSettings::getMarkDistance( ) const
```

```
{
  return dMarkDistance;
}

/*! This method sets the desired distance to an opponent which should be
    marked.
    \param d desired marking distance to an opponent
    \return boolean indicating whether update was successful */
bool PlayerSettings::setMarkDistance( double d )
{
  dMarkDistance = d;
  return true;
}

/*! This method returns the radius around the strategic position in which
    an optimal position is considered.
    \return radius around strategic position in which an optimal position is
    considered */
double PlayerSettings::getStratAreaRadius( ) const
{
  return dStratAreaRadius;
}

/*! This method sets the radius around the strategic position in which
    an optimal position is considered.
    \param d radius around strategic position in which an optimal position is
          considered (>0)
    \return boolean indicating whether update was successful */
bool PlayerSettings::setStratAreaRadius( double d )
{
  dStratAreaRadius = d;
  return true;
}

/*! This method returns the minimum needed probability for the ball to enter
    the goal when a "risky" scoring attempt is performed. That is when an agent
    tries to score although the success rate is not very high, he will always
    shoot to a point in the goal where the probability that the ball will enter
    the goal is higher than the value returned by this method.
    \return minimum needed probability that the ball will enter the goal */
double PlayerSettings::getShootRiskProbability( ) const
{
  return dShootRiskProbability;
}
```

```
/*! This method sets the minimum needed probability for the ball to enter
    the goal when a "risky" scoring attempt is performed. That is when an agent
    tries to score although the success rate is not very high, he will always
    shoot to a point in the goal where the probability that the ball will enter
    the goal is higher than this value.
    \param d minimum needed probability that the ball enters the goal [0..1]
    \return boolean indicating whether update was successful */
bool PlayerSettings::setShootRiskProbability( double d )
{
 dShootRiskProbability = d;
 return true;
}

/*! This method returns the number of cycles waited by the goalkeeper after he
    has catched the ball.
    \return number of cycles goalkeeper waits after catch */
int PlayerSettings::getCyclesCatchWait( ) const
{
 return iCyclesCatchWait;
}

/*! This method sets the number of cycles waited by the goalkeeper after he
    has catched the ball.
    \param i number of cycles goalkeeper waits after catch
    \return boolean indicating whether update was successful */
bool PlayerSettings::setCyclesCatchWait( int i )
{
 iCyclesCatchWait = i;
 return true;
}

/*! This method returns the number of seconds before the server is assumed
    dead. When no message is received from the server for this amount of
    seconds, it is assumed that the server program has been closed and the
    agent will exits.
    \return number of seconds before server is assumed dead*/
int PlayerSettings::getServerTimeOut( ) const
{
 return iServerTimeOut;
}

/*! This method sets the number of seconds before the server is assumed dead.
    When no message is received from the server for this amount of seconds, it
    is assumed that the server is stopped and the agent exits.
    \param i number of seconds before server is assumed dead
    \return bool indicating whether update was succesfull. */
```

```
bool PlayerSettings::setServerTimeOut( int i )
{
  iServerTimeOut = i;
  return true;
}

/*! This method returns the threshold to determine when the agent turns towards
    the dribble direction when dribbling. When the global angle
    difference is larger than this value, the agent performs a turnWithBallTo.
   \return threshold value for angle */
double PlayerSettings::getDribbleAngThr( ) const
{
  return dDribbleAngThr;
}

/*! This method sets the threshold to determine when the agent turns towards
    the dribble direction when dribbling. When the global angle
    difference is larger than this value, the agent performs a turnWithBallTo.
    \param d threshold value for angle
    \return bool indicating whether update was succesfull. */
bool PlayerSettings::setDribbleAngThr( double d )
{
  dDribbleAngThr = d;
  return true;
}


/*! This method returns the threshold to determine when the ball is kicked to
    another position in the turnWithBallTo skill. When the global angle
    difference is larger than this value, the ball is repositioned otherwise
    it is not.
   \return threshold value for angle */
double PlayerSettings::getTurnWithBallAngThr( ) const
{
  return dTurnWithBallAngThr;
}

/*! This method sets the threshold to determine when the ball is kicked to
    another position in the turnWithBallTo skill. When the global angle
    difference is larger than this value, the ball is repositioned otherwise
    it is not.
    \param d threshold value for angle
    \return bool indicating whether update was succesfull. */
bool PlayerSettings::setTurnWithBallAngThr( double d )
{
  dTurnWithBallAngThr = d;
```

```
  return true;
}

/*! This method returns the threshold to determine when the ball is frozen
    in the turnWithBallTo skill. When the ball speed is larger than this value,
    the ball is frozen otherwise not
    \return threshold value for freezing the ball */
double PlayerSettings::getTurnWithBallFreezeThr( ) const
{
  return dTurnWithBallFreezeThr;
}

/*! This method sets the threshold to determine when the ball is frozen
    in the turnWithBallTo skill. When the ball speed is larger than this value,
    the ball is frozen otherwise not
    \param d threshold value for freezing the ball
    \return bool indicating whether update was succesfull. */
bool PlayerSettings::setTurnWithBallFreezeThr( double d )
{
  dTurnWithBallFreezeThr = d;
  return true;
}

/*! This method returns the initial formation of the team.
    \return number of cycles goalkeeper waits after catch */
int PlayerSettings::getInitialFormation( ) const
{
  return iInitialFormation;
}

/*! This method sets the initial formation of the team.
    \param i initital formation of the team
    \return boolean indicating whether update was successful */
bool PlayerSettings::setInitialFormation( int i )
{
  iInitialFormation = i;
  return true;
}

/*! This method returns the percentage of the field width. The corresponding y
    position is the maximum allowed y position for a player's strategic
    position.
    \return maximum y percentage on the field.  */
double PlayerSettings::getMaxYPercentage( ) const
{
  return dMaxYPercentage;
```

```
}

/*! This method sets the percentage of the field width. The corresponding y
    position is the maximum allowed y position for a player's strategic
    position.
    \param d percentage of the field width
    \return bool indicating whether update was succesfull. */
bool PlayerSettings::setMaxYPercentage( double d )
{
  dMaxYPercentage = d;
  return true;
}
```

```
#include "Player.h"

/*!This method is the first complete simple team and defines the actions taken
   by all the players on the field (excluding the goalie). It is based on the
   high-level actions taken by the simple team FC Portugal that it released in
   2000. The players do the following:
   - if ball is kickable
       kick ball to goal (random corner of goal)
   - else if i am fastest player to ball
       intercept the ball
   - else
       move to strategic position based on your home position and pos ball */
SoccerCommand Player::deMeer5(  )
{

  SoccerCommand soc(CMD_ILLEGAL);
  VecPosition   posAgent = WM->getAgentGlobalPosition();
  VecPosition   posBall  = WM->getBallPos();
  int           iTmp;

  if( WM->isBeforeKickOff( ) )
  {
    if( WM->isKickOffUs( ) && WM->getPlayerNumber() == 9 ) // 9 takes kick
    {
      if( WM->isBallKickable() )
      {
        VecPosition posGoal( PITCH_LENGTH/2.0,
                             (-1 + 2*(WM->getCurrentCycle()%2)) *
                             0.4 * SS->getGoalWidth() );
        soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick maximal
        Log.log( 100, "take kick off" );
      }
      else
      {
        soc = intercept( false );
        Log.log( 100, "move to ball to take kick-off" );
      }
      ACT->putCommandInQueue( soc );
      ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
      return soc;
    }
    if( formations->getFormation() != FT_INITIAL || // not in kickoff formation
        posAgent.getDistanceTo( WM->getStrategicPosition() ) > 2.0 )
    {
      formations->setFormation( FT_INITIAL );       // go to kick_off formation
      ACT->putCommandInQueue( soc=teleportToPos( WM->getStrategicPosition() ));
```

```
  }
  else                                  // else turn to center
  {
   ACT->putCommandInQueue( soc=turnBodyToPoint( VecPosition( 0, 0 ), 0 ) );
   ACT->putCommandInQueue( alignNeckWithBody( ) );
  }
 }
 else
 {
  formations->setFormation( FT_433_OFFENSIVE );
  soc.commandType = CMD_ILLEGAL;

  if( WM->getConfidence( OBJECT_BALL ) < PS->getBallConfThr() )
  {
   ACT->putCommandInQueue( soc = searchBall() );   // if ball pos unknown
   ACT->putCommandInQueue( alignNeckWithBody( ) ); // search for it
  }
  else if( WM->isBallKickable())                  // if kickable
  {
   VecPosition posGoal( PITCH_LENGTH/2.0,
         (-1 + 2*(WM->getCurrentCycle()%2)) * 0.4 * SS->getGoalWidth() );
   soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick maximal

   ACT->putCommandInQueue( soc );
   ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
   Log.log( 100, "kick ball" );
  }
  else if( WM->getFastestInSetTo( OBJECT_SET_TEAMMATES, OBJECT_BALL, &iTmp )
         == WM->getAgentObjectType()  && !WM->isDeadBallThem() )
  {                                     // if fastest to ball
   Log.log( 100, "I am fastest to ball; can get there in %d cycles", iTmp );
   soc = intercept( false );                    // intercept the ball

   if( soc.commandType == CMD_DASH &&           // if stamina low
       WM->getAgentStamina().getStamina() <
         SS->getRecoverDecThr()*SS->getStaminaMax()+200 )
   {
    soc.dPower = 30.0 * WM->getAgentStamina().getRecovery(); // dash slow
    ACT->putCommandInQueue( soc );
    ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
   }
   else                                 // if stamina high
   {
    ACT->putCommandInQueue( soc );              // dash as intended
    ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
   }
```

```
    }
    else if( posAgent.getDistanceTo(WM->getStrategicPosition()) >
             1.5 + fabs(posAgent.getX()-posBall.getX())/10.0)
                                    // if not near strategic pos
    {
     if( WM->getAgentStamina().getStamina() >     // if stamina high
                    SS->getRecoverDecThr()*SS->getStaminaMax()+800 )
     {
      soc = moveToPos(WM->getStrategicPosition(),
                    PS->getPlayerWhenToTurnAngle());
      ACT->putCommandInQueue( soc );            // move to strategic pos
      ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
     }
     else                                  // else watch ball
     {
      ACT->putCommandInQueue( soc = turnBodyToObject( OBJECT_BALL ) );
      ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
     }
    }
    else if( fabs( WM->getRelativeAngle( OBJECT_BALL ) ) > 1.0 ) // watch ball
    {
     ACT->putCommandInQueue( soc = turnBodyToObject( OBJECT_BALL ) );
     ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
    }
    else                                  // nothing to do
     ACT->putCommandInQueue( SoccerCommand(CMD_TURNNECK,0.0) );
   }
  return soc;
}

/*!This method is a simple goalie based on the goalie of the simple Team of
   FC Portugal. It defines a rectangle in its penalty area and moves to the
   position on this rectangle where the ball intersects if you make a line
   between the ball position and the center of the goal. If the ball can
   be intercepted in the own penalty area the ball is intercepted and catched.
*/
SoccerCommand Player::deMeer5_goalie(  )
{
  int i;
  SoccerCommand soc;
  VecPosition   posAgent = WM->getAgentGlobalPosition();
  AngDeg        angBody  = WM->getAgentGlobalBodyAngle();

  // define the top and bottom position of a rectangle in which keeper moves
  static const VecPosition posLeftTop( -PITCH_LENGTH/2.0 +
            0.7*PENALTY_AREA_LENGTH, -PENALTY_AREA_WIDTH/4.0 );
```

```
static const VecPosition posRightTop( -PITCH_LENGTH/2.0 +
          0.7*PENALTY_AREA_LENGTH, +PENALTY_AREA_WIDTH/4.0 );

// define the borders of this rectangle using the two points.
static Line  lineFront = Line::makeLineFromTwoPoints(posLeftTop,posRightTop);
static Line  lineLeft  = Line::makeLineFromTwoPoints(
                  VecPosition( -50.0, posLeftTop.getY()), posLeftTop );
static Line  lineRight = Line::makeLineFromTwoPoints(
                  VecPosition( -50.0, posRightTop.getY()),posRightTop );


if( WM->isBeforeKickOff( ) )
{
 if( formations->getFormation() != FT_INITIAL || // not in kickoff formation
     posAgent.getDistanceTo( WM->getStrategicPosition() ) > 2.0 )
 {
  formations->setFormation( FT_INITIAL );       // go to kick_off formation
  ACT->putCommandInQueue( soc=teleportToPos(WM->getStrategicPosition()) );
 }
 else                                  // else turn to center
 {
  ACT->putCommandInQueue( soc = turnBodyToPoint( VecPosition( 0, 0 ), 0 ));
  ACT->putCommandInQueue( alignNeckWithBody( ) );
 }
 return soc;
}

if( WM->getConfidence( OBJECT_BALL ) < PS->getBallConfThr() )
{                                     // confidence ball too  low
 ACT->putCommandInQueue( searchBall() );        // search ball
 ACT->putCommandInQueue( alignNeckWithBody( ) );
}
else if( WM->getPlayMode() == PM_PLAY_ON || WM->isFreeKickThem() ||
       WM->isCornerKickThem() )
{
 if( WM->isBallCatchable() )
 {
  ACT->putCommandInQueue( soc = catchBall() );
  ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
 }
  else if( WM->isBallKickable() )
 {
    soc = kickTo( VecPosition(0,posAgent.getY()*2.0), 2.0 );
    ACT->putCommandInQueue( soc );
    ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
 }
```

```
  else if( WM->isInOwnPenaltyArea( getInterceptionPointBall( &i, true ) ) &&
        WM->getFastestInSetTo( OBJECT_SET_PLAYERS, OBJECT_BALL, &i ) ==
                                WM->getAgentObjectType() )
  {
   ACT->putCommandInQueue( soc = intercept( true ) );
   ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
  }
  else
  {
   // make line between own goal and the ball
   VecPosition posMyGoal = ( WM->getSide() == SIDE_LEFT )
        ? SoccerTypes::getGlobalPositionFlag(OBJECT_GOAL_L, SIDE_LEFT )
        : SoccerTypes::getGlobalPositionFlag(OBJECT_GOAL_R, SIDE_RIGHT);
   Line lineBall = Line::makeLineFromTwoPoints( WM->getBallPos(),posMyGoal);

   // determine where your front line intersects with the line from ball
   VecPosition posIntersect = lineFront.getIntersection( lineBall );

   // outside rectangle, use line at side to get intersection
   if (posIntersect.isRightOf( posRightTop ) )
    posIntersect = lineRight.getIntersection( lineBall );
   else if (posIntersect.isLeftOf( posLeftTop )  )
    posIntersect = lineLeft.getIntersection( lineBall );

   if( posIntersect.getX() < -49.0 )
    posIntersect.setX( -49.0 );

   // and move to this position
   if( posIntersect.getDistanceTo( WM->getAgentGlobalPosition() ) > 0.5 )
   {
    soc = moveToPos( posIntersect, PS->getPlayerWhenToTurnAngle() );
    ACT->putCommandInQueue( soc );
    ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
   }
   else
   {
    ACT->putCommandInQueue( soc = turnBodyToObject( OBJECT_BALL ) );
    ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
   }
  }
 }
 else if( WM->isFreeKickUs() == true || WM->isGoalKickUs() == true )
 {
  if( WM->isBallKickable() )
  {
   if( WM->getTimeSinceLastCatch() == 25 && WM->isFreeKickUs() )
```

```
    {
     // move to position with lesser opponents.
     if( WM->getNrInSetInCircle( OBJECT_SET_OPPONENTS,
                                 Circle(posRightTop, 15.0 )) <
         WM->getNrInSetInCircle( OBJECT_SET_OPPONENTS,
                                  Circle(posLeftTop,  15.0 )) )
       soc.makeCommand( CMD_MOVE,posRightTop.getX(),posRightTop.getY(),0.0);
     else
       soc.makeCommand( CMD_MOVE,posLeftTop.getX(), posLeftTop.getY(), 0.0);
     ACT->putCommandInQueue( soc );
    }
    else if( WM->getTimeSinceLastCatch() > 28 )
    {
     soc = kickTo( VecPosition(0,posAgent.getY()*2.0), 2.0 );
     ACT->putCommandInQueue( soc );
    }
    else if( WM->getTimeSinceLastCatch() < 25 )
    {
     VecPosition posSide( 0.0, posAgent.getY() );
     if( fabs( (posSide - posAgent).getDirection() - angBody) > 10 )
     {
      soc = turnBodyToPoint( posSide );
      ACT->putCommandInQueue( soc );
     }
     ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
    }
   }
   else if( WM->isGoalKickUs()  )
   {
    ACT->putCommandInQueue( soc = intercept( true ) );
    ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
   }
   else
    ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
  }
  else
  {
   ACT->putCommandInQueue( soc = turnBodyToObject( OBJECT_BALL ) );
   ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
  }
  return soc;
}
```

Objects.h

```
#ifndef _OBJECTS_
#define _OBJECTS_

#include "SoccerTypes.h"  // needed for ObjectT

/*****************************************************************************
/
/********************** CLASS OBJECT
*****************************************/
/*****************************************************************************
/

/*! Class Object contains RoboCup information that is available for all objects
    in the simulation. All (relative) information is relative to an agent as
    declared in AgentObject. Update of an object (or one of the subclasses)
    happens by calling the standard get and set methods available in these
    classes. Calculations on these attributes do not occur in these classes,
    but in the update methods of the WorldModel. */
class Object
{
protected:
  ObjectT     objectType;           /*!< Type of this object              */
  Time        timeLastSeen;         /*!< Time last see message has arrived */

  VecPosition posGlobal;            /*!< Global position in the field     */
  Time        timeGlobalPosition;   /*!< Server time of global position   */
  VecPosition posRelative;          /*!< Relative position of the object  */
  Time        timeRelativePosition; /*!< Server time of relative position */
  VecPosition posGlobalLastSee;     /*!< Global position of last see msg  */
  Time        timeGlobalPosDerivedFromSee;/*!< Time pos derived from see msg */


public:
  Object( );

  /*! abstract function that should be defined in a subclass */
  virtual void show( ostream& os = cout ) = 0;

  // non-standard get and set methods (all defined here)
  AngDeg      getRelativeAngle          (                        );
  double      getRelativeDistance       (                        );
  double      getConfidence             ( Time       time        );

  // standard get and set methods
  bool        setType                   ( ObjectT    o           );
  ObjectT     getType                   (                        ) const;
```

```
  bool       setRelativePosition          ( double     dDist,
                                            AngDeg     dAng,
                                            Time       time            );
  bool       setRelativePosition          ( VecPosition v,
                                            Time       time            );
  VecPosition getRelativePosition         (                          ) const;

  bool       setTimeRelativePosition      ( Time       time            );
  Time       getTimeRelativePosition      (                          ) const;

  bool       setGlobalPosition            ( VecPosition p,
                                            Time       time            );
  VecPosition getGlobalPosition           (                          ) const;

  bool       setTimeGlobalPosition        ( Time       time            );
  Time       getTimeGlobalPosition        (                          ) const;

  bool       setGlobalPositionLastSee     ( VecPosition p,
                                            Time       time            );
  VecPosition getGlobalPositionLastSee    (                          ) const;

  bool       setTimeGlobalPosDerivedFromSee( Time       time            );
  Time       getTimeGlobalPosDerivedFromSee(                          ) const;

  bool       setTimeLastSeen              ( Time       time            );
  Time       getTimeLastSeen              (                          ) const;

};

/*****************************************************************************
/
/********************** CLASS FIXEDOBJECT
************************************/
/*****************************************************************************
/

/*! Class FixedObject contains RoboCup information that is available for
    objects that cannot move (flags, goals, lines). No additional information
    is added to the superclass Object. */
class FixedObject : public Object
{
public:
  // specific methods
  VecPosition getGlobalPosition( SideT s, double dGoalWidth = 14.02 ) const;
  AngDeg      getGlobalAngle   ( SideT s                            );
```

```
  void show                ( ostream & os = cout                );
};

/*****************************************************************************
/
/********************** CLASS DYNAMICOBJECT
**********************************/
/*****************************************************************************
/

/*! Class DynamicObject contains RoboCup information that is available for
    objects that can move (players, ball). Different variables are added to
    the superclass Object */
class DynamicObject: public Object
{
protected:

  // global velocity information
  VecPosition vecGlobalVelocity;      /*!< Global velocity of the player     */
  Time        timeGlobalVelocity;     /*!< Time of the corresponding velocity*/

  // sensor information
  double      dRelativeDistanceChange;/*!< Relative distance change          */
  double      dRelativeAngleChange;   /*!< Relative angle change             */
  Time        timeChangeInformation;  /*!< Time of change information        */

  VecPosition vecGlobalVelocityLastSee;/*!< vel. derived from last see       */


public:
  DynamicObject( );

  // standard get and set methods
  bool        setRelativeDistanceChange( double      d,   Time time );
  double      getRelativeDistanceChange(                        ) const;

  bool        setRelativeAngleChange   ( double      d,   Time time );
  double      getRelativeAngleChange   (                        ) const;

  bool        setTimeChangeInformation ( Time        time         );
  Time        getTimeChangeInformation (                        ) const;

  bool        setGlobalVelocity        ( VecPosition v,   Time time );
  VecPosition getGlobalVelocity        (                        ) const;
  double      getSpeed                 (                        ) const;
```

```
  bool       setTimeGlobalVelocity    ( Time       time          );
  Time        getTimeGlobalVelocity    (                         ) const;

  bool       setGlobalVelocityLastSee ( VecPosition vec          );
  VecPosition getGlobalVelocityLastSee (                         ) const;

};

/*****************************************************************************
/
/********************* CLASS PLAYEROBJECT
************************************/
/*****************************************************************************
/

/*! Class PlayerObject contains RoboCup information that is available for
    players. Different variables are added to the superclass DynamicObject   */
class PlayerObject: public DynamicObject
{
protected:
  bool    isKnownPlayer;        /*!< are we sure about player number       */
  ObjectT objRangeMin;          /*!< Minimum in range of possible player obj*/
  ObjectT objRangeMax;          /*!< Maximum in range of possible player obj*/
  bool    isGoalie;             /*!< is this object a goalie               */
  AngDeg  angGlobalBodyAngle;   /*!< Global body angle                     */
  AngDeg  angGlobalNeckAngle;   /*!< Global neck angle                     */
  AngDeg  angGlobalBodyAngleLastSee;/*!< Global body angle from last see msg */
  Time    timeGlobalAngles;     /*!< Server time of global angles          */
  int     iHeteroPlayerType;    /*!< index of heterogeneous player type    */

  Time    m_timeTackle;         /*!< time tackle command was observerd     */

  AngDeg  m_angGlobalArm;       /*!< global pointing direction of arm      */
  Time    m_timeGlobalArm;      /*!< time arm last seen                    */

private:
  AngDeg  angRelativeBodyAngle;  /*!< Relative body angle to main player */
  AngDeg  angRelativeNeckAngle;  /*!< Relative neck angle to main player */
  Time    timeRelativeAngles;    /*!< Server time of relative angles     */


public:
  PlayerObject( );

  void    show( ostream & os = cout );
```

```
    void    show( const char * strTeamName, ostream & os = cout );

    // standard get and set methods
    bool    setPossibleRange     ( ObjectT objMin, ObjectT objMax );
    bool    isInRange            ( ObjectT obj,    bool bTeamFirst);
    ObjectT getMinRange          (                                 );
    ObjectT getMaxRange          (                                 );

    bool    setIsKnownPlayer     ( bool     b                      );
    bool    getIsKnownPlayer     (                                 ) const;

    bool    setIsGoalie          ( bool     b                      );
    bool    getIsGoalie          (                                 ) const;

    bool    setRelativeBodyAngle ( AngDeg   ang,   Time    time   );
    AngDeg  getRelativeBodyAngle (                                 ) const;
    bool    setGlobalBodyAngle   ( AngDeg   ang,   Time    time   );
    AngDeg  getGlobalBodyAngle   (                                 ) const;

    bool    setRelativeNeckAngle ( AngDeg   ang,   Time    time   );
    AngDeg  getRelativeNeckAngle (                                 ) const;
    bool    setGlobalNeckAngle   ( AngDeg   ang,   Time    time   );
    AngDeg  getGlobalNeckAngle   (                                 ) const;

    bool    setTimeRelativeAngles( Time     time                   );
    Time    getTimeRelativeAngles(                                 ) const;
    bool    setTimeGlobalAngles  ( Time     time                   );
    Time    getTimeGlobalAngles  (                                 ) const;

    bool    setGlobalBodyAngleLastSee( AngDeg ang                  );
    AngDeg  getGlobalBodyAngleLastSee(                             ) const;

    bool    setHeteroPlayerType  ( int      index                  ) ;
    int     getHeteroPlayerType  (                                 ) const;

    bool    setTimeTackle        ( Time     time                   );
    Time    getTimeTackle        (                                 ) const;

    bool    setGlobalArm         ( AngDeg   ang,   Time    time   );
    AngDeg  getGlobalArm         (                                 ) const;

    bool    setTimeGlobalArm     ( Time     time                   );
    Time    getTimeGlobalArm     (                                 ) const;

};
```

```
/*****************************************************************************
/
/********************* CLASS BALLOBJECT
*************************************/
/*****************************************************************************
/

/*! Class PlayerObject contains RoboCup information that is available for the
   ball. No extra variables are added to superclass DynamicObject
*/
class BallObject: public DynamicObject
{
public:
  BallObject();
  void show( ostream & os = cout );

};

/*****************************************************************************
/
/********************* CLASS STAMINA
*****************************************/
/*****************************************************************************
/

/*! The following stamina information is stored in this class.
    - actual stamina
    - recovery, determines how much stamina recovers each cycle, decreases
              below a certain threshold (never increases)
    - effort, determines which percentage of dash power is actually used.
              decreases below certain threshold, increases above a higher
              threshold.
*/
class Stamina
{
  double m_dStamina;            /*!< Stamina value (>0)             */
  double m_dEffort;            /*!< Effort value (0..1)            */
  double m_dRecovery;           /*!< Recovery (0..1)                */

public:
  Stamina( double dSta = 4000.0, double dEff=1.0, double dRec=1.0 );
  void        show        ( ostream & os = cout                  );

  // return value how tired an agent is
                                                                TiredNessT   getTiredNess(
double dRecDecThr,   double dStaminaMax );
```

```
  // standard get and set methods.
  double       getStamina  (                          ) const;
  bool         setStamina  ( double d                    );
  double       getEffort   (                          ) const;
  bool         setEffort   ( double d                    );
  double       getRecovery (                          ) const;
  bool         setRecovery ( double d                    );
};

/*****************************************************************************
/
/********************* CLASS AGENTOBJECT
************************************/
/*****************************************************************************
/

/*! Class AgentObject contains RoboCup information that is available for the
    agent. New variables are declared that extend a normal PlayerObject.*/
class AgentObject: public PlayerObject
{
  ViewAngleT   viewAngle;           /*!< View angle of this agent         */
  ViewQualityT viewQuality;         /*!< View quality of this agent       */

  Stamina      stamina;             /*!< Stamina (stamina, effort, recovery*/
  VecPosition  velSpeedRelToNeck;   /*!< Velocity vector relative to neck  */
  AngDeg       angBodyAngleRelToNeck; /*!< Relative angle of body with neck  */

  VecPosition  posPositionDifference; /*!< Global pos difference with lastsee*/

  bool         m_bArmMovable;       /*!< Indicates whether can move arm.   */
  int          m_iArmExpires;       /*!< Nr. of cycles till arm expires    */
  VecPosition  m_posGlobalArm;      ;/*!< Global point to which arm point   */
  int          m_iTackleExpires;    /*!< Nr. of cycles till tackle expires */

public:
  AgentObject( double dStaminaMax = 4000 );

  void         show( ostream & os = cout );
  void         show( const char * strTeamName, ostream & os = cout );

  // standard get and set methods
  VecPosition  getPositionDifference(               ) const;
  bool         setPositionDifference( VecPosition  v   );

  ViewAngleT   getViewAngle        (               ) const;
```

```
  bool        setViewAngle        ( ViewAngleT   v   );

  ViewQualityT getViewQuality      (                ) const;
  bool        setViewQuality      ( ViewQualityT v   );

  Stamina     getStamina          (                ) const;
  bool        setStamina          ( Stamina      sta );

  VecPosition  getSpeedRelToNeck   (                ) const;
  bool        setSpeedRelToNeck   ( VecPosition  v   );

  bool        setGlobalNeckAngle   ( AngDeg       ang );

  AngDeg       getBodyAngleRelToNeck(                ) const;
  bool        setBodyAngleRelToNeck( AngDeg       ang );

  bool        getArmMovable       (                ) const;
  bool        setArmMovable       ( bool         b   );

  int         getArmExpires       (                ) const;
  bool        setArmExpires       ( int          i   );

  VecPosition  getGlobalArmPosition (                ) const;
  bool        setGlobalArmPosition ( VecPosition  v   );

  int         getTackleExpires     (                ) const;
  bool        setTackleExpires     ( int          i   );
} ;


#endif
```

Objects.cpp

```
#include"Objects.h"

#include<stdlib.h>       // needed for free
#include<iostream>       // needed for cout
#ifdef Solaris
  #include <strings.h>   // needed for strdup
#else
  #include <string.h>    // needed for strdup
#endif

/*****************************************************************************
/
/********************* CLASS OBJECT
******************************************/
/*****************************************************************************
/

/*! This constructor creates an object, all variables are initialized by
   default (illegal) values */
Object::Object( )
{
  objectType                = OBJECT_ILLEGAL;
}

/*! This method returns the relative angle of this object to the agent.
    This equals the angle of the relative position vector.
    \return relative angle to this object */
AngDeg Object::getRelativeAngle( )
{
  return VecPosition::normalizeAngle(posRelative.getDirection());
}

/*! This method returns the relative distance to this object.
    This equals the magnitude of the relative position vector.
    \return relative distance to this object */
double Object::getRelativeDistance( )
{
  return posRelative.getMagnitude();
}

/*! This method returns the confidence of the information of this object.
    The confidence is related to the last time this object was seen and
    the specified time (normally the time of the last received message).
    \param time current time to compare with time object was last seen
    \return confidence factor of this object */
double Object::getConfidence( Time time )
```

```
{
  if( timeLastSeen.getTime() == -1 )
    return 0.0;
  double dConf =
      max( 0.0, 1.0-(double)(time.getTimeDifference(timeLastSeen))/100.0);
  if( dConf > 1.0 )
    return 0.0;
  return dConf;
}


/*! This method sets the type of this object (i.e. OBJECT_BALL, FLAG_L_R_T).
    \param o ObjectT representing the type of this object
    \return bool indicating whether value was set. */
bool Object::setType( ObjectT o )
{
  objectType = o;
  return true;
}

/*! This method returns the type of this object.
    \return type of this object */
ObjectT  Object::getType() const
{
  return objectType;
}

/*! This method sets the relative position and the time this information was
    received. The relative position is calculated using the given distance and
    the given (relative) angle.
    \param dDist relative distance to object
    \param ang relative angle to object
    \param time time relative position was received
    \return bool indicating whether the values were set */
bool Object::setRelativePosition( double dDist, AngDeg ang, Time time )
{
  posRelative.setVecPosition( dDist, ang, POLAR );
  setTimeRelativePosition( time );
  return true;
}

/*! This method sets the relative position and the time this information was
    received using a vecPosition.
    \param p new relative position
    \param time time relative position was received
    \return bool indicating whether the values were set */
```

```
bool Object::setRelativePosition( VecPosition v, Time time )
{
  posRelative = v;
  setTimeRelativePosition( time );
  return true;
}

/*! This method returns the relative position of this object. The time of this
    information is related to the time returned by getTimeRelativePosition(),
    but is not checked. So if you want to know the relevance of this position
    use this method.
    \return relative position of this object */
VecPosition Object::getRelativePosition() const
{
  return posRelative;
}

/*! This method sets the time the relative position was received.
    \param time time relative position was received
    \return bool indicating whether the value was set */
bool Object::setTimeRelativePosition( Time time )
{
  timeRelativePosition = time;
  return true;
}

/*! This method returns the time that corresponds to the relative position of
    this object.
    \return time of the relative position of this object */
Time Object::getTimeRelativePosition() const
{
  return timeRelativePosition;
}

/*! This method sets the global position and the time this information was
    calculated.
    \param p new global position
    \param time time global position was received
    \return bool indicating whether the values were set */
bool Object::setGlobalPosition( VecPosition p, Time time )
{
  posGlobal = p;
  setTimeGlobalPosition( time );
  return true;
}
```

```
/*! This method returns the global position of this object. The time of this
    information is related to the time returned by getTimeGlobalPosition(),
    but is not checked. So if you want to know the relevance of this position
    use this method.
    \return global position of this object */
VecPosition Object::getGlobalPosition() const
{
  return posGlobal;
}

/*! This method sets the time the global position was calculated
    \param time time global position was calculated
    \return bool indicating whether the value was set */
bool Object::setTimeGlobalPosition( Time time )
{
  timeGlobalPosition = time;
  return true;
}

/*! This method returns the time that corresponds to the global position of
    this object.
    \return time of the global position of this object */
Time Object::getTimeGlobalPosition() const
{
  return timeGlobalPosition;
}

/*! This method sets the global position calculated using the last see
    message and the time of this see message. This opposed to the "normal"
    global position that is also updated when no see message has arrived
    in a new cycle.
    \param p new global position
    \param time time global position was received
    \return bool indicating whether the values were set */
bool Object::setGlobalPositionLastSee( VecPosition p, Time time )
{
  posGlobalLastSee = p;
  setTimeGlobalPosDerivedFromSee( time );
  return true;
}

/*! This method returns the global position of this object at the time of
    the last see message. The time of this information is related to the
    time returned by getTimeGlobalPosDerivedFromSee().
    \return global position of this object */
VecPosition Object::getGlobalPositionLastSee() const
```

```
{
 return posGlobalLastSee;
}

/*! This method sets the time the global position was calculated
    using a see message.
    \param time time global position was calculated with see message
    \return bool indicating whether the value was set */
bool Object::setTimeGlobalPosDerivedFromSee( Time time )
{
 timeGlobalPosDerivedFromSee = time;
 return true;
}

/*! This method returns the time that the global position was calculated
    using a see message.
    \return time of the global position of this object during the last see*/
Time Object::getTimeGlobalPosDerivedFromSee() const
{
 return timeGlobalPosDerivedFromSee;
}

/*! This method sets the time of the last see message in which this object was
    seen.
    \param time time this object was last seen
    \return bool indicating whether the value was set */
bool Object::setTimeLastSeen( Time time)
{
 timeLastSeen = time;
 return true;
}

/*! This method returns the time that corresponds to the time this object was
    located in the last see message.
    \return time of the last see message that was used to update this object */
Time Object::getTimeLastSeen() const
{
 return timeLastSeen;
}


/*****************************************************************************
/
/********************** CLASS FIXEDOBJECT
************************************/
```

```
/*************************************************************************
/

/*! This method prints all the information about this FixedObject to the
    specified output stream
    \param os output stream to print all relevant information to. */
void FixedObject::show( ostream & os )
{
 char buf[MAX_TEAM_NAME_LENGTH];

 // DEFAULT_TEAM_NAME is used since it is not a player, name does not matter
 SoccerTypes::getObjectStr( buf, objectType, DEFAULT_TEAM_NAME );
 os <<  buf
    << " rel("  << posRelative << " r:" << posRelative.getMagnitude()
    << " phi:"  << posRelative.getDirection()
    << " t:"    << timeRelativePosition << ")"
    << " seen:" << timeLastSeen << "\n";
}

/*! This method returns the global position of this fixed object. Only works
    when the object type equals a flag or a goal. Furthermore the side of the
    agent has to be passed since the global positions differ for the left and
    the right side. For some flags the size of the goal is important. So this
    value can be passed also. Otherwise the default value is 14.02.
    \param s side of agent (global position differs for left and right side)
    \param dGoalWidth width of a goal, needed for pole objects next to goal
    \return global position of this fixed object. */
VecPosition FixedObject::getGlobalPosition( SideT s, double dGoalWidth )const
{
 return SoccerTypes::getGlobalPositionFlag( getType(), s, dGoalWidth );
}

/*! This methods returns the global angle of this fixed object in the world.
    Only works when the fixed object is a line. The angle for the left team
    rises clockwise, i.e. left=0, bottom=90, etc. For the right team this
    is counterclockwise: right=0, top=90, etc)
    \param s side of agent (angles differ for left and right side)
    \return global angle of this line in the world. */
AngDeg FixedObject::getGlobalAngle( SideT s )
{
 return SoccerTypes::getGlobalAngleLine( getType(), s );
}

/*************************************************************************
/
```

```
/********************* CLASS DYNAMICOBJECT
***********************************/
/*****************************************************************************
/

/*! This is the constructor for DynamicObject. A DynamicObject is created with
    all the variables initialized by (illegal) default values */
DynamicObject::DynamicObject( ):Object( )
{
 dRelativeDistanceChange   = UnknownDoubleValue;
 dRelativeAngleChange      = UnknownDoubleValue;
}

/*! This method sets the global velocity of this object and the time of this
    information
    \param v new global velocity
    \param time time global velocity was received
    \return bool indicating whether the values were set */
bool DynamicObject::setGlobalVelocity( VecPosition v, Time time)
{
 if( v.getMagnitude() < EPSILON )
   vecGlobalVelocity.setVecPosition( 0.0, 0.0 );
 else
    vecGlobalVelocity = v;
 setTimeGlobalVelocity( time );
 return true;
}

/*! This method returns the global velocity of this object. The time of this
    information is related to the time returned by getTimeGlobalVelocity().
    \return global position of this object */
VecPosition DynamicObject::getGlobalVelocity( ) const
{
 return vecGlobalVelocity;
}

/*! This method returns the speed of this object.
    The speed is the magnitude of the global velocity of the object
    \return speed of this object (zero for non-moving objects) */
double DynamicObject::getSpeed( ) const
{
 return vecGlobalVelocity.getMagnitude();
}

/*! This method sets the time that corresponds to the last update of the
    global velocity of this object.
```

```
    \param time time corresponding to current value of global velocity
    \return bool indicating whether the value was set */
bool DynamicObject::setTimeGlobalVelocity( Time time )
{
  timeGlobalVelocity = time;
  return true;
}

/*! This method returns the time that belongs to the global velocity of this
    object.
    \return time of the global velocity of this object */
Time DynamicObject::getTimeGlobalVelocity() const
{
  return timeGlobalVelocity;
}


/*! This method sets the relative distance change and the time this
    information was calculated.
    \param d new relative distance change
    \param time time relative distance change was calculated
    \return bool indicating whether the values were set */
bool DynamicObject::setRelativeDistanceChange( double d, Time time )
{
  dRelativeDistanceChange = d;
  setTimeChangeInformation( time );
  return true;
}

/*! This method returns the relative distance change of this object. Note that
    this value is zero when object is at the same distance, but at a complete
    different angle. This occurs when an object has moved a lot in one cycle.
    This information belongs to the server time that is returned by
    getTimeChangeInformation().
    \return relative distance change of object in the last cycle */
double DynamicObject::getRelativeDistanceChange() const
{
  return dRelativeDistanceChange;
}

/*! This method sets the relative angle change and the server time this
    information belongs to.
    \param d new relative angle change
    \param time time relative angle change was received
    \return bool indicating whether the values were set */
bool DynamicObject::setRelativeAngleChange( double d, Time time )
```

```
{
  dRelativeAngleChange = d;
  setTimeChangeInformation( time );
  return true;
}

/*! This method returns the relative angle change of this object.
    This information belongs to the server time that is returned by
    getTimeChangeInformation().
    \return relative angle change of object in the last cycle */
double DynamicObject::getRelativeAngleChange() const
{
  return dRelativeAngleChange;
}

/*! This method sets the time the change information was calculated.
    \param time time information for change was calculated
    \return bool indicating whether the values was set */
bool DynamicObject::setTimeChangeInformation( Time time )
{
  timeChangeInformation = time ;
  return true;
}

/*! This method returns the server time that belongs to the relative distance
    and relative angle change of this object.
    \return time of the change information of this DynamicObject */
Time DynamicObject::getTimeChangeInformation()  const
{
  return timeChangeInformation;
}

/*! This method sets the global velocity of the object calculated after the last
    see message. The time of this information corresponds to
    'getTimeChangeInformation'.
    \return global body velocity derived from the last see message */
bool DynamicObject::setGlobalVelocityLastSee ( VecPosition vec )
{
  vecGlobalVelocityLastSee = vec;
  return true;
}

/*! This method returns the global velocity of the object calculated after the
    last see message. The time of this information corresponds to
    'getTimeChangeInformation'.
    \return global body velocity derived from the last see message */
```

```
VecPosition DynamicObject::getGlobalVelocityLastSee ( )   const
{
 return vecGlobalVelocityLastSee;
}


/*************************************************************************
/
/********************* CLASS PLAYEROBJECT
************************************/
/*************************************************************************
/

/*! This is the constructor for PlayerObject. A PlayerObject is created with
   all variables initialized to (illegal) default values */
PlayerObject::PlayerObject( ):DynamicObject( )
{
 angGlobalBodyAngle   = UnknownAngleValue;
 angGlobalNeckAngle   = UnknownAngleValue;
 isKnownPlayer        = false;
 isGoalie             = false;

 angRelativeBodyAngle = UnknownAngleValue;
 angRelativeNeckAngle = UnknownAngleValue;

 iHeteroPlayerType    = 0;
}

/*! This method sets the facing direction of the body and the time of this
   information (all relative to the agent).
   \param ang new relative facing direction of body
   \param time time corresponding to the facing direction
   \return bool indicating whether the values were set */
bool PlayerObject::setRelativeBodyAngle( AngDeg ang, Time time )
{
 angRelativeBodyAngle = ang;
 setTimeRelativeAngles( time );
 return true;
}

/*! This method returns the relative body angle of this object. This
   information is from the server time that is returned by
   getTimeRelativeAngles().

   \return relative body angle of this object */
AngDeg PlayerObject::getRelativeBodyAngle( )  const
```

```
{
 return angRelativeBodyAngle;
}

/*! This method sets the facing direction of the body and the time of this
    information (all global).
    \param ang new global facing direction of body
    \param time time corresponding to the facing direction
    \return bool indicating whether the values were set */
bool PlayerObject::setGlobalBodyAngle( AngDeg ang, Time time)
{
 angGlobalBodyAngle = ang;
 setTimeGlobalAngles( time );
 return true;
}

/*! This method returns the global body angle of this object. This information
    is from the server time that is returned by getTimeGlobalAngles().
    \return global body angle of this object */
AngDeg PlayerObject::getGlobalBodyAngle( )  const
{
 return angGlobalBodyAngle;
}

/*! This method returns the facing direction of the neck and the time of
    this information (all relative to the agent).
    \param ang new relative facing direction of neck
    \param time time facing direction was received
    \return bool indicating whether the values were set */
bool PlayerObject::setRelativeNeckAngle( AngDeg ang, Time time )
{
 angRelativeNeckAngle = ang;
 setTimeRelativeAngles( time );
 return true;
}

/*! This method returns the relative neck angle of this object. This
    information is from  the time  that is returned by getTimeRelativeAngles().
    \return relative neck angle of this object */
AngDeg PlayerObject::getRelativeNeckAngle( )  const
{
 return angRelativeNeckAngle;
}

/*! This method returns the facing direction of the neck and the time of
    this information (all global).
```

```
    \param ang new global facing direction of neck
    \param iTime time facing direction was received
    \return bool indicating whether the values were set */
bool PlayerObject::setGlobalNeckAngle( AngDeg ang, Time time )
{
  angGlobalNeckAngle = ang;
  setTimeGlobalAngles( time );
  return true;
}

/*! This method returns the global neck angle of this object. This
    information is from  the time that is returned by getTimeGlobalAngles().
    \return global neck angle of this object */
AngDeg PlayerObject::getGlobalNeckAngle( )  const
{
  return angGlobalNeckAngle;
}

/*! This method sets the time the facing direction was calculated.
    \param time time the facing direction was received
    \return bool indicating whether the values were set */
bool PlayerObject::setTimeRelativeAngles( Time time )
{
  timeRelativeAngles = time;
  return true;
}

/*! This method returns the server time in which the relative body and neck
    angle of this object were calculated.
    \return time of the relative neck and body information */
Time PlayerObject::getTimeRelativeAngles( ) const
{
  return timeRelativeAngles ;
}

/*! This method sets the time the facing direction was calculated.
    \param time time the facing direction was received
    \return bool indicating whether the values were set */
bool PlayerObject::setTimeGlobalAngles( Time time )
{
  timeGlobalAngles = time;
  return true;
}


/*! This method sets the global angle of the agent calculated after the last
```

```
    see message. The time of this information corresponds to
    'getTimeChangeInformation', since the change information arrives always
    together with the body and neck information.
    \return global body angle derived from the last see message */
bool PlayerObject::setGlobalBodyAngleLastSee( AngDeg ang )
{
  angGlobalBodyAngleLastSee = ang;
  return true;
}


/*! This method returns the global angle of the agent calculated after
    the last see message. The time of this information corresponds to
    'getTimeChangeInformation', since the change information arrives
    always together with the body and neck information.

    \return global body angle derived from the last see message */
AngDeg PlayerObject::getGlobalBodyAngleLastSee( )  const
{
  return angGlobalBodyAngleLastSee;
}


/*! This method returns the server time in which the global body and neck
    angle of this object were calculated.
    \return time of the global neck and body information */
Time PlayerObject::getTimeGlobalAngles( ) const
{
  return timeGlobalAngles ;
}

/*! This method sets the possible range from which this object must originate
    from. Since the ordening of objects in a 'see' message is always the same
    (first teammates then opponents and always sorted by player number), it
    is possible to derive the range of objects from which the supplied
    information must originate.
    \param objMin minimum object type
    \param objMax maximum object type
    \return bool indicating whether update was successful. */
bool PlayerObject::setPossibleRange( ObjectT objMin, ObjectT objMax )
{
  objRangeMin = objMin;
  objRangeMax = objMax;
  return true;
}
```

```
/*! This method returns a boolean indicating whether the supplied object type,
    is in the range of possible object types set by 'setPossibleRange'.
    \param obj Object type that should be checked
    \return boolean indicating whether 'obj' is located in the range. */
bool PlayerObject::isInRange( ObjectT obj, bool bTeammatesFirst )
{
  int  iIndObj = SoccerTypes::getIndex( obj );
  int  iIndMin = SoccerTypes::getIndex( objRangeMin );
  int  iIndMax = SoccerTypes::getIndex( objRangeMax );

  // teammates 0 to 10, opponents 11 to 21 or -10..1 I guess
  if( SoccerTypes::isOpponent( obj ) )
    iIndObj += ( bTeammatesFirst ) ? 11 : -11 ;
  if( SoccerTypes::isOpponent( objRangeMin ) )
    iIndMin += ( bTeammatesFirst ) ? 11 : -11 ;
  if( SoccerTypes::isOpponent( objRangeMax ) )
    iIndMax += ( bTeammatesFirst ) ? 11 : -11 ;

  return iIndMin <= iIndObj && iIndObj <= iIndMax ;
}

/*! This method returns the minimal possible object type as set by
   'setPossibleRange'.
    \return minimal possible object type */
ObjectT PlayerObject::getMinRange( )
{
  return objRangeMin;
}

/*! This method returns the maximal possible object type as set by
   'setPossibleRange'.
    \return maximal possible object type */
ObjectT PlayerObject::getMaxRange( )
{
  return objRangeMax;
}

/*! This method sets whether this dynamic object is a known player or not.
    A known player is a player of which we know the number. If we don't know
    the player number of a player, the player is put at the index of a player
    that isn't seen in a while and the isKnownPlayer attribute is set to false.
    \param b bool indicating whether player number is known
    \return bool indicating whether value was set. */
bool PlayerObject::setIsKnownPlayer( bool b )
{
  isKnownPlayer = b;
```

```
  return true;
}

/*! This method returns whether the current object is a known player or not.
    A known player is a player of which we know the number. If we don't know
    the player number of a player, the player is put at the index of a player
    that isn't seen in a while and the isKnownPlayer attribute is set to false.
    \return bool indicating whether player number is known */
bool PlayerObject::getIsKnownPlayer() const
{
  return isKnownPlayer;
}

/*! This method sets whether this dynamic object is a goalie or not.
    \param b bool indicating whether this dynamic object is a goalie
    \return bool indicating whether value was set. */
bool PlayerObject::setIsGoalie( bool b )
{
  isGoalie = b;
  return true;
}

/*! This method returns whether the current object is a goalie or not.
    \return bool indicating whether this dynamic object is a goalie or not */
bool PlayerObject::getIsGoalie() const
{
  return isGoalie;
}


/*! This method prints the information about this PlayerObject to the specified
    output stream. The variables are printed with the default team name.
    \param os output stream to which output is written */
void PlayerObject::show( ostream & os )
{
  show( DEFAULT_TEAM_NAME, os );
}

/*! This method prints the information about this PlayerObject to the specified
    output stream. The variables are printed with the specified team name.
    \param strTeamName team name of this player object
    \param os output stream to which output is written */
void PlayerObject::show( const char* strTeamName , ostream & os )
{
  char buf[MAX_TEAM_NAME_LENGTH];
  SoccerTypes::getObjectStr( buf, objectType, strTeamName );
```

```
  os << buf
          << " conf: "            << getConfidence( timeGlobalPosition )
          << " pos: "             << posGlobal
          << ","                  << timeGlobalPosition
          << " vel: "             << vecGlobalVelocity
          << ","                  << timeGlobalVelocity
          << " ang (b:"           << getGlobalBodyAngle()
          << ",n:"                << angGlobalNeckAngle
          << "),"                 << timeGlobalAngles
          << "known: "            << isKnownPlayer
          << " lastseen:"         << timeLastSeen << "\n";
}

/*! This method sets the heterogeneous index number of this player object.
    \param hetereogeneous index number
    \return bool indicating whether update was successful. */
bool PlayerObject::setHeteroPlayerType( int index )
{
  iHeteroPlayerType = index;
  return true;
}

/*! This method returns the heterogeneous index number of this player object.
    \return hetereogeneous index number */
int  PlayerObject::getHeteroPlayerType( ) const
{
  return iHeteroPlayerType;
}


/*! This method sets the time  related to the last tackle of this object. */
bool PlayerObject::setTimeTackle( Time time )
{
  m_timeTackle = time;
  return true;
}

/*! This method returns the time that is related to the last tackle
    of this object.  */
Time PlayerObject::getTimeTackle( ) const
{
  return m_timeTackle;
}


/*! This method sets the global direction in which the arm of this object
```

```
   has last been observed to point. The time related to this update is
   represented by time.  */
bool PlayerObject::setGlobalArm( AngDeg ang, Time time )
{
 m_angGlobalArm = ang;
 return setTimeGlobalArm( time );
}

/*! This method returns the global direction in which the arm of this object
   has last been observed to point.  */
AngDeg PlayerObject::getGlobalArm( ) const
{
 return m_angGlobalArm;
}

/*! This method sets the time that is related to the last time the arm
   of this object has been observed to point in the global direction returned
   by 'getGlobalArm'. */
bool PlayerObject::setTimeGlobalArm( Time time )
{
 m_timeGlobalArm = time;
 return true;
}

/*! This method returns the time that is related to the last time the arm of
   this object has been observed to point in the global direction returned
   by 'getGlobalArm'. */
Time PlayerObject::getTimeGlobalArm( ) const
{
 return m_timeGlobalArm;
}

/*************************************************************************
/
/********************* CLASS BALLOBJECT
*************************************/
/*************************************************************************
/

/*! This is the constructor for a BallObject. It does nothing except
   initializing the class. */
BallObject::BallObject():DynamicObject()
{

}
```

```
/*! This method prints information about the ball to the specified
    output stream

    \param os output stream to which information is written. */
void BallObject::show( ostream& os)
{
  char buf[MAX_TEAM_NAME_LENGTH];
  SoccerTypes::getObjectStr( buf, objectType, DEFAULT_TEAM_NAME );
  os <<  buf
      << "r: "              << posRelative.getMagnitude()
      << " phi: "           << posRelative.getDirection()
      << " vel: "           << vecGlobalVelocity
      << " "                << timeGlobalVelocity
      << " rel: "           << posRelative
      << ", "               << timeRelativePosition
      << " global: "        << posGlobal
      << ", "               << timeGlobalPosition
      << ", changes: "      << dRelativeDistanceChange
      << ", "               << dRelativeAngleChange
      << " last seen:"      << timeLastSeen
      << " globalposderived: " << timeGlobalPosDerivedFromSee
      << "\n";
}

/*****************************************************************************
/
/******************** CLASS AGENTOBJECT
*************************************/
/*****************************************************************************
/

/*! This is the constructor for the class AgentObject and initializes the
    variables with the AgentObject. This the class that contains information
    about the agent itself.
    \param dStaminaMax maximum stamina for this agent (default 4000.0) */
AgentObject::AgentObject( double dStaminaMax ):PlayerObject( )
{
  viewAngle           = VA_NORMAL;
  viewQuality         = VQ_HIGH;

  stamina.setStamina ( dStaminaMax );
  stamina.setEffort  ( 1.0 );
  stamina.setRecovery( 1.0 );
  velSpeedRelToNeck.setVecPosition( 0.0, 0.0 );

  angGlobalNeckAngle   = UnknownAngleValue;
```

```
  angBodyAngleRelToNeck  = UnknownAngleValue;
}

/*! This methods prints the information about this AgentObject to the specified
    output stream. The default team name is used as the name.
    \param os output stream to which information is written. */
void AgentObject::show( ostream& os )
{
  show( DEFAULT_TEAM_NAME, os );
}

/*! This methods prints the information about this AgentObject to the specified
    output stream. The specified team name is used as the name
    \param strTeamName team name for this agent.
    \param os output stream to which information is written. */
void AgentObject::show( const char * strTeamName, ostream & os )
{
  char buf[MAX_TEAM_NAME_LENGTH];
  SoccerTypes::getObjectStr( buf, objectType, strTeamName );
  os   <<  buf
       << " (global("    << posGlobal
       << ", "           << timeGlobalPosition
       << ") (vel("      << vecGlobalVelocity
       << ", "           << timeGlobalVelocity
       << ") (angles(n:" << angGlobalNeckAngle
       << ", b:"         << angBodyAngleRelToNeck << ") ";
  stamina.show( os );
}

/*! This method returns the view angle of this PlayerObject.
    The view angle equals VA_NARROW, VA_NORMAL, VA_WIDE or VA_ILLEGAL.
    \return view angle of this PlayerObject */
ViewAngleT AgentObject::getViewAngle() const
{
  return viewAngle;
}

/*! This method returns the difference between the predicted global position
    of the agent and the actual derived global position. This difference can
    be used in determining the actual movement of other objects since the
    noise caused by the difference in the global position of the agent
    is then filtered out. */
VecPosition AgentObject::getPositionDifference() const
{
  return posPositionDifference;
}
```

```
/*! This method sets the position difference between the derived global
    position from the previous cycle information and the global position
    from the latest see message.
    \param p new position difference
    \return bool indicating whether the update was succesfull. */
bool  AgentObject::setPositionDifference( VecPosition p )
{
  posPositionDifference = p;
  return true;
}

/*! This method sets the view angle of this AgentObject.
    \param v new view angle (VA_NARROW, VA_NORMAL, VA_WIDE or VA_ILLEGAL)
    \return bool indicating whether value was set */
bool AgentObject::setViewAngle( ViewAngleT v )
{
  viewAngle = v;
  return true;
}

/*! This method returns the view quality of this AgentObject.
    The view angle equals VQ_LOW, VQ_HIGH, or VQ_ILLEGAL.
    \return view quality of this AgentObject */
ViewQualityT AgentObject::getViewQuality() const
{
  return viewQuality;
}

/*! Set the view quality of this AgentObject.
    \param v new view quality (VQ_LOW, VQ_HIGH, VQ_ILLEGAL)
    \return bool indicating whether value was set */
bool AgentObject::setViewQuality( ViewQualityT v )
{
  viewQuality = v;
  return true;
}

/*! This method returns the Stamina of the AgentObject.
    \return stamina from the agent. */
Stamina AgentObject::getStamina( ) const
{
  return stamina;
}

/*! This method sets the stamina of this AgentObject.
```

```
    \param sta new stamina for this AgentObject
    \return bool indicating whether value was set */
bool AgentObject::setStamina( Stamina sta )
{
  stamina = sta;
  return true ;
}

/*! This method returns the velocity (speed and direction) of this AgentObject.
    This information is directly availablefrom the sense message, in which the
    speed factor and the angle of this speed (relative to the neck) are given.
    \return velocity agent relative to the neck. */
VecPosition AgentObject::getSpeedRelToNeck( ) const
{
  return velSpeedRelToNeck;
}

/*! This method sets the velocity (speed and direction) of this AgentObject.
    This information comes directly from the sense message.
    \param v new velocity for this agentobject
    \return bool indicating whether value was set */
bool AgentObject::setSpeedRelToNeck( VecPosition v )
{
  velSpeedRelToNeck = v;
  return true;
}

/*! This method sets the global neck angle for this AgentObject.
    \param ang value for the global neck angle.
    \return bool indicating whether value was set */
bool AgentObject::setGlobalNeckAngle( AngDeg ang )
{
  angGlobalNeckAngle= ang;
  angGlobalBodyAngle= VecPosition::normalizeAngle(getBodyAngleRelToNeck()+ang);
  return true ;
}

/*! This method returns the relative angle of the body to the neck of this
    AgentObject.
    Example: global angle neck is 90 degrees and global body angle
    is 0, means that relative angle of body to neck is -90 degrees.
    \return relative body angle to the neck */
AngDeg AgentObject::getBodyAngleRelToNeck( ) const
{
  return angBodyAngleRelToNeck;
}
```

```
/*! This method sets the relative body angle to the neck for this AgentObject.
    \param ang relative body angle to the neck
    \return bool indicating whether value was set */
bool AgentObject::setBodyAngleRelToNeck( AngDeg ang )
{
  angBodyAngleRelToNeck = ang;
//  angGlobalBodyAngle = VecPosition::normalizeAngle(getGlobalNeckAngle()+ang);
  return true;
}

/*! This method sets whether it is allowed to move the arm of the agent. */
bool AgentObject::getArmMovable( ) const
{
  return m_bArmMovable;
}

/*! This method returns whether it is allowed to move the arm of the agent. */
bool AgentObject::setArmMovable( bool b )
{
  m_bArmMovable = b;
  return true;
}

/*! This method returns the numer of cycles it will take the arm to expire. In
    case of zero, the arm doesn't point anymore and the other players won't see
    the arm pointing anymore. */
int AgentObject::getArmExpires( ) const
{
  return m_iArmExpires;
}

/*! This method sets the numer of cycles it will take the arm to expire. In
    case of zero, the arm doesn't point anymore and the other players won't see
    the arm pointing anymore. */
bool AgentObject::setArmExpires( int i )
{
  m_iArmExpires = i;
  return true;
}

/*! This method returns the global position to which the arm points. If
    'getArmExpires' returns zero, it means that the arm doesn't point anymore
    and this value is invalid. */
VecPosition AgentObject::getGlobalArmPosition( ) const
{
```

```
  return m_posGlobalArm;
}

/*! This method sets the global position to which the arm points. */
bool AgentObject::setGlobalArmPosition ( VecPosition  v )
{
  m_posGlobalArm = v;
  return true;
}

/*! This method returns the number of cycles it will take the tackle to expire.
    In case of 0 a tackle command can be issued. */
int AgentObject::getTackleExpires( ) const
{
  return m_iTackleExpires;
}

/*! This method sets the number of cycles it will take the tackle to expire.
    In case of 0 a tackle command can be issued. */
bool AgentObject::setTackleExpires( int i )
{
  m_iTackleExpires = i;
  return true;
}


/*****************************************************************************
/
/********************* CLASS STAMINA
*****************************************/
/*****************************************************************************
/

/*! This is the constructor for this class. It sets the stamina, effort and
    recovery on the supplied values.
    \param dSta new stamina value (default 4000.0)
    \param dEff new effort value (default 1.0)
    \param dRec new recovery value (default 1.0)*/
Stamina::Stamina( double dSta, double dEff, double dRec )
{
  setStamina ( dSta );
  setEffort  ( dEff );
  setRecovery( dRec );
}

/*! This method prints the stamina values (stamina, effort and recovery) to the
```

```
    specified output stream.
    \param os output stream */
void Stamina::show( ostream & os)
{
  os  << "(sta:"   << m_dStamina
      << ", eff:"  << m_dEffort
      << ", rec: " << m_dRecovery << ")" << "\n";
}

/*! This method returns a TiredNessT enumeration that indicates how tired
    this associated agent with this Stamina value is. */
TiredNessT  Stamina::getTiredNess( double dRecDecThr, double dStaminaMax )
{
  double dStaDiffWithThr = getStamina() - dRecDecThr * dStaminaMax;
//  cerr << getStamina() << " " << dStaminaMax <<
//    " " << dRecDecThr << " " << dStaDiffWithThr << endl;
  if( dStaDiffWithThr < 100 )
    return TIREDNESS_TERRIBLE;
  else if( dStaDiffWithThr < 750 )
    return TIREDNESS_VERY_BAD;
  else if( dStaDiffWithThr < 1500 )
    return TIREDNESS_BAD;
  else if( dStaDiffWithThr < 2250 )
    return TIREDNESS_AVERAGE;

  return TIREDNESS_GOOD;
}

/*! This method returns the current stamina value.
    \return current stamina value (>0) */
double Stamina::getStamina() const
{
  return m_dStamina;
}

/*! This method sets the stamina value. This value should be positive,
    otherwise the value is set to 0 and false is returned.
    \param d new stamina value
    \return bool indicating whether value was set. */
bool Stamina::setStamina( double d )
{
  if( d < 0.0 )
  {
    m_dStamina = 0.0;
    return false;
  }
```

```
  else
    m_dStamina = d;
  return true;
}

/*! This method returns the effort. The effort denotes the percentage of
    the power in a dash that is actually used. Normally this is 1.0
    (100%), but when it comes below a threshold, it decreases. It will again
    rise when stamina becomes higher than a certain threshold defined in
    ServerSettings.
    \return effort value between 0 and 1 */
double Stamina::getEffort() const
{
  return m_dEffort;
}

/*! This method sets the effort value. This value should be between 0 and 1,
    otherwise the value is set to the closest value in this interval
    (0 for negative values, 1 for higher values) and false is returned.
    \param d new effort value (0..1)
    \return bool indicating whether value was set. */
bool Stamina::setEffort( double d )
{
  if( d < 0.0 )
  {
    m_dEffort = 0.0;
    return false;
  }
   else if( d > 1.0 )
  {
    m_dEffort = 1.0;
    return false;
  }
  else
    m_dEffort = d;
  return true;
}

/*! This method returns the recovery. Recovery denotes the percentage of
    the stamina increase that is added to the stamina every cycle. If recovery
    is 1.0 all of the increase of stamina is added to the current stamina.
    When stamina becomes below a certain threshold defined in ServerSettings,
    the recovery is decreased. It can never increase!
    \return recovery value between 0 and 1 */
double Stamina::getRecovery() const
{
```

```
  return m_dRecovery;
}

/*! This method sets the recovery value. This value should be between 0 and 1,
    otherwise the value is set to the closest value in this interval (0 for
    negative values, 1 for higher values) and false is returned.
    \param d new recovery value (0..1)
    \return bool indicating whether value was set. */
bool Stamina::setRecovery( double d )
{
  if( d < 0.0 )
  {
    m_dRecovery = 0.0;
    return false;
  }
   else if( d > 1.0 )
  {
    m_dRecovery = 1.0;
    return false;
  }
  else
    m_dRecovery = d;
  return true;
}

/*************************************************************************
/
/********************* TESTING PURPOSES
**************************************/
/*************************************************************************
/

/*
int main( void )
{
  PlayerObject p();
  BallObject b();
  FixedObject s();

  cout << p.getDeltaRelativeAngle() << "\n" <<
      b.getDeltaRelativeDistance() << "\n" <<
  p.getTimeRelativeDistance() << "\n";
  return 0;

}
*/
```

SoccerTypes.h

```
#ifndef _SOCCERTYPES_
#define _SOCCERTYPES_

#include <iostream>       // needed for output stream.
#include "Geometry.h"     // needed for AngDeg
#include "ServerSettings.h"


/*************************************************************************
/
/********************* DEFINES
*********************************************/
/*************************************************************************
/

#define MAX_TEAMMATES          11       /*!< Maximum number of teammates     */
#define MAX_OPPONENTS          11       /*!< Maximum number of opponents     */
#define MAX_HETERO_PLAYERS     7        /*!< Maximum number of hetero players*/
#define MAX_MSG              4096       /*!< maximum message size from server*/
#define MAX_SAY_MSG            10       /*!< maximum size of say message     */
#define MAX_TEAM_NAME_LENGTH   64       /*!< maximum length of teamname      */
#define MAX_FLAGS              55       /*!< maximum number of flags on field*/
#define MAX_LINES              4       /*!< maximum number of lines on field*/
#define DEFAULT_TEAM_NAME     "Team_L"  /*!< default teamname for own team   */
#define DEFAULT_OPPONENT_NAME "Team_R"  /*!< default teamname for opponent   */
#define PITCH_LENGTH        105.0       /*!< length of the pitch             */
#define PITCH_WIDTH          68.0       /*!< width  of the pitch             */
#define PITCH_MARGIN          5.0       /*!< margin next to the pitch        */
#define PENALTY_AREA_LENGTH  16.5       /*!< length of the penalty area      */
#define PENALTY_AREA_WIDTH   40.32      /*!< width of the penalty area       */
#define PENALTY_X (PITCH_LENGTH/2.0-PENALTY_AREA_LENGTH) /*!< penalty line
of
                                the opponent team               */


/*************************************************************************
/
/********************* CONSTANTS
*********************************************/
/*************************************************************************
/
const double  UnknownDoubleValue  = -1000.0; /*!< indicates unknown double   */
const AngDeg  UnknownAngleValue   = -1000.0; /*!< indicates unknown angle    */
const int     UnknownIntValue     = -1000;   /*!< indicates unknown int      */
const int     UnknownTime         = -20;     /*!< indicates unknown time     */
const long    UnknownMessageNr    = -30;     /*!< indicates unknown messagenr*/
```

```
/*****************************************************************************
/
/********************** ENUMERATIONS
*****************************************/
/*****************************************************************************
/

/*! ObjectT is an enumeration of all possible objects that are part of the
    RoboCup soccer simulation. The class SoccerTypes contains different methods
    to easily work with these objects and convert them to text strings and text
    strings to ObjectT. */
enum ObjectT { // don't change order
  OBJECT_BALL,           /*!< Ball                  */
  OBJECT_GOAL_L,         /*!< Left goal             */    // 2 goals
  OBJECT_GOAL_R,         /*!< Right goal            */
  OBJECT_GOAL_UNKNOWN,   /*!< Unknown goal          */
  OBJECT_LINE_L,         /*!< Left line             */    // 4 lines
  OBJECT_LINE_R,         /*!< Right line            */
  OBJECT_LINE_B,         /*!< Bottom line           */
  OBJECT_LINE_T,         /*!< Top line              */
  OBJECT_FLAG_L_T,       /*!< Flag left top         */   // 53 flags
  OBJECT_FLAG_T_L_50,    /*!< Flag top left 50      */
  OBJECT_FLAG_T_L_40,    /*!< Flag top left 40      */
  OBJECT_FLAG_T_L_30,    /*!< Flag top left 30      */
  OBJECT_FLAG_T_L_20,    /*!< Flag top left 20      */
  OBJECT_FLAG_T_L_10,    /*!< Flag top left 10      */
  OBJECT_FLAG_T_0,       /*!< Flag top left 0       */
  OBJECT_FLAG_C_T,       /*!< Flag top center       */
  OBJECT_FLAG_T_R_10,    /*!< Flag top right 10     */
  OBJECT_FLAG_T_R_20,    /*!< Flag top right 20     */
  OBJECT_FLAG_T_R_30,    /*!< Flag top right 30     */
  OBJECT_FLAG_T_R_40,    /*!< Flag top right 40     */
  OBJECT_FLAG_T_R_50,    /*!< Flag top right 50     */
  OBJECT_FLAG_R_T,       /*!< Flag right top        */
  OBJECT_FLAG_R_T_30,    /*!< Flag right top 30     */
  OBJECT_FLAG_R_T_20,    /*!< Flag right top 20     */
  OBJECT_FLAG_R_T_10,    /*!< Flag right top 10     */
  OBJECT_FLAG_G_R_T,     /*!< Flag goal right top   */
  OBJECT_FLAG_R_0,       /*!< Flag right 0          */
  OBJECT_FLAG_G_R_B,     /*!< Flag goal right bottom */
  OBJECT_FLAG_R_B_10,    /*!< Flag right bottom 10  */
  OBJECT_FLAG_R_B_20,    /*!< Flag right bottom 20  */
  OBJECT_FLAG_R_B_30,    /*!< Flag right bottom 30  */
  OBJECT_FLAG_R_B,       /*!< Flag right bottom     */
  OBJECT_FLAG_B_R_50,    /*!< Flag bottom right 50  */
```

```
OBJECT_FLAG_B_R_40,      /*!< Flag bottom right 40     */
OBJECT_FLAG_B_R_30,      /*!< Flag bottom right 30     */
OBJECT_FLAG_B_R_20,      /*!< Flag bottom right 20     */
OBJECT_FLAG_B_R_10,      /*!< Flag bottom right 10     */
OBJECT_FLAG_C_B,         /*!< Flag center bottom       */
OBJECT_FLAG_B_0,         /*!< Flag bottom 0            */
OBJECT_FLAG_B_L_10,      /*!< Flag bottom left 10      */
OBJECT_FLAG_B_L_20,      /*!< Flag bottom left 20      */
OBJECT_FLAG_B_L_30,      /*!< Flag bottom left 30      */
OBJECT_FLAG_B_L_40,      /*!< Flag bottom left 40      */
OBJECT_FLAG_B_L_50,      /*!< Flag bottom left 50      */
OBJECT_FLAG_L_B,         /*!< Flag left bottom         */
OBJECT_FLAG_L_B_30,      /*!< Flag left bottom 30      */
OBJECT_FLAG_L_B_20,      /*!< Flag left bottom 20      */
OBJECT_FLAG_L_B_10,      /*!< Flag left bottom 10      */
OBJECT_FLAG_G_L_B,       /*!< Flag goal left bottom    */
OBJECT_FLAG_L_0,         /*!< Flag left 0              */
OBJECT_FLAG_G_L_T,       /*!< Flag goal left top       */
OBJECT_FLAG_L_T_10,      /*!< Flag left bottom 10      */
OBJECT_FLAG_L_T_20,      /*!< Flag left bottom 20      */
OBJECT_FLAG_L_T_30,      /*!< Flag left bottom 30      */
OBJECT_FLAG_P_L_T,       /*!< Flag penaly left top     */
OBJECT_FLAG_P_L_C,       /*!< Flag penaly left center  */
OBJECT_FLAG_P_L_B,       /*!< Flag penaly left bottom  */
OBJECT_FLAG_P_R_T,       /*!< Flag penaly right top    */
OBJECT_FLAG_P_R_C,       /*!< Flag penaly right center */
OBJECT_FLAG_P_R_B,       /*!< Flag penaly right bottom */
OBJECT_FLAG_C,           /*!< Flag center field        */
OBJECT_TEAMMATE_1,       /*!< Teammate nr 1            */    // teammates 61
OBJECT_TEAMMATE_2,       /*!< Teammate nr 2            */
OBJECT_TEAMMATE_3,       /*!< Teammate nr 3            */
OBJECT_TEAMMATE_4,       /*!< Teammate nr 4            */
OBJECT_TEAMMATE_5,       /*!< Teammate nr 5            */
OBJECT_TEAMMATE_6,       /*!< Teammate nr 6            */
OBJECT_TEAMMATE_7,       /*!< Teammate nr 7            */
OBJECT_TEAMMATE_8,       /*!< Teammate nr 8            */
OBJECT_TEAMMATE_9,       /*!< Teammate nr 9            */
OBJECT_TEAMMATE_10,      /*!< Teammate nr 10           */
OBJECT_TEAMMATE_11,      /*!< Teammate nr 11           */
OBJECT_TEAMMATE_UNKNOWN, /*!< Teammate nr unkown       */
OBJECT_OPPONENT_1,       /*!< Opponent nr 1            */    // opponents 73
OBJECT_OPPONENT_2,       /*!< Opponent nr 2            */
OBJECT_OPPONENT_3,       /*!< Opponent nr 3            */
OBJECT_OPPONENT_4,       /*!< Opponent nr 4            */
OBJECT_OPPONENT_5,       /*!< Opponent nr 5            */
OBJECT_OPPONENT_6,       /*!< Opponent nr 6            */
```

```
  OBJECT_OPPONENT_7,      /*!< Opponent nr 7           */
  OBJECT_OPPONENT_8,      /*!< Opponent nr 8           */
  OBJECT_OPPONENT_9,      /*!< Opponent nr 9           */
  OBJECT_OPPONENT_10,     /*!< Opponent nr 10          */
  OBJECT_OPPONENT_11,     /*!< Opponent nr 11          */
  OBJECT_OPPONENT_UNKNOWN, /*!< Opponent nr unknown     */    // 84
  OBJECT_PLAYER_UNKNOWN,  /*!< Unknown player          */
  OBJECT_UNKNOWN,         /*!< Unknown object          */
  OBJECT_TEAMMATE_GOALIE,  /*!< Goalie of your side     */
  OBJECT_OPPONENT_GOALIE,  /*!< Goalie of opponent side  */
  OBJECT_ILLEGAL,         /*!< illegal object          */
  OBJECT_MAX_OBJECTS      /*!< maximum nr of objects    */ // 90
} ;

/*!The ObjectSetT enumerations holds the different object sets, which
   consists of one or multiple ObjectT types. */
enum ObjectSetT
{
  OBJECT_SET_TEAMMATES,       /*!< teammates                    */
  OBJECT_SET_OPPONENTS,       /*!< opponents                    */
  OBJECT_SET_PLAYERS,         /*!< players                      */
  OBJECT_SET_TEAMMATES_NO_GOALIE,/*!< teammates without the goalie  */
  OBJECT_SET_FLAGS,           /*!< flags                        */
  OBJECT_SET_LINES,           /*!< lines                        */
  OBJECT_SET_ILLEGAL          /*!< illegal                      */
} ;

/*!The PlayModeT enumeration contains all playmodes of the soccer playmode.
   The associated string values can be returned using the methods in the
   SoccerTypes class */
enum PlayModeT {
  PM_BEFORE_KICK_OFF,       /*!< before_kick_off:   before kick off        */
  PM_KICK_OFF_LEFT,         /*!< kick_off_l:        kick off for left team  */
  PM_KICK_OFF_RIGHT,        /*!< kick_off_r:        kick off for right team */
  PM_KICK_IN_LEFT,          /*!< kick_in_l:         kick in for left team   */
  PM_KICK_IN_RIGHT,         /*!< kick_in_r:         kick in for right team  */
  PM_CORNER_KICK_LEFT,      /*!< corner_kick_l:     corner kick left team   */
  PM_CORNER_KICK_RIGHT,     /*!< corner_kick_r:     corner kick right team  */
  PM_GOAL_KICK_LEFT,        /*!< goal_kick_l:       goal kick for left team */
  PM_GOAL_KICK_RIGHT,       /*!< goal_kick_r:       goal kick for right team*/
  PM_GOAL_LEFT,             /*!< goal_l:            goal scored by team left*/
  PM_GOAL_RIGHT,            /*!< goal_r:            goal scored by team righ*/
  PM_FREE_KICK_FAULT_LEFT,  /*!< free_kick_fault_l: free_kick to yourself   */
  PM_FREE_KICK_FAULT_RIGHT,         /*!< free_kick_fault_r: free_kick to self   */
  PM_FREE_KICK_LEFT,        /*!< free_kick_l:       free kick for left team */
  PM_FREE_KICK_RIGHT,       /*!< free_kick_r:       free kick for right team*/
```

```
  PM_INDIRECT_FREE_KICK_RIGHT,/*!<indirect_free_kick_r: ind. free kick right */
  PM_INDIRECT_FREE_KICK_LEFT,/*!< indirect_free_kick_l: ind. free kick left  */
  PM_BACK_PASS_LEFT,        /*!< back_pass_l:       left team passed back   */
  PM_BACK_PASS_RIGHT,       /*!< back_pass_r:       right team passed back  */
  PM_OFFSIDE_LEFT,          /*!< offside_l:         offside for left team   */
  PM_OFFSIDE_RIGHT,         /*!< offside_r:         offside for right team  */
  PM_PLAY_ON,               /*!< play_on:           play on (during match)  */
  PM_TIME_OVER,             /*!< time_over:         time over (after match) */
  PM_PENALTY_SETUP_LEFT,    /*!< penalty_setup_left left setups for penalty */
  PM_PENALTY_SETUP_RIGHT,   /*!< penalty_setup_right right setup for penalty*/
  PM_PENALTY_READY_LEFT,    /*!< penalty_ready_left ready for penalty l team*/
  PM_PENALTY_READY_RIGHT,   /*!< penalty_ready_right ready for pen r team   */
  PM_PENALTY_TAKEN_LEFT,    /*!< penalty_taken_left  penalty started left   */
  PM_PENALTY_TAKEN_RIGHT,   /*!< penalty_taken_right penalty started right  */
  PM_FROZEN,                /*!< game play is frozen                        */
  PM_QUIT,                  /*!< quit                                       */
  PM_ILLEGAL                /*!< unknown playmode                           */
} ;

/*! The RefereeT enumeration contains all messages that the referee can sent.
    The SoccerTypes class contains different methods to convert these messages
    to the corresponding playmode. */
enum RefereeMessageT {
  REFC_ILLEGAL,             /*!< unknown message                        */
  REFC_BEFORE_KICK_OFF,     /*!< before_kick_off:   before kick off     */
  REFC_KICK_OFF_LEFT,       /*!< kick_off_l:        kickoff for left team */
  REFC_KICK_OFF_RIGHT,      /*!< kick_off_r:        kickoff for right team*/
  REFC_KICK_IN_LEFT,        /*!< kick_in_l:         kick in for left team */
  REFC_KICK_IN_RIGHT,       /*!< kick_in_r:         kick in for right team*/
  REFC_CORNER_KICK_LEFT,    /*!< corner_kick_l:     corner kick left team */
  REFC_CORNER_KICK_RIGHT,   /*!< corner_kick_r:     corner kick right team*/
  REFC_GOAL_KICK_LEFT,      /*!< goal_kick_l:       goal kick left team   */
  REFC_GOAL_KICK_RIGHT,     /*!< goal_kick_r:       goal kick right team  */
  REFC_FREE_KICK_LEFT,      /*!< free_kick_l:       free kick left team   */
  REFC_FREE_KICK_RIGHT,     /*!< free_kick_r:       free kick right team  */
  REFC_INDIRECT_FREE_KICK_RIGHT,/*!< indirect_free_kick_r: ind freekick right*/
  REFC_INDIRECT_FREE_KICK_LEFT,/*!< indirect_free_kick_l: ind. free kick left*/
  REFC_FREE_KICK_FAULT_LEFT,  /*!< free_kick_fault_l: free_kick to yourself */
  REFC_FREE_KICK_FAULT_RIGHT, /*!< free_kick_fault_r: free_kick to yourself */
  REFC_BACK_PASS_LEFT,      /*!< back_pass_l:       left team passed back */
  REFC_BACK_PASS_RIGHT,     /*!< back_pass_r:       right team passed back*/
  REFC_PLAY_ON,             /*!< play_on:           play on (during match)*/
  REFC_TIME_OVER,           /*!< time_over:         time over(after match)*/
  REFC_FROZEN,              /*!< frozen:            game play is frozen   */
  REFC_QUIT,                /*!< quit:              quit                  */
  REFC_OFFSIDE_LEFT,        /*!< offside_l          offside left          */
```

```
  REFC_OFFSIDE_RIGHT,         /*!< offside_r          offside right         */
  REFC_HALF_TIME,             /*!< half_time:         it is half time       */
  REFC_TIME_UP,               /*!< time_up:           match has finished    */
  REFC_TIME_UP_WITHOUT_A_TEAM, /*!< time_up_without_a_team: match finished   */
  REFC_TIME_EXTENDED,         /*!< time_extended:     time cont. overtime   */
  REFC_FOUL_LEFT,             /*!< foul_l:            foul made by left     */
  REFC_FOUL_RIGHT,            /*!< foul_r:            foul made by right    */
  REFC_GOAL_LEFT,             /*!< goal_l:            goal made by left     */
  REFC_GOAL_RIGHT,            /*!< goal_r:            goal made by right    */
  REFC_DROP_BALL,             /*!< drop_ball:         ball is dropped       */
  REFC_GOALIE_CATCH_BALL_LEFT, /*!< goalie_catch_ball_l: left goalie catch   */
  REFC_GOALIE_CATCH_BALL_RIGHT,/*!< goalie_catch_ball_r: right goalie catch  */
  REFC_PENALTY_SETUP_LEFT,    /*!< penalty_setup_left left setup for penalty*/
  REFC_PENALTY_SETUP_RIGHT,   /*!< penalty_setup_right right setups for pen.*/
  REFC_PENALTY_READY_LEFT,    /*!< penalty_ready_left  ready for pen. l team*/
  REFC_PENALTY_READY_RIGHT,   /*!< penalty_ready_right ready for pen. r team*/
  REFC_PENALTY_TAKEN_LEFT,    /*!< penalty_ready_left  pen. taken by  l team*/
  REFC_PENALTY_TAKEN_RIGHT,   /*!< penalty_ready_right pen. taken by  r team*/
  REFC_PENALTY_MISS_LEFT,     /*!< penalty_miss_left   penalty missed r team*/
  REFC_PENALTY_MISS_RIGHT,    /*!< penalty_miss_right  penalty missed l team*/
  REFC_PENALTY_SCORE_LEFT,    /*!< penalty_score_left  penalty score l team */
  REFC_PENALTY_SCORE_RIGHT,   /*!< penalty_score_right penalty score r team */
  REFC_PENALTY_FOUL_LEFT,     /*!< penalty_foul_left   penalty foul l team  */
  REFC_PENALTY_FOUL_RIGHT,    /*!< penalty_foul_right  penalty foul r team  */
  REFC_PENALTY_ONFIELD_LEFT,  /*!< penalty_onfield_left pen. on left field  */
  REFC_PENALTY_ONFIELD_RIGHT, /*!< penalty_onfield_right pen. on right field*/
  REFC_PENALTY_WINNER_LEFT,   /*!< penalty_winner_l    penalty won by r team*/
  REFC_PENALTY_WINNER_RIGHT,  /*!< penalty_winner_r    penalty won by l team*/
  REFC_PENALTY_DRAW           /*!< penalty_draw        penalty result = draw*/
} ;

/*! The ViewAngleT enumeration contains the different view angles that are
    possible for a player */
enum ViewAngleT  {
  VA_NARROW,  /*!< view angle narrow  */
  VA_NORMAL,  /*!< view angle normal  */
  VA_WIDE,    /*!< view angle wide    */
  VA_ILLEGAL  /*!< illegal view angle */
};

/*!The ViewQualityT enumeration contains the different view qualities possible
   for a player. */
enum ViewQualityT {
  VQ_HIGH,    /*!< view quality high    */
  VQ_LOW,     /*!< view quality low     */
  VQ_ILLEGAL  /*!< illegal view quality */
```

```
};

/*!The SideT enumeration contains the two sides */
enum SideT {
  SIDE_LEFT,     /*!< left side    */
  SIDE_RIGHT,    /*!< right SIDE   */
  SIDE_ILLEGAL   /*!< illegal SIDE */
} ;

/*!The CommandT enumeration contains the different types for the SoccerCommand
   that are possible. */
enum CommandT {
  CMD_ILLEGAL,      /*!< illegal command */
  CMD_DASH,         /*!< dash command (player only)         */
  CMD_TURN,         /*!< turn command (player only)         */
  CMD_TURNNECK,     /*!< turn_neck command (player only)    */
  CMD_CHANGEVIEW,   /*!< change view command (player only)  */
  CMD_CATCH,        /*!< catch command (goalie only)        */
  CMD_KICK,         /*!< kick command (player only)         */
  CMD_MOVE,         /*!< move command                       */
  CMD_SENSEBODY,    /*!< sense_body command (player only)   */
  CMD_SAY,          /*!< say command                        */
  CMD_CHANGEPLAYER, /*!< change_player command (coach only) */
  CMD_ATTENTIONTO,  /*!< pay attention to specific player   */
  CMD_TACKLE,       /*!< tackle in current body direction   */
  CMD_POINTTO,      /*!< point arm towards a point on field */
  CMD_MAX_COMMANDS  /*!< maximum number of commands         */
} ;

/*!The PlayerT enumeration contains the different playertypes that are defined
   in a formation. This should not be confused with the later introducted
   player_types in soccerserver 7.xx that denotes the different possible
   heterogeneous players. A player type in the context PlayerT denotes the
   kind of player (midfielder, attacker) on the field. Its role on the pitch.*/
enum PlayerT {
  PT_ILLEGAL,            /*!< illegal player type */
  PT_GOALKEEPER,         /*!< goalkeeper          */
  PT_DEFENDER_CENTRAL,   /*!< central defender    */
  PT_DEFENDER_SWEEPER,   /*!< sweeper defender    */
  PT_DEFENDER_WING,      /*!< wing defender       */
  PT_MIDFIELDER_CENTER,  /*!< central midfielder  */
  PT_MIDFIELDER_WING,    /*!< wing midfielder     */
  PT_ATTACKER_WING,      /*!< wing attacker       */
  PT_ATTACKER,           /*!< central attacker    */
  MAX_PLAYER_TYPES
} ;
```

```
/*! The PlayerSetT enumeration contains different sets of playertypes that
    are defined in a formation. Possible sets are all midfielders, etc. */
enum PlayerSetT {
  PS_DEFENDERS,          /*!< all defenders       */
  PS_MIDFIELDERS,        /*!< all midfielders     */
  PS_ATTACKERS,          /*!< all attackers       */
  PS_ALL                 /*!< all players         */
} ;

/*!The FormationT enumeration contains the different formation types that
   are defined. */
enum FormationT {
  FT_ILLEGAL,            /*!< illegal formation type                */
  FT_INITIAL,            /*!< initial formation type (before kick off)*/
  FT_433_OFFENSIVE,      /*!< 433 offensive formation               */
  FT_334_OFFENSIVE,      /*!< 434 offensive formation               */
  FT_DEFENSIVE,          /*!< defensive formation type              */
  FT_OPEN_DEFENSIVE,     /*!< open defensive formation type         */
  FT_343_ATTACKING,      /*!< attacking formation type              */
  MAX_FORMATION_TYPES
} ;

/*!The BallStatus enumeration contains the status of the ball. This is returned
   when the coach has issued the check_ball message. */
enum BallStatusT {
  BS_ILLEGAL,          /*!< illegal ball status      */
  BS_IN_FIELD,         /*!< ball is in the field     */
  BS_GOAL_LEFT,        /*!< ball is in left goal     */
  BS_GOAL_RIGHT,       /*!< ball is in right goal    */
  BS_OUT_OF_FIELD      /*!< ball is not in the field */
} ;

/*!The ActionT enumeration contains different (high-level) actions. */
enum ActionT {
  ACT_ILLEGAL,                 /*!< illegal action (default)            */
  ACT_SEARCH_BALL,             /*!< search for the ball                 */
  ACT_KICK_BALL,               /*!< kick the ball                       */
  ACT_CATCH_BALL,              /*!< catch the ball                      */
  ACT_INTERCEPT,               /*!< intercept the ball                  */
  ACT_MARK,                    /*!< mark an opponent                    */
  ACT_TELEPORT_TO_STRATEGIC_POSITION, /*!< move to a strategic position(move)*/
  ACT_WATCH_BALL,              /*!< watch the ball                      */
  ACT_ANTICIPATE_BALL,         /*!< turn in anticipation for ball       */
  ACT_GOTO_STRATEGIC_POSITION,  /*!< go to a strategic position (dash)   */
  ACT_TURN_BODY_TO_CENTER,     /*!< turn body to center of field        */
```

```
  ACT_MOVE_TO_DEAD_BALL_POSITION, /*!< move to pos in dead ball situation    */
  ACT_INTERCEPT_SCORING_ATTEMPT,  /*!< intercept ball heading to goal        */
  ACT_DEFEND_GOALLINE,            /*!< defend the goalline (for goalkeeper   */
  ACT_TELEPORT_AFTER_CATCH,       /*!< teleport after catch (for goalkeeper) */
  ACT_TACKLE,                     /*!< tackle the ball                       */
  ACT_HOLD_BALL                   /*!< hold the ball                         */
} ;


/*! The MarkT enumeration contains different marking techniques. */
enum MarkT {
  MARK_ILLEGAL,    /*!< illegal marking                                    */
  MARK_GOAL,       /*!< mark goal (stand on obj-goal line)                 */
  MARK_BISECTOR,   /*!< mark bisector stand between goal,obj and ball,obj  */
  MARK_BALL        /*!< mark ball (stand on obj-ball line)                 */
} ;

/*! The DribbleT enumeration contains different dribble possibilities. */
enum DribbleT {
  DRIBBLE_ILLEGAL, /*!< illegal dribbling                                  */
  DRIBBLE_WITHBALL,/*!< dribble with ball very close                       */
  DRIBBLE_SLOW,    /*!< dribble slowly but kicking ball slightly ahead     */
  DRIBBLE_FAST     /*!< dribble fast by kicking ball up front              */
} ;

/*! The PassT enumeration contains different passing possibilities. */
enum PassT {
  PASS_ILLEGAL,    /*!< illegal pass                                       */
  PASS_FAST,       /*!< pass fast to a teammate                            */
  PASS_NORMAL      /*!< pass normal to a teammate                          */
} ;

/*! The ClearBallT enumeration contains different clear ball possibilities. */
enum ClearBallT {
  CLEAR_BALL_ILLEGAL,    /*!< illegal clear ball                           */
  CLEAR_BALL_OFFENSIVE,  /*!< clear ball to the front of the field         */
  CLEAR_BALL_DEFENSIVE,  /*!< clear ball to the center line of the field   */
  CLEAR_BALL_OFFENSIVE_SIDE, /*!< clear ball to front and side of the field */
  CLEAR_BALL_GOAL        /*!< clear ball to position in front of the goal  */
} ;

/*! The TiredNessT enumeration contains different values that indicate how
    tired an agent is. */
enum TiredNessT {
  TIREDNESS_ILLEGAL,     /*!< illegal tiredness value                     */
  TIREDNESS_GOOD,        /*!< player is not tired at all                  */
```

```
  TIREDNESS_AVERAGE,     /*!< average tiredness                        */
  TIREDNESS_BAD,         /*!< player starts to get tired               */
  TIREDNESS_VERY_BAD,    /*!< player is so tired it can hardly move    */
  TIREDNESS_TERRIBLE     /*!< player is so tired it cannot move        */
} ;

/*! The FeaturesT enumeration contains different features that can be saved.
    In this case, features represent specific information concerning the
    current state. When specific information is calculated once (e.g., the
    fastest opponent to the ball). This information can be stored. In the
    next request for this information, the stored result is immediately
    returned. */
enum FeatureT {
  FEATURE_ILLEGAL,                /*!< illegal feature                 */
  FEATURE_FASTEST_OPPONENT_TO_BALL,  /*!< fastest opponent to the ball  */
  FEATURE_FASTEST_TEAMMATE_TO_BALL,  /*!< fastest teammate to the ball  */
  FEATURE_FASTEST_PLAYER_TO_BALL,    /*!< fastest player to the ball    */
  FEATURE_FASTEST_TEAMMATE_TO_BALL_NO_GOALIE, /*!< fastest teammate, no
goali*/
  FEATURE_INTERCEPTION_POINT_BALL,   /*!< interception point ball      */
  FEATURE_INTERCEPT_CLOSE,           /*!< close interception point ball */
  FEATURE_INTERCEPT_CYCLES_ME,       /*!< nr of cycles for me to intercept */
  FEATURE_BEST_SCORING_POINT,        /*!< best scoring point in the goal */
  MAX_FEATURES
} ;

/*! The DirectionT enumeration contains the different directions */
enum DirectionT {
  DIR_ILLEGAL,              /*!< illegal message                    */
  DIR_NORTH,                /*!< north direction                    */
  DIR_NORTHWEST,            /*!< north west direction               */
  DIR_NORTHEAST,            /*!< north east direction               */
  DIR_CENTER,               /*!< center direction                   */
  DIR_EAST,                 /*!< east direction                     */
  DIR_WEST,                 /*!< west direction                     */
  DIR_SOUTHWEST,            /*!< south west direction               */
  DIR_SOUTHEAST,            /*!< south east direction               */
  DIR_SOUTH,                /*!< south direction                    */
  DIR_MAX                   /*!< number of directions               */
} ;

/*! The SucceedT enumeration contains the different succeed rate probabilites*/
enum SucceedT  {
  SUCCEED_ILLEGAL,          /*!< illegal message                    */
  SUCCEED_ALWAYS,           /*!< wil always succeed                 */
  SUCCEED_DOUBTFUL,         /*!< in some occassions it may succeed  */
```

```
  SUCCEED_NEVER              /*!< this will never succeed              */
} ;


/*****************************************************************************
/
/********************** CLASS TIME   ***************************************/
/*****************************************************************************
/

/*! This class contains the time representation of the soccer server.
    It is represented by an ordered pair (t,s) where t denotes the current
    server cycle and s is the number of cycles since the clock has stopped.
    Here the value for t equals that of the time stamp contained
    in the last message received from the server, whereas the value $s$ will
    always be 0 while the game is in progress. It is only during certain dead
    ball situations (e.g. an offside call leading to a free kick) that this
    value will be different, since in these cases the server time will stop
    while cycles continue to pass (i.e. actions can still be performed).
    Representing the time in this way has the advantage that it allows the
    players to reason about the number of cycles between events in a meaningful
    way. */
class Time
{
  int m_iTime;           /*!< Number of cycles, denoting the time */
  int m_iStopped;        /*!< Number of cycles stopped at m_iTime */

public:
        Time            ( int     iTime = -1, int iStopped = 0 );
  bool   updateTime       ( int     iTime                      );
  bool   setTimeStopped   ( int     iTime                       );
  int    getTime          (                                    );
  int    getTimeStopped   (                                     );
  int    getTimeDifference( Time    t                          );
  bool   isStopped        (                                    );
  Time   getTimeAddedWith ( int     iCycles                    );
  bool   addToTime        ( int     iCycles                    );
  void   show             ( ostream &os = cout                 );

  // overloaded arithmetic operators
  Time   operator +       ( const int  &i );
  Time   operator +       ( Time t );
  Time   operator -       ( const int  &i );
  int    operator -       ( Time t );
  void   operator =       ( const int  &i );
  void   operator +=      ( Time t );
```

```
  void   operator +=     ( const int  &i );
  void   operator -=     ( Time t );
  void   operator -=     ( const int  &i );
  bool   operator !=     ( Time t );
  bool   operator !=     ( const int  &i );
  bool   operator ==     ( Time t );
  bool   operator ==     ( const int  &i );
  bool   operator <      ( Time t );
  bool   operator <      ( const int  &i );
  bool   operator <=     ( Time t );
  bool   operator <=     ( const int  &i );
  bool   operator >      ( Time t );
  bool   operator >      ( const int  &i );
  bool   operator >=     ( Time t );
  bool   operator >=     ( const int  &i );


  // methods for producing output
  friend ostream&   operator << ( ostream &os, Time t );
} ;


/*****************************************************************************
/
/********************* CLASS SOCCERCOMMAND
************************************/
/*****************************************************************************
/

/*!This class resembles a SoccerCommand that contains all the information about
   a command that can be sent to the server, but the format is independent from
   the server implementation. A SoccerCommand can be created and before it is
   sent to the server, be converted to the actual string representation
   understood by the server. */
class SoccerCommand
{
  ServerSettings *SS; /*!< ServerSettings are used to check ranges of command*/

  // private methods to generate text string to sent to server
  bool  makeCatchCommand       ( char *str );
  bool  makeChangeViewCommand  ( char *str );
  bool  makeDashCommand        ( char *str );
  bool  makeKickCommand        ( char *str );
  bool  makeMoveCommand        ( char *str );
  bool  makeSayCommand         ( char *str );
  bool  makeSenseBodyCommand   ( char *str );
```

```
  bool  makeTurnCommand       ( char *str );
  bool  makeTurnNeckCommand   ( char *str );
  bool  makeChangePlayerCommand( char *str );
  bool  makeAttentionToCommand ( char *str );
  bool  makeTackleCommand     ( char *str );
  bool  makePointToCommand    ( char *str );

public:

  // different variables that are used by the different possible commands
  // only the variables that are related to the current commandType have
  // legal values
  Time        time;       /*!< command time, will be set by worldmodel    */
  CommandT    commandType; /*!< type of this command                       */
  double      dAngle;     /*!< angle of this command (for turn,turn_neck)  */
  double      dPower;     /*!< power of this command (for kick,dash)       */
  ViewQualityT vq;        /*!< view quality (for change_view)              */
  ViewAngleT   va;        /*!< view angle (for change_view)                */
  double      dX;         /*!< x coordinate (for move)                     */
  double      dY;         /*!< y coordinate (for move)                     */
  char        str[MAX_SAY_MSG];/*!< str (for say)                          */
  int         iTimes;     /*!< how many cycles will a command  be sent     */

  SoccerCommand( CommandT com = CMD_ILLEGAL, double d1=UnknownDoubleValue,
                                 double d2=UnknownDoubleValue,
                                 double d3=UnknownDoubleValue );
  SoccerCommand( CommandT com,               char   *msg                 );

  // command to set the different values of the SoccerCommand
  void makeCommand( CommandT com, double     d1 = UnknownDoubleValue,
                            double     d2 = UnknownDoubleValue,
                            double     d3 = UnknownDoubleValue      );
  void makeCommand( CommandT com, ViewAngleT v,    ViewQualityT q         );
  void makeCommand( CommandT com, char       *msg                         );

  bool isIllegal  (                                                       );

  void show       ( ostream& os                                           );

  // used to return the string representation of this SoccerCommand
  bool getCommandString( char *str,         ServerSettings *ss           );
} ;

/*****************************************************************************
/
```

```
/********************* CLASS FEATURE
*****************************************/
/*************************************************************************
/

/*! This class contains information for one specific feature (e.g., fastest
    teammate to the ball. A feature corresponds to a specific time and is often
    related to a specific object and a specific value. Therefore, these values
    are stored in this class. */
class Feature
{
  Time         m_timeSee;  /*!< see time corresponding to stored information*/
  Time         m_timeSense;/*!< sense time corresponding to stored info     */
  Time         m_timeHear; /*!< hear time corresponding to stored info      */
  ObjectT      m_object;   /*!< object information stored with this feature */
  double       m_dInfo;    /*!< information stored with this feature        */
  VecPosition  m_vec;      /*!< information stored with this feature        */
  SoccerCommand m_soc;     /*!< command stored with this feature            */

public:
  // standard get and set methods
  Feature(                                                  );
  Feature(              Time         timeSee,
                        Time         timeSense,
                        Time         timeHear,
                        ObjectT      object,
                        double       dInfo = UnknownDoubleValue,
                        SoccerCommand soc   = SoccerCommand(CMD_ILLEGAL),
                        VecPosition  pos = VecPosition(0,0)          );
  bool         setFeature  ( Time         timeSee,
                        Time         timeSense,
                        Time         timeHear,
                        ObjectT      object,
                        double       dInfo = UnknownDoubleValue,
                        SoccerCommand soc   = SoccerCommand(CMD_ILLEGAL),
                        VecPosition  pos = VecPosition(0,0)          );
  bool         setTimeSee  ( Time         time                        );
  Time         getTimeSee  (                                          );
  bool         setTimeSense( Time         time                        );
  Time         getTimeSense(                                          );
  bool         setTimeHear ( Time         time                        );
  Time         getTimeHear (                                          );
  bool         setObject   ( ObjectT      obj                         );
  ObjectT      getObject   (                                          );
  bool         setInfo     ( double       d                           );
  double       getInfo     (                                         );
```

```
  bool        setVec     ( VecPosition   pos                    );
  VecPosition getVec     (                                      );
  bool        setCommand ( SoccerCommand soc                    );
  SoccerCommand getCommand (                                    );

} ;

/*****************************************************************************
/
/********************** CLASS SOCCERTYPES
************************************/
/*****************************************************************************
/

/*! The class SoccerTypes contains different methods to work with the
    different enumerations defined in SoccerTypes.h. It is possible to
    convert soccertypes to strings and strings to soccertypes. It is
    also possible to get more specific information about some of the
    soccertypes. All methods are static so it is possible to call the
    methods without instantiating the class. */
class SoccerTypes
{
 public:
  // methods that deal with the differen objects
  static char*          getObjectStr            ( char        *strBuf,
                                                  ObjectT     o,
                                                  const char  *strTe=NULL);
  static ObjectT        getObjectFromStr        ( char        **str,
                                                  bool        *isGoalie,
                                                  const char  *strTeam   );
  static bool           isInSet                 ( ObjectT     o,
                                                  ObjectSetT  o_s,
                                                  ObjectT     objectGoalie=
                                                    OBJECT_TEAMMATE_1    );
  static bool           isPlayerTypeInSet       ( PlayerT     p,
                                                                          PlayerSetT  p_s       );
  static bool           isFlag                  ( ObjectT     o         );
  static bool           isLine                  ( ObjectT     o         );
  static bool           isGoal                  ( ObjectT     o         );
  static ObjectT        getOwnGoal              ( SideT       s         );
  static ObjectT        getGoalOpponent         ( SideT       s         );
  static bool           isBall                  ( ObjectT     o         );
  static bool           isTeammate              ( ObjectT     o         );
  static bool           isOpponent              ( ObjectT     o         );
  static bool           isGoalie                ( ObjectT     o         );
  static bool           isPlayer                ( ObjectT     o         );
```

```
  static bool          isKnownPlayer          ( ObjectT     o        );
  static int           getIndex               ( ObjectT     o        );
  static ObjectT       getTeammateObjectFromIndex( int        iIndex    );
  static ObjectT       getOpponentObjectFromIndex( int        iIndex    );
  static VecPosition   getGlobalPositionFlag  ( ObjectT     o,
                                                SideT       s,
                                                double dGoalWidth =14.02);
  static AngDeg        getGlobalAngleLine     ( ObjectT     o,
                                                SideT       s        );

  // methods that deal with the differen play modes
  static PlayModeT     getPlayModeFromStr     ( char        *str     );
  static PlayModeT     getPlayModeFromRefereeMessage( RefereeMessageT rm   );
  static char*         getPlayModeStr         ( PlayModeT   p        );
  static char*         getRefereeMessageStr   ( RefereeMessageT r     );
  static RefereeMessageT getRefereeMessageFromStr  ( char        *str     );

  // methods that deal with the frequency of the visual information
  static char*         getViewAngleStr        ( ViewAngleT  v        );
  static ViewAngleT    getViewAngleFromStr    ( char        *str     );
  static AngDeg        getHalfViewAngleValue  ( ViewAngleT  va       );
  static char*         getViewQualityStr      ( ViewQualityT v       );
  static ViewQualityT  getViewQualityFromStr  ( char        *str     );

  // methods that deal with the commands
  static char*         getCommandStr          ( CommandT    com      );
  static bool          isPrimaryCommand       ( CommandT    com      );

  // methods that deal with the side information
  static char*         getSideStr             ( SideT       s        );
  static SideT         getSideFromStr         ( char*       str      );

  // methods that deal with the status of the ball.
  static char*         getBallStatusStr       ( BallStatusT  bs      );
  static BallStatusT   getBallStatusFromStr   ( char        *str     );

  static AngDeg        getAngleFromDirection  ( DirectionT  dir      );
} ;

#endif
```

SoccerTypes.cpp

```
#include <iostream>       // needed for outputsteam in showcerr
#include <stdio.h>        // needed for sprintf
#ifdef Solaris
  #include <strings.h>    // needed for strncmp
#else
  #include <string.h>     // needed for strncmp
#endif

#include "SoccerTypes.h"
#include "Parse.h"


/*****************************************************************************
/
/********************* CLASS TIME
*****************************************/
/*****************************************************************************
/

/*! This is the constructor for the Time class, it receives two arguments. The
    actual time and how long the time has stopped.
    \param iTime actual time
    \param iStopped number of cycles time stopped */
Time::Time( int iTime, int iStopped )
{
  m_iTime    = iTime;
  m_iStopped = iStopped;
}

/*! This method updates the time to 'iTime'. When the actual time was already
    'iTime' the current time is kept unchanged and the time stopped is raised
    with one. Otherwise the actual time is changed to 'iTime' and the stopped
    time is set to 0.
    \param iTime new time
    \return boolean indicating whether update was successful */
bool Time::updateTime( int iTime )
{
  if( m_iTime == iTime )
    m_iStopped++;
  else
  {
    m_iTime    = iTime;
    m_iStopped = 0;
  }
  return true;
}
```

```
/*! This methods sets the stopped time, which denotes the number of cycles time
    stood still.
    \param iTime new stopped time
    \return boolean indicating whether update was successful */
bool Time::setTimeStopped( int iTime )
{
  m_iStopped = iTime;
  return true;
}

/*! This method returns the actual time, that is the number of cycles that have
    passed.
    \return actual time. */
int Time::getTime( )
{
  return m_iTime;
}

/*! This method returns the time the time has stopped, that is the number of
    cycles the time stood on the current value.
    \return stopped time */
int Time::getTimeStopped( )
{
  return m_iStopped;
}

/*! This method returns the time difference between two time objects.
    \param t Time with which current time should be compared
    \return time difference */
int Time::getTimeDifference( Time t )
{
  if( getTime() < t.getTime() )
    return getTime() - t.getTime() - t.getTimeStopped();
  else if( getTime() == t.getTime() )
    return getTimeStopped() - t.getTimeStopped();
  else
    return getTime() - t.getTime();
}

/*! This method returns a boolean value indicating whether the time
    currently is stopped.

    \return boolean indicating whether the time currently is stopped */
bool Time::isStopped( )
{
```

```
  return m_iStopped != 0;
}

/*! This method returns a new time class denoting the time when
    'iCycles' are added to the current time.  There are different
    situations possible. When the added time is positive and the time
    stands still, the cycles are added to the stopped time, otherwise
    they are added to the actual time. When the added time is negative
    and the time stands still, the cycles are subtracted from the
    stopped time.  Otherwise the time is subtracted from the actual
    time.

    \param iCycles denotes the time that should be added (when negative
    subtracted) to the current time

    \return new time object with 'iCycles' added to the current time. 'iCycles'
    can be negative in which case a subtraction is performed. */
Time Time::getTimeAddedWith( int iCycles )
{
  int iTime    = getTime();
  int iStopped = getTimeStopped();

  if( iCycles > 0 )                       // add cycles
  {
    if( iStopped > 0 )                     // time stopped
      iStopped += iCycles;                   // add it to stopped cycles
    else
      iTime    += iCycles;                   // otherwise add to normal time
  }
  else                                  // subtract cycles
  {
    if( iStopped > 0 && iStopped >= iCycles)   // time stopped and enough time
      iStopped += iCycles;                   // subtract cycle (iCycles=neg)
    else if( iStopped > 0 )                // stopped but not enough time
    {
      iStopped = 0;                        // take as many as possible
      iCycles += iStopped;
      iTime   += iCycles;                  // and take rest from m_iTime
    }
    else                                // time not stopped
      iTime   += iCycles;                   // take all from m_iTime
    if( iTime < 0 )
      iTime = 0;                           // negative time not possible
  }
  return Time( iTime, iStopped );
}
```

```
/*! This method adds 'iCycles' to the current time. The current values are
    updated. The method getTimeAddedWith is used to calculated the new time.
    \param iCycles time added to the current time
    \return boolean indicating whether update was successful */
bool Time::addToTime( int iCycles )
{
  *this = getTimeAddedWith( iCycles );
  return true;
}

/*! This method prints the time to the specified output stream. Time is printed
    as the two tuple (t,s) where t denotes the actual time and s the number of
    stopped cycles.
    \param os output stream to which output is written (default cout) */
void Time::show( ostream &os )
{
  os << "(" << getTime() << "," << getTimeStopped() << ")";
}

/*! This method returns the time as if 'i' cycles would be added to the current
    time. The method getTimeAddedWith is used for this. No changes are made to
    the current object.
    \param iCycles denotes the time that should be added to the current time
    \return new time object with 'iCycles' added to the current time */
Time Time::operator + ( const int &i )
{
  return getTimeAddedWith( i );
}

/*! This method returns the result of adding time 't' to the current
    time. No changes are made to the current object.
    It is defined by (t1,s1) + (t2,s2) = (t1+t2,s2). The stopped time of the
    first time tuple is neglected, since this has already been passed.
    \param t Time object that should be added to the current time
    \return new time object with 't' added to the current time */
Time Time::operator + ( Time t )
{
  return Time( getTime() + t.getTime(), t.getTimeStopped() );
}

/*! This method returns the time as if 'i' cycles would be subtracted from the
    current time. The method getTimeAddedWith is used for this. No changes are
    made to the current object.
    \param iCycles denotes the time that should be subtracted from the time
    \return new time object with 'iCycles' subracted  */
```

```
Time Time::operator - ( const int &i )
{
  return getTimeAddedWith( -i );
}

/*! This method returns the result the difference between the two times and
    is equal to the method getTimeDifference.
    \param t Time object that should be subtracted from the current time
    \return new time object with 't' subtracted from the current time */
int Time::operator - ( Time t )
{
  return getTimeDifference( t );
}

/*! This method returns a new time object (i,0). The argument 'i' is thus
    denoted as the actual time.
    \param i new time
    \return time object */
void Time::operator = ( const int &i )
{
  updateTime( i );
  setTimeStopped( 0 );
}

/*! This method updates the time by adding 'i' cycles to the current
    time. The method addToTime is used for this.
    \param i denotes the time that should be added to the current time */
void Time::operator += ( const int &i )
{
  addToTime( i );
}

/*! This method updates the time by adding the time 't' to the current
    time.
    It is defined by (t1,s1) + (t2,s2) = (t1+t2,s2). The stopped time of the
    first time tuple is neglected, since this has already been passed.
    \param t Time object that should be added to the current time */
void Time::operator += ( Time t )
{
  updateTime    ( getTime() + t.getTime() );
  setTimeStopped( t.getTimeStopped()      );
}

/*! This method updates the time by subtractign 'i' cycles from the
    current time. The method addToTime is used for this with '-i' as
    its argument.
```

```
    \param i denotes the time that should be subtracted */
void Time::operator -= ( const int &i )
{
  addToTime( -i );
}

/*! This method updates the time by subtracting time 't' from the current
    time. It is defined by (t1,s1) + (t2,s2) = (t1-t2,0). The stopped time is
    set to zero.
    \param t Time object that should be subtracted from the current time */
void Time::operator -= ( Time t )
{
  updateTime    ( getTime() - t.getTime() );
  setTimeStopped( 0                       );
}


/*! This method returns a boolean indicating whether the current time
    is inequal to the time specified by the integer 'i'. 'i' is first
    converted to the time object (i,0). When the time difference
    returned by getTimeDifference between these two time objects is
    inequal to zero true is returned, false otherwise

    \param i actual time with which current time should be compared
    \return bool indicating whether current time equals specified time */
bool Time::operator != ( const int &i )
{
  return getTimeDifference( Time(i, 0) ) != 0;
}

/*! This method returns a boolean indicating whether the current time
    is inequal to the time t. When the time difference returned by
    getTimeDifference between these two time objects is inequal to
    zero true is returned, false otherwise.

    \param t time with which current time should be compared
    \return bool indicating whether current time equals specified time */
bool Time::operator != ( Time t )
{
  return getTimeDifference( t ) != 0;
}

/*! This method returns a boolean indicating whether the current time
    equals the time as specified by the integer 'i'. 'i' is first
    converted to the time object (i,0). When the time difference
    returned by getTimeDifference between these two time objects
```

```
    equals zero, true is returned, false otherwise

    \param i actual time with which current time should be compared
    \return bool indicating whether current time equals specified time */
bool Time::operator == ( const int &i )
{
  return !( *this != i );
}

/*! This method returns a boolean indicating whether the current time equals
    the time t. When the time difference returned by getTimeDifference between
    these two time objects equals zero, true is returned, false otherwise
    \param t time with which current time should be compared
    \return bool indicating whether current time equals specified time */
bool Time::operator == ( Time t )
{
  return !( *this != t );
}

/*! This method returns a boolean indicating whether the current time
    is smaller than the time t. When the time difference returned by
    getTimeDifference is smaller than zero, true is returned, false
    otherwise

    \param t time with which current time should be compared
    \return bool indicating whether current time is smaller than given time */
bool Time::operator <  ( Time t )
{
  return getTimeDifference( t ) < 0;
}

/*! This method returns a boolean indicating whether the current time
    is smaller than the time denoted by the integer 'i'. Herefore
    first a time object (i,0) is created.

    \param t time with which current time should be compared
    \return bool indicating whether current time is smaller than given time */
bool Time::operator <  ( const int  &i )
{
  return Time( i, 0 ) >= *this ;
}

/*! This method returns a boolean indicating whether the current time
    is smaller than or equal to the time t.

    \param t time with which current time should be compared
```

```
    \return bool indicating whether current time is smaller to or equal
     than 't' */
bool Time::operator <= ( Time t )
{
  return ( *this < t ) || (*this == t );
}

/*! This method returns a boolean indicating whether the current time
    is smaller than or equal to the time denoted by the integer
    'i'. Herefore first a time object (i,0) is created.

    \param t time with which current time should be compared
    \return bool indicating whether current time is smaller than or equal to
    't' */
bool Time::operator <= ( const int  &i )
{
  return ( *this < i ) || (*this == i );
}

/*! This method returns a boolean indicating whether the current time is larger
    than the time t, that is it is not smaller than or equal to 't'.
    \param t time with which current time should be compared
    \return bool indicating whether current time is larger than 't' */
bool Time::operator >  ( Time t )
{
  return !( *this <= t );
}

/*! This method returns a boolean indicating whether the current time
    is larger than the time denoted by the integer 'i'. Herefore first
    a time object (i,0) is created.

    \param t time with which current time should be compared

    \return bool indicating whether current time is larger than the
    given time*/
bool Time::operator >  ( const int  &i )
{
  return !( *this <= i );
}

/*! This method returns a boolean indicating whether the current time is larger
    than or equal to than the time t, that is it is not smaller than 't'.
    \param t time with which current time should be compared
    \return bool indicating whether current time is larger or equal than 't' */
bool Time::operator >= ( Time t )
```

```
{
  return !( *this < t );

}

/*! This method returns a boolean indicating whether the current time is larger
    than or equal to the time denoted by the integer 'i'. Herefore first a time
    object (i,0) is created.
    \param t time with which current time should be compared
    \return bool indicating whether current time is larger than or equal to the
     given time*/
bool Time::operator >= ( const int  &i )
{
  return !( *this < i );
}

/*! Overloaded version of the C++ output operator for a Time class. This
    operator makes it possible to use Time objects in output statements (e.g.
    cout << t). The current cycle and the stopped time are printed in the
    format (t1,t2).
    \param os output stream to which information should be written
    \param v a Time object which must be printed
    \return output stream containing (x,y) */
ostream& operator <<( ostream &os, Time t )
{
  return os << "(" << t.getTime() << "," << t.getTimeStopped() << ")";
}

/*************************************************************************
/
/********************* CLASS SOCCERCOMMAND
************************************/
/*************************************************************************
/


/*! This is a constructor for the SoccerCommand class. It creates a command
    using the passed arguments (with all default illegal values). Depending
    on the specified CommandT the parameters are used in different ways. See
    the method makeCommand for an explanation of these values.
    \param com commandType for this SoccerCommand
    \param d1 1st argument, meaning depends on com (default UnknownDoubleValue)
    \param d2 2nd argument, meaning depends on com (default UnknownDoubleValue)
    \param d3 3rd argument, meaning depends on com (default UnknownDoubleValue)
    \return SoccerCommand with the specified parameters.*/
SoccerCommand::SoccerCommand( CommandT com, double d1, double d2, double d3 )
```

```
{
  // first initialize variables
  commandType = com;
  dPower      = UnknownDoubleValue;
  dAngle      = UnknownDoubleValue;
  va          = VA_ILLEGAL;
  vq          = VQ_ILLEGAL;
  iTimes      = 1;
  strcpy( str, "\0" );
  if( com == CMD_CHANGEVIEW )
    makeCommand( commandType, (ViewAngleT)d1, (ViewQualityT)d2 );
  else if( com != CMD_SAY )
    makeCommand( commandType, d1, d2, d3 );
}

/*! This is a constructor for the SoccerCommand when the commandType is a say
    message.
    \param com commandType for this SoccerCommand (must be CMD_SAY).
    \param msg message for this SoccerCommand */
SoccerCommand::SoccerCommand( CommandT com, char *msg )
{
  makeCommand( com, msg )  ;
}

/*! This method create a SoccerCommand from the specified command type and the
    parameters. The parameters have a different meaning depending on the given
    command type. Not all command types are listed, since the other command
    types need different parameters. So see the other overloaded methods for
    that.
    - CMD_DASH:        d1 = power for dashing
    - CMD_TURN:        d1 = angle body is turned
    - CMD_TURNNECK     d1 = angle neck is turned
    - CMD_KICK         d1 = power for kick command, d2 = angle for kick
    - CMD_MOVE         d1 = x position, d2 = y position, d3 = body_angle
    - CMD_CATCH        d1 = catch angle
    - CMD_CHANGEPLAYER d1 = player number, d2 = nr of new player type
    - CMD_ATTENTIONTO  d1 = which team, d2 = player number
    - CMD_TACKLE       d1 = power of the tackle
    \param com command type specifying the kind of command
    \param d1 meaning depends on the specified command type (see above)
    \param d2 meaning depends on the specified command type (see above)
    \param d3 meaning depends on the specified command type (see above) */
void SoccerCommand::makeCommand( CommandT com, double d1,
                                 double d2, double d3 )
{
  // set the variables that have a meaning for the specified command type
```

```
  commandType = com;
  switch ( com )
  {
    case CMD_TURN:
    case CMD_TURNNECK:
    case CMD_CATCH:
      dAngle = d1;
      break;
    case CMD_DASH:
    case CMD_TACKLE:
      dPower = d1;
      break;
    case CMD_KICK:
      dPower = d1;
      dAngle = d2;
      break;
    case CMD_MOVE:
      dX = d1;
      dY = d2;
      if( d3 != UnknownDoubleValue )
        dAngle = d3;
      else
        dAngle = 0;
      break;
    case CMD_CHANGEPLAYER:
    case CMD_ATTENTIONTO:
    case CMD_POINTTO:
      dX = d1;
      dY = d2;
      break;
    default:
      break;
  }
}

/*! This method creates a SoccerCommand for the command type CMD_CHANGEVIEW.
    \param com command type specifying the kind of command
    \param v view angle for the change view command
    \param q view quality for the change view command */
void SoccerCommand::makeCommand( CommandT com, ViewAngleT v, ViewQualityT q )
{
  commandType = com;
  if( com == CMD_CHANGEVIEW )
  {
    va = (ViewAngleT)  v;
    vq = (ViewQualityT)q;
```

```
  }
}

/*! This method creates a command for the command type CMD_SAY that accepts a
    string as parameter.
    \param com command type specifying the kind of command.
    \param msg string message that is added to the say message. */
void SoccerCommand::makeCommand( CommandT com, char* msg )
{
  commandType = com;
  if( com == CMD_SAY )
     strcpy( str, msg );
}

/*! This method prints the current command to the specified output stream.
    \param os output stream to which information is printed. */
void SoccerCommand::show( ostream& os )
{
//  os << "time: " << time << " "; // can be illegal when not yet set in WM.

  switch( commandType )
  {
    case CMD_ILLEGAL:
      os << "illegal\n" ;
      break;
    case CMD_DASH:
      os << commandType << " " << dPower << "\n";
      break;
    case CMD_TURN:
    case CMD_TURNNECK:
      os << commandType << " " << dAngle << "\n";
      break;
    case CMD_CATCH:
      os << "catch " << dAngle << "\n";
      break;
    case CMD_KICK:
      os << commandType << " " << dPower << " " << dAngle << "\n";
      break;
    case CMD_MOVE:
      os << commandType << " " << dX << " " << dY << " " << dAngle << "\n";
      break;
    case CMD_SENSEBODY:
      os << "sense_body" << "\n";
      break;
    case CMD_SAY:
      os << "say " << str << "\n";
```

```
      break;
    case CMD_CHANGEPLAYER:
      os << "change_player " << (int)dX << " " << (int)dY << "\n";
      break;
    case CMD_ATTENTIONTO:
      os << "attentionto " << (int)dX << " " << (int)dY << "\n";
      break;
    case CMD_TACKLE:
      os << "tackle " << (int)dPower << "\n";
      break;
    case CMD_POINTTO:
      os << "pointto " << dX << " " << dY << "\n";
      break;
    default:
      os << "unknown" << "\n";
      break;
  }
return;

}

/*! This method returns whether this SoccerCommand is illegal, that is the
    SoccerCommand hasn't been filled yet. This means that no command would be
    performed when this command is sent to the server.
    \return bool indicating whether the current Command is illegal */
bool SoccerCommand::isIllegal( )
{
  return commandType == CMD_ILLEGAL;
}

/*! This method returns a command string that is understood by the
    server from a SoccerCommand. The resulting string is put in the
    second argument and returned by the method. A reference to
    ServerSettings is passed as the second argument to check whether
    the values in the SoccerCommand are legal.

    \param str resulting string (enough space for MAX_MSG should be allocated)
    \param ss reference to serversettings class.
    \return resulting boolean indicating whether error occurred or not */
bool SoccerCommand::getCommandString( char *str, ServerSettings *ss )
{
  SS = ss;
  bool b = true;
  switch( commandType )
  {
    case CMD_DASH:        b = makeDashCommand(         str ); break;
```

```
    case CMD_TURN:        b = makeTurnCommand(         str ); break;
    case CMD_TURNNECK:    b = makeTurnNeckCommand(     str ); break;
    case CMD_CHANGEVIEW:  b = makeChangeViewCommand(   str ); break;
    case CMD_CATCH:       b = makeCatchCommand(        str ); break;
    case CMD_KICK:        b = makeKickCommand(         str ); break;
    case CMD_MOVE:        b = makeMoveCommand(         str ); break;
    case CMD_SENSEBODY:   b = makeSenseBodyCommand(    str ); break;
    case CMD_SAY:         b = makeSayCommand(          str ); break;
    case CMD_CHANGEPLAYER:b = makeChangePlayerCommand( str ); break;
    case CMD_ATTENTIONTO: b = makeAttentionToCommand(  str ); break;
    case CMD_TACKLE:      b = makeTackleCommand(       str ); break;
    case CMD_POINTTO:     b = makePointToCommand(      str ); break;
    case CMD_ILLEGAL:     b = true; str[0] = '\0';           break;
    default:
      b = false;
      cerr << "(ActHandler::makeCommandString) Unkown command!"<<"\n";
   }
   if( b == false )                    // if string could not be created
     str[0] = '\0';                    // create the empty string

   return b;
}

/*! This method makes a catch command from a SoccerCommand and puts the result
    in str. Resulting string looks like: (catch dAngle). Enough space
    should be allocated for str.
    \param command SoccerCommand that is a catch command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeCatchCommand( char *str  )
{
  if( SS->getMinMoment( ) <= dAngle && dAngle <= SS->getMaxMoment( ) )
    sprintf( str, "(catch %d)", (int)dAngle );
  else
  {
    fprintf(stderr,
     "(SoccerCommand::makeCatchCommand) angle %f out of bounds\n",dAngle);
    return false;
  }
  return true;
}

/*! This method makes a change view command from a SoccerCommand and puts the
    result in str. Resulting string looks like: (change_view va vq).
    Enough space should be allocated for str.
    \param command SoccerCommand that is a change view command
```

```
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeChangeViewCommand( char *str  )
{
  if( va != VA_ILLEGAL && vq != VQ_ILLEGAL )
    sprintf( str,"(change_view %s %s)", SoccerTypes::getViewAngleStr  ( va ),
                                  SoccerTypes::getViewQualityStr( vq ) );
  else
  {
    fprintf( stderr,
       "(SoccerCommand::makeChangeViewCommand) wrong arguments %s %s\n",
                        SoccerTypes::getViewAngleStr  ( va ),
                        SoccerTypes::getViewQualityStr( vq )  );
    return false;
  }
  return true;
}

/*! This method makes a dash command from a SoccerCommand and puts the result
    in str. Resulting string looks like: (dash dPower). Enough space
    should be allocated for str.
    \param command SoccerCommand that is a dash command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeDashCommand( char *str  )
{
  if( SS->getMinPower() <= dPower && dPower <= SS->getMaxPower() )
    sprintf( str,"(dash %d)", (int)dPower );
  else
  {
    fprintf( stderr,
       "(SoccerCommand::makeDashCommand) power %d out of bounds (%d,%d)\n",
       (int)dPower, SS->getMinPower(), SS->getMaxPower() );
    dPower = 0.0;
    sprintf( str, "(dash 0)" );
    return false;
  }
  return true;
}

/*! This method makes a kick command from a SoccerCommand and puts the result
    in str. Resulting string looks like: (kick dPower dAngle).
    Enough space should be allocated for str.
    \param command SoccerCommand that is a kick command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
```

```
bool SoccerCommand::makeKickCommand( char *str  )
{
 if( SS->getMinPower( )  <= dPower && dPower <= SS->getMaxPower( ) &&
     SS->getMinMoment( ) <= dAngle && dAngle <= SS->getMaxMoment( ) )
   sprintf( str,"(kick %d %d)", (int)dPower, (int)dAngle );
 else
 {
   fprintf(stderr,
         "(SoccerCommand::makeKickCommand) one argument %d or %d is wrong\n",
               (int)dPower, (int)dAngle );
   return false;
 }
 return true;
}

/*! This method makes a move command from a SoccerCommand and puts the result
    in str. Resulting string looks like: (move dX dY).
    Enough space should be allocated for str.
    \param command SoccerCommand that is a move command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeMoveCommand( char *str  )
{
 if( -PITCH_LENGTH/2 - PITCH_MARGIN <= dX &&
      PITCH_LENGTH/2 + PITCH_MARGIN >= dX &&
     -PITCH_WIDTH/2  - PITCH_MARGIN <= dY &&
      PITCH_WIDTH/2  + PITCH_MARGIN >= dY   )
   sprintf( str,"(move %d %d)", (int)dX, (int)dY);
 else
 {
   fprintf( stderr,
        "(SoccerCommand::makeMoveCommand) one argument %d or %d is wrong\n",
              (int)dX, (int)dY) ;
   return false;
 }
 return true;
}

/*! This method makes a say command from a SoccerCommand and puts the result
    in str. Resulting string looks like: (say str). Enough space
    should be allocated for str.
    \param command SoccerCommand that is a say command
    \param str_com string that will be filled with corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeSayCommand( char *str_com  )
{
```

```
  if( str != NULL && str[0] != '\0' )
    sprintf( str_com, "(say \"%s\")", str );
  else
  {
    fprintf( stderr, "(SoccerCommand::makeSayCommand) no string filled in\n" );
    return false;
  }
  return true;
}

/*! This method makes a sense_body command from a SoccerCommand and puts the
    result in str. Resulting string looks like: (sense_body). Enough space
    should be allocated for str.
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeSenseBodyCommand( char *str  )
{
  sprintf( str,"(sense_body)" );
  return true;
}

/*! This method makes a turn command from a SoccerCommand and puts the result
    in str. Resulting string looks like: (turn dAngle). Enough space
    should be allocated for str.
    \param command SoccerCommand that is a turn command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeTurnCommand( char *str  )
{
  if( SS->getMinMoment( ) <= dAngle && dAngle <= SS->getMaxMoment( ) )
    sprintf( str,"(turn %d)", (int)dAngle );
  else
  {
    fprintf( stderr,
      "(SoccerCommand::makeTurnCommand) argument %d incorrect (%d, %d)\n",
      (int)dAngle, SS->getMinMoment( ), SS->getMaxMoment( ) );
    dAngle = 0.0;
    sprintf( str, "(turn 0)" );
    return false;
  }
  return true;
}

/*! This method makes a turn_neck command from a SoccerCommand and puts the
    result in str. Resulting string looks like: (turn_neck dAngle).
    Enough space should be allocated for str.
```

```
    \param command SoccerCommand that is a turn_neck command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeTurnNeckCommand( char *str  )
{
  if( SS->getMinNeckMoment( ) <= (int)dAngle &&
      (int)dAngle             <= SS->getMaxNeckMoment( ) )
    sprintf( str,"(turn_neck %d)", (int)dAngle );
  else
  {
    fprintf( stderr,
       "(SoccerCommand::makeTurnNeckCommand) argument %d is wrong\n",
                                        (int)dAngle );
    dAngle = 0.0;
    sprintf( str, "(turn_neck 0)" );
    return false;
  }
  return true;
}

/*! This method makes a change_player_type command from a SoccerCommand and
    puts the result in str. Resulting string looks like:
    (change_player_type dX dY). Where dX stands for the
    teammate that should be changed and dY for the heterogenous player that
    it should become.
    Enough space should be allocated for str.
    \param command SoccerCommand that is a turn_neck command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeChangePlayerCommand( char *str  )
{
  if( (int)dX > 0  && (int)dX <= MAX_TEAMMATES &&
      (int)dY >= 0 && (int)dY < MAX_HETERO_PLAYERS  )
    sprintf( str,"(change_player_type %d %d)", (int)dX, (int)dY );
  else
  {
    fprintf( stderr,
    "(SoccerCommand::makeChangePlayerCommand) argument %d or %d is wrong\n",
                                     (int)dX, (int)dY  );
    return false;
  }
  return true;
}

/*! This method makes a attentionto command from a SoccerCommand and
    puts the result in str. Resulting string looks like:
```

```
    (attentionto opp|our dY). Where 'opp' is used when dX < 0 and
    'our' otherwise. dY stands for the player number of the team
    we want to pay attention to. When dY equals -1.0 the command
    (attentionto off) is created.
    Enough space should be allocated for str.
    \param command SoccerCommand that is a attentionto command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeAttentionToCommand( char *str  )
{
  char strTeam[10];
  if( dY < 0 )
  {
    sprintf( str, "(attentionto off)" );
    return true;
  }

  if( dX < 0 )
    strcpy( strTeam, "opp" );
  else
    strcpy( strTeam, "our" );

  if( (int)dY > 0  && (int)dY <= MAX_TEAMMATES )
    sprintf( str,"(attentionto %s %d)", strTeam, (int)dY );
  else
  {
    fprintf( stderr,
      "(SoccerCommand::makeAttentionToCommand) argument %s or %d is wrong\n",
                                    strTeam, (int)dY  );
    return false;
  }
  return true;
}

/*! This method makes a tackle command from a SoccerCommand and puts the result
    in str. Resulting string looks like: (tackle dPower). Enough space
    should be allocated for str.
    \param command SoccerCommand that is a tackle command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makeTackleCommand( char *str  )
{
  if( SS->getMinPower() <= dPower && dPower <= SS->getMaxPower() )
    sprintf( str,"(tackle %d)", (int)dPower );
  else
  {
```

```
    fprintf( stderr,
        "(SoccerCommand::makeTackleCommand) power %d out of bounds (%d,%d)\n",
        (int)dPower, SS->getMinPower(), SS->getMaxPower() );
    return false;
   }
  return true;
}

/*! This method makes a pointto command from a SoccerCommand and
    puts the result in str. Resulting string looks like:
    (pointto dist dir | off)). When dX is smaller than  -1.0 the command
    (pointto off) is created.
    Enough space should be allocated for str.
    \param command SoccerCommand that is a pointto command
    \param str string that will be filled with the corresponding SoccerCommand
    \return bool indicating whether string is filled or not  */
bool SoccerCommand::makePointToCommand( char *str  )
{
  if( dX < 0 )
  {
    sprintf( str, "(pointto off)" );
    return true;
  }

  if( dY >= SS->getMinMoment() && dY <= SS->getMaxMoment()  )
    sprintf( str,"(pointto %1.2f %1.2f)", dX, dY );
  else
  {
    fprintf( stderr,
     "(SoccerCommand::makePointToCommand) arg %f or %f is wrong\n", dX, dY  );
    return false;
  }
  return true;
}

/*****************************************************************************
/
/********************** CLASS FEATURE
*****************************************/
/*****************************************************************************
/

/*! This is the constructor for the Feature class. A feature (specific
    information that applies to the current time can be stored here. */
Feature::Feature( )
{
```

```
  setFeature( Time(UnknownTime, 0), Time(UnknownTime, 0),Time(UnknownTime, 0),
            OBJECT_ILLEGAL, UnknownDoubleValue );
}

/*! This is the constructor for the Feature class. A feature (specific
    information that applies to the current time can be stored here.
    For example, when the information about the fastest player to the ball
    is calculated, this can be stored in a feauture. This information can
    then be used for another calculation in the same cycle.
    \param timeSee time of see message to which this feature applies.
    \param timeSense time of sense message to which this feature applies.
    \param object object to which this feature applies
    \param dInfo specific information that can be stored about this feature.
    \param soc soccercommand stored with this feature
    \param pos position information stored with this feature */
Feature::Feature( Time timeSee, Time timeSense, Time timeHear, ObjectT object,
                  double dInfo, SoccerCommand soc, VecPosition pos )
{
  setFeature( timeSee, timeSense, timeHear, object, dInfo, soc, pos );
}

/*! This methods sets the values of the Feature class. A feature (specific
    information that applies to the current time can be stored here.
    For example, when the information about the fastest player to the ball
    is calculated, this can be stored in a feauture. This information can
    then be used for another calculation in the same cycle.
    \param timeSee time of see message to which this feature applies.
    \param timeSense time of sense message to which this feature applies.
    \param timeSense time of hear message to which this feature applies.
    \param object object to which this feature applies
    \param dInfo specific information that can be stored about this feature.
    \param soc command stored with this feature
    \param pos position information stored with this feature
    \return boolean indicating whether update was successful. */
bool Feature::setFeature( Time timeSee,  Time timeSense, Time timeHear,
                          ObjectT object, double  dInfo, SoccerCommand soc,
                          VecPosition pos )
{
  bool b;
  b  = setTimeSee( timeSee );
  b &= setTimeSense( timeSense );
  b &= setTimeHear( timeHear );
  b &= setObject( object );
  b &= setInfo( dInfo );
  b &= setCommand( soc );
  b &= setVec( pos );
```

```
  return b;
}

/*! This method sets the see time for a feature.
    \param time see time that applies to this feature. */
bool Feature::setTimeSee( Time time )
{
  m_timeSee = time;
  return true;
}

/*! This method returns the see time for a feature.
    \return time that is related to this feature. */
Time Feature::getTimeSee( )
{
  return m_timeSee;
}

/*! This method sets the sense time for a feature.
    \param time sense time that applies to this feature. */
bool Feature::setTimeSense( Time time )
{
  m_timeSense = time;
  return true;
}

/*! This method returns the sense time for a feature.
    \return time that is related to this feature. */
Time Feature::getTimeSense( )
{
  return m_timeSense;
}

/*! This method sets the hear time for a feature.
    \param time hear time that applies to this feature. */
bool Feature::setTimeHear( Time time )
{
  m_timeHear = time;
  return true;
}

/*! This method returns the hear time for a feature.
    \return time that is related to this feature. */
Time Feature::getTimeHear( )
{
  return m_timeHear;
```

```
}

/*! This method sets the object for a feature.
    \param object object type that applies to this feature. */
bool Feature::setObject ( ObjectT object )
{
  m_object = object;
  return true;
}

/*! This method returns the object related to this feature.
    \return object stored with this feature. */
ObjectT Feature::getObject( )
{
  return m_object;
}

/*! This method sets the information for a feature.
    \param d double information that applies to this feature. */
bool Feature::setInfo( double d )
{
  m_dInfo = d;
  return true;
}

/*! This method returns the information for a feature.
    \return information stored with this feature. */
double Feature::getInfo( )
{
  return m_dInfo;
}

/*! This method sets the posiiton corresponding to this feature.
    \return boolean indicating whether update was succesfull. */
bool Feature::setVec( VecPosition   pos )
{
  m_vec = pos;
  return true;
}

/*! This method returns the position information for a feature.
    \return position information stored with this feature. */
VecPosition Feature::getVec( )
{
  return m_vec;
}
```

```
/*! This method sets the command corresponding to this feature.
    \return boolean indicating whether update was succesfull. */
bool Feature::setCommand( SoccerCommand soc )
{
  m_soc = soc;
  return true;
}

/*! This method returns the command corresponding to this feature.
    \return command stored with this feature. */
SoccerCommand Feature::getCommand( )
{
  return m_soc;
}

/*****************************************************************************
/
/********************* CLASS SOCCERTYPES
*************************************/
/*****************************************************************************
/

/*! This constant defines the string names corresponding to the
    ObjectT names, defined in SoccerTypes.h. Players are not added
    since they depend on the team name. Note that the order is
    important, since the names are in the same order as the ObjectT
    enumeration. */
const char * ObjectNames[] =
{
"(b)", "(g l)", "(g r)", "(g ?)", "(l l)", "(l r)", "(l b)", "(l t)",
"(f l t)","(f t l 50)", "(f t l 40)", "(f t l 30)", "(f t l 20)",
"(f t l 10)", "(f t 0)","(f c t)","(f t r 10)", "(f t r 20)", "(f t r 30)",
"(f t r 40)","(f t r 50)", "(f r t)", "(f r t 30)", "(f r t 20)","(f r t 10)",
"(f g r t)" , "(f r 0)", "(f g r b)" , "(f r b 10)", "(f r b 20)",
"(f r b 30)", "(f r b)"   , "(f b r 50)", "(f b r 40)", "(f b r 30)",
"(f b r 20)", "(f b r 10)", "(f c b)"   , "(f b 0)"   , "(f b l 10)",
"(f b l 20)", "(f b l 30)", "(f b l 40)", "(f b l 50)", "(f l b)",
"(f l b 30)", "(f l b 20)", "(f l b 10)", "(f g l b)" , "(f l 0)",
"(f g l t)" , "(f l t 10)", "(f l t 20)", "(f l t 30)", "(f p l t)",
"(f p l c)", "(f p l b)",   "(f p r t)",  "(f p r c)",  "(f p r b)", "(f c)" };

/*! This method returns the string that corresponds to a specific object. This
    string name is exactly the same as the (short) name of the RoboCup
    Simulation.
    \param strBuf is the string in which the string representation is stored
```

```
    \param o ObjectT that has to be converted to the string representation
    \param strTeamName teamname that should be placed in case of player object
    \return pointer to strBuf, which contains the string representation */
char* SoccerTypes::getObjectStr( char* strBuf, ObjectT o,
                                 const char *strTeamName )
{
  if( o >= OBJECT_BALL && o <=   OBJECT_FLAG_C )
    sprintf( strBuf, ObjectNames[(int)o] );
  else if( isKnownPlayer( o ) && strTeamName != NULL )
    sprintf( strBuf, "(p %s %d)", strTeamName, getIndex( o ) + 1);
  else if( isKnownPlayer( o ) && isTeammate( o ) )
    sprintf( strBuf, "(p l %d)", getIndex( o ) + 1);
  else if( isKnownPlayer( o ) && isOpponent( o ) )
    sprintf( strBuf, "(p r %d)", getIndex( o ) + 1);
  else if( o == OBJECT_OPPONENT_UNKNOWN || o ==
OBJECT_TEAMMATE_UNKNOWN )
    sprintf( strBuf, "(p %s)", strTeamName );
  else if( o == OBJECT_PLAYER_UNKNOWN )
    sprintf( strBuf, "(p)" );
  else if( o == OBJECT_UNKNOWN )
    sprintf( strBuf, "(unknown)" );
  else
    sprintf( strBuf, "illegal: %d", (int)o );
  return strBuf;
}

/*! This method returns an ObjectT that corresponds to the string
    passed as the first argument. The string representation equals the
    representation used in the Soccer Server. Format is with
    parenthesis, so possible arguments for str are (ball), (p Team_L
    1), etc.

    \param str pointer to string containing string representation of object
    \param isGoalie bool representing the fact whether object is a goalie
    \param strMyTeamName in case of player or opponent object, own teamname
          has to be matched, when it matches it is teammate otherwise opponent

    \return return the corresponding ObjectT, OBJECT_ILLEGAL in case
    of error */
ObjectT SoccerTypes::getObjectFromStr( char** str, bool *isGoalie,
                                       const char* strMyTeamName )
{
  ObjectT o = OBJECT_ILLEGAL;
  char* ptrStr = *str;
  *isGoalie = false;
```

```
switch( ptrStr[1] )
{
 case 'b':                    // (ball)
 case 'B':                    // (B) in case of ball very close
   o = OBJECT_BALL; break;
 case 'G':
   o = OBJECT_GOAL_UNKNOWN;     // (G) in case of goal very close, ignored
  break;                       // (g l) or (g r) goal left or goal right
 case 'g': o = (ptrStr[3] == 'l') ? OBJECT_GOAL_L : OBJECT_GOAL_R; break;
 case 'l':                    // (l l), (l r), (l b) or (l t)
  switch( ptrStr[3] )
  {
   case 'l': o = OBJECT_LINE_L;  break;
   case 'r': o = OBJECT_LINE_R;  break;
   case 'b': o = OBJECT_LINE_B;  break;
   case 't': o = OBJECT_LINE_T;  break;
   default:  o = OBJECT_ILLEGAL; break;
  }
  break;
 case 'F':                    // (F) unkown flag very close.. ignored
  o = OBJECT_UNKNOWN; break;
 case 'f':                    // (f ...),  many options...
  switch( ptrStr[3] )
  {
   case 'l':                  // (f l ... lines on left part of field
    if( ptrStr[6] == ')' )    // only one character at index '5'
    {
     switch( ptrStr[5] )
     {
      case '0': o = OBJECT_FLAG_L_0; break; // (f l 0)
      case 't': o = OBJECT_FLAG_L_T; break; // (f l t)
      case 'b': o = OBJECT_FLAG_L_B; break; // (f l b)
      default:  o = OBJECT_ILLEGAL;  break;
     }
    }
    else                      // more than one character from index '5'
    {
     switch( ptrStr[7] )
     {
      case '1':               // (f l t 10) or (f l b 10)
        o = (ptrStr[5]=='t')? OBJECT_FLAG_L_T_10 :OBJECT_FLAG_L_B_10;
        break;
      case '2':               // (f l t 20) or (f l b 20)
        o = (ptrStr[5]=='t')? OBJECT_FLAG_L_T_20 :OBJECT_FLAG_L_B_20;
        break;
      case '3':               // (f l t 30) or (f l b 30)
```

```
        o = (ptrStr[5]=='t')? OBJECT_FLAG_L_T_30 :OBJECT_FLAG_L_B_30;
        break;
       default:
        o = OBJECT_ILLEGAL;
        break;
     }
    }
    break;
  case 'r':                  // (f r ... lines on right side of field
    if( ptrStr[6] == ')' )    // only one character at index '5'
    {
     switch( ptrStr[5] )
     {
      case '0': o = OBJECT_FLAG_R_0; break; // (f l 0)
      case 't': o = OBJECT_FLAG_R_T; break; // (f l t)
      case 'b': o = OBJECT_FLAG_R_B; break; // (f l b)
     }
    }
    else
    {
     switch( ptrStr[7] )     // more than one character from index '5'
     {
      case '1':
        o = (ptrStr[5]=='t')? OBJECT_FLAG_R_T_10 :OBJECT_FLAG_R_B_10;
        break;
      case '2':              // (f r t 10) or (f r b 10)
        o = (ptrStr[5]=='t')? OBJECT_FLAG_R_T_20 :OBJECT_FLAG_R_B_20;
        break;               // (f r t 20) or (f r b 20)
      case '3':
        o = (ptrStr[5]=='t')? OBJECT_FLAG_R_T_30 :OBJECT_FLAG_R_B_30;
        break;               // (f r t 30) or (f r b 30)
      default:
        o = OBJECT_ILLEGAL;
        break;
     }
    }
    break;
  case 't':                  // lines on top part of field
    if( ptrStr[5] == '0' )
     o = OBJECT_FLAG_T_0;    // (f t 0) center flag
    else
    {
     switch( ptrStr[7] )     // rest of the top flags
     {
      case '1':              // (f t l 10) or (f t r 10)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_T_L_10 : OBJECT_FLAG_T_R_10;
```

```
        break;
      case '2':           // (f t l 20) or (f t r 20)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_T_L_20 : OBJECT_FLAG_T_R_20;
        break;
      case '3':           // (f t l 30) or (f t r 30)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_T_L_30 : OBJECT_FLAG_T_R_30;
        break;
      case '4':           // (f t l 40) or (f t r 40)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_T_L_40 : OBJECT_FLAG_T_R_40;
        break;
      case '5':           // (f t l 50) or (f t r 50)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_T_L_50 : OBJECT_FLAG_T_R_50;
        break;
    }
   }
   break;
  case 'b':               // lines on bottom part of field
   if( ptrStr[5] == '0')
    o = OBJECT_FLAG_B_0;    // (f b 0) center flag
   else
   {
    switch( ptrStr[7] )     // rest of the bottom flags
    {
      case '1':           // (f b l 10) or (f b r 10)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_B_L_10 : OBJECT_FLAG_B_R_10;
        break;
      case '2':           // (f b l 20) or (f b r 20)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_B_L_20 : OBJECT_FLAG_B_R_20;
        break;
      case '3':           // (f b l 30) or (f b r 30)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_B_L_30 : OBJECT_FLAG_B_R_30;
        break;
      case '4':           // (f b l 40) or (f b r 40)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_B_L_40 : OBJECT_FLAG_B_R_40;
        break;
      case '5':           // (f b l 50) or (f b r 50)
        o = (ptrStr[5]=='l') ? OBJECT_FLAG_B_L_50 : OBJECT_FLAG_B_R_50;
        break;
    }
   }
   break;
  case 'c':               // center flags
   if( ptrStr[4] == ')' )
    o = OBJECT_FLAG_C;      // (f c) flag in midpoint field
   else                   // (f c t) or (f c b)
    o = (ptrStr[5] == 't') ? OBJECT_FLAG_C_T : OBJECT_FLAG_C_B;
```

```
      break;
    case 'g':                  // goal flags
      if( ptrStr[5] == 'l' )    // (g l t) or (g l b)
        o = (ptrStr[7] == 't') ? OBJECT_FLAG_G_L_T : OBJECT_FLAG_G_L_B;
      else                     // (g r t) or (g r b)
        o = (ptrStr[7] == 't') ? OBJECT_FLAG_G_R_T : OBJECT_FLAG_G_R_B;
      break;
    case 'p':                  // flags at sides penalty area
      switch( ptrStr[7] )
      {
        case 't':              // (p l t) or (p r t) top penalty area
          o = (ptrStr[5] == 'l') ? OBJECT_FLAG_P_L_T : OBJECT_FLAG_P_R_T;
          break;
        case 'c':              // (p l c) or (p r c) center penalty area
          o = (ptrStr[5] == 'l') ? OBJECT_FLAG_P_L_C : OBJECT_FLAG_P_R_C;
          break;
        case 'b':              // (p l b) or (p r b) bottom penalty area
          o = (ptrStr[5] == 'l') ? OBJECT_FLAG_P_L_B : OBJECT_FLAG_P_R_B;
          break;
        default:
          o = OBJECT_ILLEGAL;
          break;
      }
      break;
    default:
      o = OBJECT_ILLEGAL;
  }
  break; // end flags (finally)
case 'p': // (p team nr) or (p team) or (p) player teammate or opponent
case 'P': // or (P)
  ptrStr += 2;

  if( Parse::gotoFirstSpaceOrClosingBracket(&ptrStr) == ')' )
    o = OBJECT_PLAYER_UNKNOWN; // if (p) or (P) player is unknown.
  // check also with quotes since later versions use string around "teamname"
  else if( strncmp( ptrStr+1, strMyTeamName, strlen( strMyTeamName )) == 0 ||
           strncmp( ptrStr+2, strMyTeamName, strlen( strMyTeamName )) == 0 )
  {
    ptrStr++;
    if( Parse::gotoFirstSpaceOrClosingBracket(&ptrStr) == ' ' )
    {                                   // also team number
      switch( Parse::parseFirstInt( &ptrStr ) )     // get team number
      {
        case 1:  o = OBJECT_TEAMMATE_1;  break;     // and find team member
        case 2:  o = OBJECT_TEAMMATE_2;  break;
        case 3:  o = OBJECT_TEAMMATE_3;  break;
```

```
        case 4:  o = OBJECT_TEAMMATE_4;  break;
        case 5:  o = OBJECT_TEAMMATE_5;  break;
        case 6:  o = OBJECT_TEAMMATE_6;  break;
        case 7:  o = OBJECT_TEAMMATE_7;  break;
        case 8:  o = OBJECT_TEAMMATE_8;  break;
        case 9:  o = OBJECT_TEAMMATE_9;  break;
        case 10: o = OBJECT_TEAMMATE_10; break;
        case 11: o = OBJECT_TEAMMATE_11; break;
        default: o = OBJECT_ILLEGAL;
       }
       if( ptrStr[0] != ')' && ptrStr[1] == 'g' ) // goalie
         *isGoalie = true;
      }
      else
       o = OBJECT_TEAMMATE_UNKNOWN;                 // (p team) but no nr
    }
    else                                    // not a teammate
    {
     ptrStr++;
     if( Parse::gotoFirstSpaceOrClosingBracket( &ptrStr ) == ' ' )
     {                                      // also team number
      switch( Parse::parseFirstInt( &ptrStr ) )     // get team numer
      {
        case 1:  o = OBJECT_OPPONENT_1;  break;     // and return opponent
        case 2:  o = OBJECT_OPPONENT_2;  break;
        case 3:  o = OBJECT_OPPONENT_3;  break;
        case 4:  o = OBJECT_OPPONENT_4;  break;
        case 5:  o = OBJECT_OPPONENT_5;  break;
        case 6:  o = OBJECT_OPPONENT_6;  break;
        case 7:  o = OBJECT_OPPONENT_7;  break;
        case 8:  o = OBJECT_OPPONENT_8;  break;
        case 9:  o = OBJECT_OPPONENT_9;  break;
        case 10: o = OBJECT_OPPONENT_10; break;
        case 11: o = OBJECT_OPPONENT_11; break;
        default: o = OBJECT_ILLEGAL;
      }
      if( ptrStr[0] != ')' && ptrStr[1] == 'g' ) // goalie
        *isGoalie = true;
     }
     else
      o = OBJECT_OPPONENT_UNKNOWN;                 // number not known

    }
    break;
  default:
    cerr << "(SoccerTypes::getObjectFromStr) Unknown msg: " <<  ptrStr << "\n";
```

```
    o = OBJECT_ILLEGAL;
    break;
   }
  // go to the end of the object
  Parse::gotoFirstOccurenceOf( ')', &ptrStr );
  *str=ptrStr;
  return o;
}

/*! This method returns the index of an object relative to the first object
    in that set.
    The index is always 1 smaller than its number, so OBJECT_OPPONENT_1
    will become 0. This can be used for indexing an array of objects.
    \param o ObjectT type of object of which the index has to be calculated
    \return index of object or -1 when o was not a correct object */
int SoccerTypes::getIndex( ObjectT o )
{
  if( o >= OBJECT_OPPONENT_1 && o <= OBJECT_OPPONENT_11 )
    return o - OBJECT_OPPONENT_1;
  else if( o >= OBJECT_TEAMMATE_1 && o <= OBJECT_TEAMMATE_11 )
    return o - OBJECT_TEAMMATE_1;
  else if( o >= OBJECT_GOAL_L && o <= OBJECT_GOAL_R )
    return o - OBJECT_GOAL_L;
  else if( o >= OBJECT_FLAG_L_T && o <= OBJECT_FLAG_C )
    return o - OBJECT_FLAG_L_T + 2; // 2 added for the two goals
  else if( o >= OBJECT_LINE_L && o <= OBJECT_LINE_T )
    return o - OBJECT_LINE_L;
  else
    return -1;
}

/*! This method returns the object type of a teammate with index iIndex. When
    iIndex equals 3 for example OBJECT_TEAMMATE_4 is returned.
    \param iIndex index of teammate range is [0..10]
    \return object type corresponding to this index */
ObjectT SoccerTypes::getTeammateObjectFromIndex( int iIndex )
{
  return (ObjectT) ( OBJECT_TEAMMATE_1 + iIndex );
}

/*! This method returns the object type of an opponent with index iIndex. When
    iIndex equals 9 for example OBJECT_OPPONENT_10 is returned.
    \param iIndex index of opponent range is [0..10]
    \return object type corresponding to this index */
ObjectT SoccerTypes::getOpponentObjectFromIndex( int iIndex )
{
```

```
  return (ObjectT) ( OBJECT_OPPONENT_1 + iIndex );
}

/*! This method returns a boolean indicating whether the object o is
    part of the object set o_s. OBJECT_TEAMMATE_1 as o and
    OBJECT_SET_TEAMMATES will return for example the value true.

    \param o ObjectT of which should be checked whether it is a part of o_s
    \param o_s ObjectSet in which o should be
    \return true when o is included in set o_s, false otherwise */
bool SoccerTypes::isInSet( ObjectT o, ObjectSetT o_g, ObjectT objGoalie )
{
  switch( o_g )
  {
    case OBJECT_SET_TEAMMATES: return isTeammate( o ) && isKnownPlayer( o );
    case OBJECT_SET_OPPONENTS: return isOpponent( o ) && isKnownPlayer( o );
    case OBJECT_SET_PLAYERS:   return isPlayer  ( o ) && isKnownPlayer( o );
    case OBJECT_SET_FLAGS:     return isFlag    ( o );
    case OBJECT_SET_LINES:     return isLine    ( o );
    case OBJECT_SET_TEAMMATES_NO_GOALIE:
                          return isTeammate( o ) && isKnownPlayer( o ) &&
                               o != objGoalie;
    case OBJECT_SET_ILLEGAL:
    default:                  break;
  }
  return false;
}


/*! This method returns a boolean value which indicates whether a given player
    type belongs to a specific set. The set PS_DEFENDERS for example contains
    both the defender sweeper and the wing defenders. */
bool SoccerTypes::isPlayerTypeInSet( PlayerT p, PlayerSetT p_s )
{
  switch( p_s )
  {
    case PS_DEFENDERS:
      return p == PT_DEFENDER_CENTRAL || p == PT_DEFENDER_WING;
    case PS_MIDFIELDERS:
      return p == PT_MIDFIELDER_CENTER || p ==  PT_MIDFIELDER_WING;
    case PS_ATTACKERS:
      return p == PT_ATTACKER_WING || p == PT_ATTACKER;
    case PS_ALL:
      return true;
    default:
      break;
```

```
  }
  return false;

}


/*! This method determines whether object o is a flag.
    \param o an object type
    \return bool indicating whether o is a flag (return true) or not (false) */
bool SoccerTypes::isFlag( ObjectT o )
{
  return ( o >= OBJECT_FLAG_L_T && o <= OBJECT_FLAG_C ) ||
         ( o >= OBJECT_GOAL_L   && o <= OBJECT_GOAL_R        );
}

/*! This method determines whether object o is a line.
     \param o an object type
     \return bool indicating whether o is a line return true) or not (false) */
bool SoccerTypes::isLine( ObjectT o )
{
  return o >= OBJECT_LINE_L && o <= OBJECT_LINE_T;
}

/*! This method determines whether object o is a goal
    \param o an object type
    \return bool indicating whether o is a goal (return true) or not (false) */
bool SoccerTypes::isGoal( ObjectT o )
{
  return o == OBJECT_GOAL_L || o == OBJECT_GOAL_R;
}

/*! This method determines whether object o is a teammate
    \param o an object type
    \return bool indicating whether o is a teammate (true) or not (false) */
bool SoccerTypes::isTeammate( ObjectT o )
{
  return o >= OBJECT_TEAMMATE_1 && o <= OBJECT_TEAMMATE_UNKNOWN;
}

/*! This method determines whether object o is an opponent
    \param o an object type
    \return bool indicating whether o is an opponent (true) or not (false) */
bool SoccerTypes::isOpponent( ObjectT o )
{
  return o >= OBJECT_OPPONENT_1 && o <= OBJECT_OPPONENT_UNKNOWN;
}
```

```
/*! This method determines whether object o is a player without checking
    whether its number or side is available.
    \param o an object type

    \return bool indicating whether o is a known player (true) or not
    (false) */
bool SoccerTypes::isPlayer( ObjectT o )
{
  return isKnownPlayer( o )          || o == OBJECT_TEAMMATE_UNKNOWN ||
         o == OBJECT_OPPONENT_UNKNOWN || o == OBJECT_PLAYER_UNKNOWN;
}

/*! This method determines whether object o is a known player, thus
    containing a number
    \param o an object type

    \return bool indicating whether o is a known player (true) or not
    (false) */
bool SoccerTypes::isKnownPlayer( ObjectT o )
{
  return (o >= OBJECT_OPPONENT_1 && o <= OBJECT_OPPONENT_11) ||
         (o >= OBJECT_TEAMMATE_1 && o <= OBJECT_TEAMMATE_11);
}

/*! This method determines whether object o is a goalie = teammate number is 1
    (for now)
    \param o an object type
    \return bool indicating whether o is a goalie (true) or not (false) */
bool SoccerTypes::isGoalie( ObjectT o )
{
  return o == OBJECT_TEAMMATE_1 || o == OBJECT_OPPONENT_1;
}

/*! This method determines whether object o is the ball
    \param o an object type
    \return bool indicating whether o is the ball (true) or not (false) */
bool SoccerTypes::isBall( ObjectT o )
{
  return o == OBJECT_BALL;
}

/*! This method returns the object representing the own goal
    \param s own side
    \return object of the own goal, OBJECT_ILLEGAL when s is SIDE_ILLEGAL*/
ObjectT SoccerTypes::getOwnGoal( SideT s )
```

```
{
 if( SIDE_LEFT == s )
   return OBJECT_GOAL_L;
 else if( SIDE_RIGHT == s )
   return OBJECT_GOAL_R;

 cerr << "(SoccerTypes::isOwnGoal) Wrong side argument" << "\n";
 return OBJECT_ILLEGAL;
}

/*! This method returns the object representing the opponent goal
   \param s own side

   \return object of the goal opponent, OBJECT_ILLEGAL when s is
   SIDE_ILLEGAL*/
ObjectT SoccerTypes::getGoalOpponent( SideT s )
{
 if( SIDE_LEFT == s )
   return OBJECT_GOAL_R;
 else if( SIDE_RIGHT == s )
   return OBJECT_GOAL_L;

 cerr << "(SoccerTypes::isGoalOpponent) Wrong side argument" << "\n";
 return OBJECT_ILLEGAL;
}

/*! This method returns the global position on the field of a flag (a goal
   is also a flag). Since the global positions for both teams differ, the
   side of the agent team is also needed. Note that the global
   positions of the flags will not change in the second half.
   \param o flag of which global position should be determined
   \param s side of your team.
   \param dGoalWidth for some flags the goalWidth is necessary (default 14.02)
   \return VecPosition representing the global position. x and y value are
         both UnknownDoubleValue when o is not a flag or goal. */
VecPosition SoccerTypes::getGlobalPositionFlag( ObjectT o, SideT s,
                                   double dGoalWidth )
{
 VecPosition pos;
 if( !(isFlag(o) || isGoal(o)) )
    return VecPosition(UnknownDoubleValue, UnknownDoubleValue);
 switch( o ) // for every object the global position is entered
 {
   case OBJECT_GOAL_L:
     pos.setVecPosition( -PITCH_LENGTH/2.0, 0.0 );                break;
   case OBJECT_GOAL_R:
```

```
  pos.setVecPosition(  PITCH_LENGTH/2.0, 0.0 );                break;
case OBJECT_FLAG_L_T:
  pos.setVecPosition( -PITCH_LENGTH/2.0, -PITCH_WIDTH/2.0 );     break;
case OBJECT_FLAG_T_L_50:
  pos.setVecPosition( -50.0, -PITCH_WIDTH/2.0 - PITCH_MARGIN );  break;
case OBJECT_FLAG_T_L_40:
  pos.setVecPosition( -40.0, -PITCH_WIDTH/2.0 - PITCH_MARGIN );  break;
case OBJECT_FLAG_T_L_30:
  pos.setVecPosition( -30.0, -PITCH_WIDTH/2.0 - PITCH_MARGIN );  break;
case OBJECT_FLAG_T_L_20:
  pos.setVecPosition( -20.0, -PITCH_WIDTH/2.0 - PITCH_MARGIN );  break;
case OBJECT_FLAG_T_L_10:
  pos.setVecPosition( -10.0, -PITCH_WIDTH/2.0 - PITCH_MARGIN );  break;
case OBJECT_FLAG_T_0:
  pos.setVecPosition(   0.0, -PITCH_WIDTH/2.0 - PITCH_MARGIN );  break;
case OBJECT_FLAG_C_T:
  pos.setVecPosition(   0.0, -PITCH_WIDTH/2.0);                break;
case OBJECT_FLAG_T_R_10:
  pos.setVecPosition(  10.0,  -PITCH_WIDTH/2.0 - PITCH_MARGIN);  break;
case OBJECT_FLAG_T_R_20:
  pos.setVecPosition(  20.0,  -PITCH_WIDTH/2.0 - PITCH_MARGIN);  break;
case OBJECT_FLAG_T_R_30:
  pos.setVecPosition(  30.0,  -PITCH_WIDTH/2.0 - PITCH_MARGIN);  break;
case OBJECT_FLAG_T_R_40:
  pos.setVecPosition(  40.0,  -PITCH_WIDTH/2.0 - PITCH_MARGIN);  break;
case OBJECT_FLAG_T_R_50:
  pos.setVecPosition(  50.0,  -PITCH_WIDTH/2.0 - PITCH_MARGIN);  break;
case OBJECT_FLAG_R_T:
  pos.setVecPosition( PITCH_LENGTH/2.0,  -PITCH_WIDTH/2.0 );     break;
case OBJECT_FLAG_R_T_30:
  pos.setVecPosition( PITCH_LENGTH/2.0 + PITCH_MARGIN,  -30.0 ); break;
case OBJECT_FLAG_R_T_20:
  pos.setVecPosition( PITCH_LENGTH/2.0 + PITCH_MARGIN,  -20.0 ); break;
case OBJECT_FLAG_R_T_10:
  pos.setVecPosition( PITCH_LENGTH/2.0 + PITCH_MARGIN,  -10.0 ); break;
case OBJECT_FLAG_G_R_T:
  pos.setVecPosition( PITCH_LENGTH/2.0,  -dGoalWidth/2.0 );      break;
case OBJECT_FLAG_R_0:
  pos.setVecPosition( PITCH_LENGTH/2.0 + PITCH_MARGIN,  0.0 );   break;
case OBJECT_FLAG_G_R_B:
  pos.setVecPosition( PITCH_LENGTH/2.0,  dGoalWidth/2.0 );       break;
case OBJECT_FLAG_R_B_10:
  pos.setVecPosition( PITCH_LENGTH/2.0 + PITCH_MARGIN,  10.0 );  break;
case OBJECT_FLAG_R_B_20:
  pos.setVecPosition( PITCH_LENGTH/2.0 + PITCH_MARGIN,  20.0 );  break;
case OBJECT_FLAG_R_B_30:
```

```
  pos.setVecPosition( PITCH_LENGTH/2.0 + PITCH_MARGIN,  30.0 );  break;
case OBJECT_FLAG_R_B:
  pos.setVecPosition( PITCH_LENGTH/2.0,  PITCH_WIDTH/2.0 );      break;
case OBJECT_FLAG_B_R_50:
  pos.setVecPosition(  50.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_R_40:
  pos.setVecPosition(  40.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_R_30:
  pos.setVecPosition(  30.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_R_20:
  pos.setVecPosition(  20.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_R_10:
  pos.setVecPosition(  10.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_C_B:
  pos.setVecPosition(   0.0, PITCH_WIDTH/2.0 );                  break;
case OBJECT_FLAG_B_0:
  pos.setVecPosition(   0.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_L_10:
  pos.setVecPosition( -10.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_L_20:
  pos.setVecPosition( -20.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_L_30:
  pos.setVecPosition( -30.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_L_40:
  pos.setVecPosition( -40.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_B_L_50:
  pos.setVecPosition( -50.0, PITCH_WIDTH/2.0 + PITCH_MARGIN );   break;
case OBJECT_FLAG_L_B:
  pos.setVecPosition( -PITCH_LENGTH/2.0,  PITCH_WIDTH/2.0 );     break;
case OBJECT_FLAG_L_B_30:
  pos.setVecPosition( -PITCH_LENGTH/2.0 - PITCH_MARGIN,  30.0 ); break;
case OBJECT_FLAG_L_B_20:
  pos.setVecPosition( -PITCH_LENGTH/2.0 - PITCH_MARGIN,  20.0 ); break;
case OBJECT_FLAG_L_B_10:
  pos.setVecPosition( -PITCH_LENGTH/2.0 - PITCH_MARGIN,  10.0 ); break;
case OBJECT_FLAG_G_L_B:
  pos.setVecPosition( -PITCH_LENGTH/2.0,  dGoalWidth/2.0 );      break;
case OBJECT_FLAG_L_0:
  pos.setVecPosition( -PITCH_LENGTH/2.0 - PITCH_MARGIN,  0.0 );  break;
case OBJECT_FLAG_G_L_T:
  pos.setVecPosition( -PITCH_LENGTH/2.0,  -dGoalWidth/2.0 );     break;
case OBJECT_FLAG_L_T_10:
  pos.setVecPosition( -PITCH_LENGTH/2.0 - PITCH_MARGIN, -10.0 ); break;
case OBJECT_FLAG_L_T_20:
  pos.setVecPosition( -PITCH_LENGTH/2.0 - PITCH_MARGIN, -20.0 ); break;
case OBJECT_FLAG_L_T_30:
```

```
      pos.setVecPosition( -PITCH_LENGTH/2.0 - PITCH_MARGIN, -30.0 ); break;
    case OBJECT_FLAG_P_L_T:
      pos.setVecPosition( -PITCH_LENGTH/2.0 + PENALTY_AREA_LENGTH,
                          - PENALTY_AREA_WIDTH/2.0 ); break;
    case OBJECT_FLAG_P_L_C:
      pos.setVecPosition( -PITCH_LENGTH/2.0 + PENALTY_AREA_LENGTH, 0.0 );break;
    case OBJECT_FLAG_P_L_B:
      pos.setVecPosition( -PITCH_LENGTH/2.0 + PENALTY_AREA_LENGTH,
                           PENALTY_AREA_WIDTH/2.0 ); break;
    case OBJECT_FLAG_P_R_T:
      pos.setVecPosition(  PITCH_LENGTH/2.0 - PENALTY_AREA_LENGTH,
                          -PENALTY_AREA_WIDTH/2.0 ); break;
    case OBJECT_FLAG_P_R_C:
      pos.setVecPosition(  PITCH_LENGTH/2.0 - PENALTY_AREA_LENGTH, 0.0 );break;
    case OBJECT_FLAG_P_R_B:
      pos.setVecPosition(  PITCH_LENGTH/2.0 - PENALTY_AREA_LENGTH,
                           PENALTY_AREA_WIDTH/2.0 ); break;
    case OBJECT_FLAG_C:
      pos.setVecPosition(  0.0 , 0.0 );                    break;
    default:
      cerr << "(SoccerTypes::getGlobalPositionObject) wrong objecttype! " <<
        (int)o << "\n" ;
   }

   if( s == SIDE_RIGHT ) // change side for team on the right side.
      pos.setVecPosition( -pos.getX(), -pos.getY() );
   return pos;
}

/*! This method returns the global angle of a lines on the field.  The
    global angle differs for the left and right side. For both teams
    the line behind the opponent goal is seen with global angle
    0. Only for the left team this is the right line and for the right
    team this is the left line.

    \param o one of the four line objects
    \param s side on which the team was started
    \return AngDeg global angle of this line. */
AngDeg SoccerTypes::getGlobalAngleLine( ObjectT o , SideT s )
{
  AngDeg angle;
  switch( o )
  {
    case OBJECT_LINE_L: angle = 180.0; break;
    case OBJECT_LINE_R: angle =   0.0; break;
    case OBJECT_LINE_T: angle = -90.0; break;
```

```
    case OBJECT_LINE_B: angle =  90.0; break;
   default:
    cerr << "(SoccerTypes::getGlobalAngleLine) wrong objecttype! " <<
      (int)o << "\n";
    return UnknownAngleValue;
   }

   if( s == SIDE_RIGHT )
    angle += 180;

   return VecPosition::normalizeAngle( angle );
}

/*! This method returns the string representation of a PlayModeT as is used
    in the Robocup Soccer Simulation and also said by the referee.
    \param pm PlayModeT which should be converted
    \return  pointer to the string (enough memory has to be allocated) */
char* SoccerTypes::getPlayModeStr( PlayModeT pm )
{
   switch( pm )
   {
    case PM_BEFORE_KICK_OFF:          return "before_kick_off";
    case PM_KICK_OFF_LEFT:            return "kick_off_l";
    case PM_KICK_OFF_RIGHT:           return "kick_off_r";
    case PM_KICK_IN_LEFT:             return "kick_in_l";
    case PM_KICK_IN_RIGHT:            return "kick_in_r";
    case PM_CORNER_KICK_LEFT:         return "corner_kick_l";
    case PM_CORNER_KICK_RIGHT:        return "corner_kick_r";
    case PM_GOAL_KICK_LEFT:           return "goal_kick_l";
    case PM_GOAL_KICK_RIGHT:          return "goal_kick_r";
    case PM_GOAL_LEFT:                return "goal_r";
    case PM_GOAL_RIGHT:               return "goal_l";
    case PM_FREE_KICK_FAULT_LEFT:     return "free_kick_fault_l";
    case PM_FREE_KICK_FAULT_RIGHT:    return "free_kick_fault_r";
    case PM_FREE_KICK_LEFT:           return "free_kick_l";
    case PM_FREE_KICK_RIGHT:          return "free_kick_r";
    case PM_INDIRECT_FREE_KICK_RIGHT: return "indirect_free_kick_r";
    case PM_INDIRECT_FREE_KICK_LEFT:  return "indirect_free_kick_l";
    case PM_BACK_PASS_LEFT:           return "back_pass_l";
    case PM_BACK_PASS_RIGHT:          return "back_pass_r";
    case PM_OFFSIDE_LEFT:             return "offside_l";
    case PM_OFFSIDE_RIGHT:            return "offside_l";
    case PM_PENALTY_SETUP_LEFT:       return "penalty_setup_left";
    case PM_PENALTY_SETUP_RIGHT:      return "penalty_setup_right";
    case PM_PENALTY_READY_LEFT:       return "penalty_ready_left";
    case PM_PENALTY_READY_RIGHT:      return "penalty_ready_right";
```

```
    case PM_PENALTY_TAKEN_LEFT:       return "penalty_taken_left";
    case PM_PENALTY_TAKEN_RIGHT:      return "penalty_taken_right";
    case PM_PLAY_ON:                  return "play_on";
    case PM_FROZEN:                   return "play_off";
    case PM_QUIT:                     return "quit";
    case PM_ILLEGAL:
    default:                          return NULL;
   }
}

/*! This method returns the play mode associated with a string.
   \param str representing the play mode
   \return PlayModeT of string, PM_ILLEGAL if it is not recognized */
PlayModeT SoccerTypes::getPlayModeFromStr( char* str )
{
  // all play modes are sent as referee message, so get referee message
  // and look up the associated play mode
  return getPlayModeFromRefereeMessage( getRefereeMessageFromStr( str ) );
}

/*! This method returns the play mode from the referee message.
    \param rm RefereeMessage that contains the play mode
    \return PlayModeT of RefereeMessage, PM_ILLEGAL if it is not recognized */
PlayModeT SoccerTypes::getPlayModeFromRefereeMessage( RefereeMessageT rm )
{
  switch( rm )
  {
    case REFC_BEFORE_KICK_OFF:          return PM_BEFORE_KICK_OFF;
    case REFC_KICK_OFF_LEFT:            return PM_KICK_OFF_LEFT;
    case REFC_KICK_OFF_RIGHT:           return PM_KICK_OFF_RIGHT;
    case REFC_KICK_IN_LEFT:             return PM_KICK_IN_LEFT;
    case REFC_KICK_IN_RIGHT:            return PM_KICK_IN_RIGHT;
    case REFC_CORNER_KICK_LEFT:         return PM_CORNER_KICK_LEFT;
    case REFC_CORNER_KICK_RIGHT:        return PM_CORNER_KICK_RIGHT;
    case REFC_GOAL_KICK_LEFT:           return PM_GOAL_KICK_LEFT;
    case REFC_GOAL_KICK_RIGHT:          return PM_GOAL_KICK_RIGHT;
    case REFC_FREE_KICK_LEFT:           return PM_FREE_KICK_LEFT;
    case REFC_FREE_KICK_RIGHT:          return PM_FREE_KICK_RIGHT;
    case REFC_INDIRECT_FREE_KICK_LEFT:  return PM_INDIRECT_FREE_KICK_LEFT;
    case REFC_INDIRECT_FREE_KICK_RIGHT: return
PM_INDIRECT_FREE_KICK_RIGHT;
    case REFC_FREE_KICK_FAULT_LEFT:     return PM_FREE_KICK_FAULT_LEFT;
    case REFC_FREE_KICK_FAULT_RIGHT:    return PM_FREE_KICK_FAULT_RIGHT;
    case REFC_BACK_PASS_LEFT:           return PM_BACK_PASS_LEFT;
    case REFC_BACK_PASS_RIGHT:          return PM_BACK_PASS_RIGHT;
    case REFC_FOUL_LEFT:                return PM_FREE_KICK_RIGHT;
```

```
    case REFC_FOUL_RIGHT:              return PM_FREE_KICK_LEFT;
    case REFC_OFFSIDE_LEFT:            return PM_OFFSIDE_LEFT;
    case REFC_OFFSIDE_RIGHT:           return PM_OFFSIDE_RIGHT;
    case REFC_GOAL_LEFT:               return PM_GOAL_LEFT;
    case REFC_GOAL_RIGHT:              return PM_GOAL_RIGHT;
    case REFC_PENALTY_SETUP_LEFT:      return PM_PENALTY_SETUP_LEFT;
    case REFC_PENALTY_SETUP_RIGHT:     return PM_PENALTY_SETUP_RIGHT;
    case REFC_PENALTY_READY_LEFT:      return PM_PENALTY_READY_LEFT;
    case REFC_PENALTY_READY_RIGHT:     return PM_PENALTY_READY_RIGHT;
    case REFC_PENALTY_TAKEN_LEFT:      return PM_PENALTY_TAKEN_LEFT;
    case REFC_PENALTY_TAKEN_RIGHT:     return PM_PENALTY_TAKEN_RIGHT;
    case REFC_PLAY_ON:                 return PM_PLAY_ON;
    case REFC_FROZEN:                  return PM_FROZEN;
    case REFC_TIME_OVER:               return PM_TIME_OVER;
    case REFC_QUIT:                    return PM_QUIT;
    default:                           return PM_ILLEGAL;
   }
}

/*! This method returns the string representation of a RefereeMessageT as is
    used in the Robocup Soccer Simulation and said by the referee.
    \param pm RefereeMessageT which should be converted
    \return  pointer to the string (enough memory should be allocated) */
char* SoccerTypes::getRefereeMessageStr( RefereeMessageT rm )
{
  switch( rm )
  {
    case REFC_BEFORE_KICK_OFF:         return "before_kick_off";
    case REFC_KICK_OFF_LEFT:           return "kick_off_l";
    case REFC_KICK_OFF_RIGHT:          return "kick_off_r";
    case REFC_KICK_IN_LEFT:            return "kick_in_l";
    case REFC_KICK_IN_RIGHT:           return "kick_in_r";
    case REFC_CORNER_KICK_LEFT:        return "corner_kick_l";
    case REFC_CORNER_KICK_RIGHT:       return "corner_kick_r";
    case REFC_GOAL_KICK_LEFT:          return "goal_kick_l";
    case REFC_GOAL_KICK_RIGHT:         return "goal_kick_r";
    case REFC_FREE_KICK_LEFT:          return "free_kick_l";
    case REFC_FREE_KICK_RIGHT:         return "free_kick_r";
    case REFC_INDIRECT_FREE_KICK_LEFT:  return "indirect_free_kick_l";
    case REFC_INDIRECT_FREE_KICK_RIGHT: return "indirect_free_kick_r";
    case REFC_FREE_KICK_FAULT_LEFT:    return "free_kick_fault_l";
    case REFC_FREE_KICK_FAULT_RIGHT:   return "free_kick_fault_r";
    case REFC_BACK_PASS_LEFT:          return "back_pass_l";
    case REFC_BACK_PASS_RIGHT:         return "back_pass_r";
    case REFC_PLAY_ON:                 return "play_on";
    case REFC_FROZEN:                  return "play_off";
```

```
    case REFC_QUIT:                  return "quit";
    case REFC_OFFSIDE_LEFT:          return "offside_l";
    case REFC_OFFSIDE_RIGHT:         return "offside_r";
    case REFC_HALF_TIME:             return "half_time";
    case REFC_TIME_UP:               return "time_up";
    case REFC_TIME_OVER:             return "time_over";
    case REFC_TIME_UP_WITHOUT_A_TEAM:  return "time_up_without_a_team";
    case REFC_TIME_EXTENDED:         return "time_extended";
    case REFC_FOUL_LEFT:             return "foul_l";
    case REFC_FOUL_RIGHT:            return "foul_r";
    case REFC_GOAL_LEFT:             return "goal_l";
    case REFC_GOAL_RIGHT:            return "goal_r";
    case REFC_DROP_BALL:             return "drop_ball";
    case REFC_GOALIE_CATCH_BALL_LEFT:  return "goalie_catch_ball_l";
    case REFC_GOALIE_CATCH_BALL_RIGHT: return "goalie_catch_ball_r";
    case REFC_PENALTY_SETUP_LEFT:    return "penalty_setup_left";
    case REFC_PENALTY_SETUP_RIGHT:   return "penalty_setup_right";
    case REFC_PENALTY_READY_LEFT:    return "penalty_ready_left";
    case REFC_PENALTY_READY_RIGHT:   return "penalty_ready_right";
    case REFC_PENALTY_TAKEN_LEFT:    return "penalty_taken_left";
    case REFC_PENALTY_TAKEN_RIGHT:   return "penalty_taken_right";
    case REFC_PENALTY_MISS_LEFT:     return "penalty_miss_left";
    case REFC_PENALTY_MISS_RIGHT:    return "penalty_miss_right";
    case REFC_PENALTY_SCORE_LEFT:    return "penalty_score_left";
    case REFC_PENALTY_SCORE_RIGHT:   return "penalty_score_right";
    case REFC_PENALTY_FOUL_LEFT:     return "penalty_foul_left";
    case REFC_PENALTY_FOUL_RIGHT:    return "penalty_foul_right";
    case REFC_PENALTY_ONFIELD_LEFT:  return "penalty_onfield_left";
    case REFC_PENALTY_ONFIELD_RIGHT: return "penalty_onfield_right";
    case REFC_PENALTY_WINNER_LEFT:   return "penalty_winner_l";
    case REFC_PENALTY_WINNER_RIGHT:  return "penalty_winner_r";
    case REFC_PENALTY_DRAW:          return "penalty_draw";
    case REFC_ILLEGAL:
    default:                         return NULL;
  }
}


/*! This method returns the referee message from the string that is passed.
    \param str pointer to a string with the referee message starting at index 0

    \return RefereeMessageT of string representation, REFC_ILLEGAL if
    it fails */
RefereeMessageT SoccerTypes::getRefereeMessageFromStr( char* str )
{
  switch( str[0] )
```

```
{
  case 'b':                  // before_kick_off, back_pass_[lr]
    if( str[1] == 'e' )
      return REFC_BEFORE_KICK_OFF;
    else
      return (str[10] == 'l') ? REFC_BACK_PASS_LEFT : REFC_BACK_PASS_RIGHT;
  case 'c':                  // corner_kick_l or corner_kick_r
    return (str[12] == 'l') ? REFC_CORNER_KICK_LEFT : REFC_CORNER_KICK_RIGHT;
  case 'd':                  // drop_ball
    return REFC_DROP_BALL;
  case 'g':
    switch( str[5] )
    {
      case 'k':              // goal_kick_l, goal_kick_r
        return (str[10] == 'l') ? REFC_GOAL_KICK_LEFT : REFC_GOAL_KICK_RIGHT;
      case 'e':              // goalie_catch_ball_l, goalie_catch_ball_r
        return (str[18] == 'l') ? REFC_GOALIE_CATCH_BALL_LEFT
                                : REFC_GOALIE_CATCH_BALL_RIGHT;
      case 'l':              // goal_l
        return REFC_GOAL_LEFT;
      case 'r':              // goal_r
        return REFC_GOAL_RIGHT;
    }
  case 'h':                  // half_time
    return REFC_HALF_TIME;
  case 'i':                  // indirect_free_kick_[lr]
    if( strlen( str ) > 19 )
      return ( str[19] == 'l' )
        ?  REFC_INDIRECT_FREE_KICK_LEFT
        :  REFC_INDIRECT_FREE_KICK_RIGHT;
    break;
  case 'f':
    if( str[5] == 'k' )      // free_kick_[lr], free_kick_fault_[lr]
    {
      if( str[10] == 'f' )
        return ( str[16] == 'l' )
                ? REFC_FREE_KICK_FAULT_LEFT
                : REFC_FREE_KICK_FAULT_RIGHT;
      else
        return (str[10] == 'l') ? REFC_FREE_KICK_LEFT : REFC_FREE_KICK_RIGHT;
    }
    else if( str[5] == 'l' ) // foul_l
      return REFC_FOUL_LEFT;
    else if( str[5] == 'r' ) // foul_r
      return REFC_FOUL_RIGHT;
  case 'k':                  // kick_in_l, kick_in_r, kick_off_l, kick_off_r
```

```
  if( str[5] == 'i' )
    return (str[8] == 'l' )   ? REFC_KICK_IN_LEFT   : REFC_KICK_IN_RIGHT;
  else if( str[5] == 'o' )
    return (str[9] == 'l' )   ? REFC_KICK_OFF_LEFT  : REFC_KICK_OFF_RIGHT;
case 'o':
  return ( str[8] == 'l' )    ? REFC_OFFSIDE_LEFT   : REFC_OFFSIDE_RIGHT;
case 'p':               // play_on or penalty_setup_l or penalty_ready_l
  if( str[1] == 'l' )    // penalty_taken_l, penalty_miss_l, penalty_foul_l
    return REFC_PLAY_ON; // penalty_onfield_l, penalty_score_l,
  else if( str[8] == 's' && str[9] == 'e' )
                    // penalty_winner_l,penalty_draw
    return ( str[14] == 'l' ) ? REFC_PENALTY_SETUP_LEFT
                              : REFC_PENALTY_SETUP_RIGHT;
  else if( str[8] == 's' && str[9] == 'c' )
    return ( str[14] == 'l' ) ? REFC_PENALTY_SCORE_LEFT
                              : REFC_PENALTY_SCORE_RIGHT;
  else if( str[8] == 'r' )
    return ( str[14] == 'l' ) ? REFC_PENALTY_READY_LEFT
                              : REFC_PENALTY_READY_RIGHT;
  else if( str[8] == 't' )
    return ( str[14] == 'l' ) ? REFC_PENALTY_TAKEN_LEFT
                              : REFC_PENALTY_TAKEN_RIGHT;
  else if( str[8] == 'm' )
    return ( str[13] == 'l' ) ? REFC_PENALTY_MISS_LEFT
                              : REFC_PENALTY_MISS_RIGHT;
  else if( str[8] == 'o' )
    return ( str[16] == 'l' ) ? REFC_PENALTY_ONFIELD_LEFT
                              : REFC_PENALTY_ONFIELD_RIGHT;
  else if( str[8] == 'f' )
    return ( str[13] == 'l' ) ? REFC_PENALTY_FOUL_LEFT
                              : REFC_PENALTY_FOUL_RIGHT;
  else if( str[8] == 'f' )
    return ( str[15] == 'l' ) ? REFC_PENALTY_WINNER_LEFT
                              : REFC_PENALTY_WINNER_RIGHT;
  else if( str[8] == 'd' )
    return REFC_PENALTY_DRAW;
case 't':
  if( str[5] == 'o' )      // time_over
    return REFC_TIME_OVER;
  else if( str[5] == 'e' ) // time_extended
    return REFC_TIME_EXTENDED;
  else if( str[5] == 'u' ) // time_up
    return REFC_TIME_UP;
  else if( str[8] == 'w' ) // time_up_without_a_team
    return REFC_TIME_UP_WITHOUT_A_TEAM;
default:
```

```
      printf("(SoccerTypes::getRefereeMessageFromStr) ?? ref msg %s\n",str);
      return REFC_ILLEGAL;
    }
   printf("(SoccerTypes::getRefereeMessageFromStr) ?? ref msg %s\n",str);
   return REFC_ILLEGAL;
}


/*! This method returns the string representation of a ViewAngleT as is used
    in the Robocup Soccer Simulation
    \param va ViewAngleT which should be converted
    \return  pointer to the string (enough memory should be allocatd) */
char* SoccerTypes::getViewAngleStr( ViewAngleT va )
{
  switch( va )
  {
    case VA_NARROW:  return "narrow";
    case VA_NORMAL:  return "normal";
    case VA_WIDE:    return "wide";
    case VA_ILLEGAL:
    default:         return NULL;
  }
}

/*! This method returns et the view angle from the specified string
    \param str pointer to a string that contains view angle string at index 0
    \return ViewAngleT of string, VA_ILLEGAL if it is not recognized */
ViewAngleT SoccerTypes::getViewAngleFromStr( char* str )
{
  switch( str[1] )
  {
    case 'a': return VA_NARROW;
    case 'o': return VA_NORMAL;
    case 'i': return VA_WIDE;
    default:  return VA_ILLEGAL;
  }
}

/*! This method returns the half angle value that belongs to the
    ViewAngle that is given as the first argument (VA_NARROW,
    VA_NORMAL or VA_WIDE). The half view angle is returned since this
    makes it easier to check whether an object lies in the view cone
    (the global relative angle must be smaller than the half view
    angle.

    \param va view angle
```

```
   \return angle denoting the half of the corresponding view angle */
AngDeg SoccerTypes::getHalfViewAngleValue( ViewAngleT va )
{
 switch( va )
 {
  case VA_NARROW:  return 22.5;
  case VA_NORMAL:  return 45.0;
  case VA_WIDE:    return 90.0;
  case VA_ILLEGAL:
  default:         return 0.0;
 }

}

/*! This method returns the string representation of a ViewQualityT as is used
   in the Robocup Soccer Simulation
   \param vq ViewQualityT which should be converted
   \return  pointer to the string (enough memory should be allocated) */
char* SoccerTypes::getViewQualityStr( ViewQualityT vq )
{
 switch( vq )
 {
  case VQ_HIGH:    return "high";
  case VQ_LOW:     return "low";
  case VQ_ILLEGAL:
  default:         return NULL;
 }
}

/*! This method returns the view quality from the string that is passed as the
   first argument
  \param str pointer to a string that contains view quality string at index 0
  \return ViewQualityT of string, VQ_ILLEGAL if it is not known */
ViewQualityT SoccerTypes::getViewQualityFromStr( char* str )
{
   if( str[0] == 'h' )      return VQ_HIGH;
   else if( str[0] == 'l' ) return VQ_LOW;
   else                     return VQ_ILLEGAL;
}

/*! This method returns the string representation of a CommandT as is
   used in the Robocup Soccer Simulation
   \param com CommandT that should be converted
   \return  pointer to the string (enough memory should be allocated) */
char* SoccerTypes::getCommandStr( CommandT com )
{
```

```
  switch( com )
  {
    case CMD_DASH:        return "dash";
    case CMD_TURN:        return "turn";
    case CMD_TURNNECK:    return "turn_neck";
    case CMD_CHANGEVIEW:  return "change_view";
    case CMD_CATCH:       return "catch";
    case CMD_KICK:        return "kick";
    case CMD_MOVE:        return "move";
    case CMD_SENSEBODY:   return "sense_body";
    case CMD_SAY:         return "neck";
    case CMD_ATTENTIONTO: return "attentionto";
    case CMD_TACKLE:      return "tackle";
    case CMD_POINTTO:     return "pointto";
    case CMD_ILLEGAL:
    default:              return NULL;

  }
}

/*! This method returns return true when argument is a primary action
    (action that can only be sent once a cycle). This is the case for kick,
    dash, move, turn and catch commands.
    \param CommandT command that should be checked
    \return true when it is a primary action, false otherwise */
bool SoccerTypes::isPrimaryCommand( CommandT com )
{
  return com == CMD_KICK || com == CMD_DASH || com == CMD_MOVE ||
         com == CMD_TURN || com == CMD_CATCH || com == CMD_TACKLE ;
}


/*! This method returns the string representation of a SideT as is used in the
    Robocup Soccer Simulation (r or l).
    \param s SideT which should be converted
    \return pointer to the string */
char* SoccerTypes::getSideStr( SideT s )
{
  switch( s )
  {
    case SIDE_LEFT:    return "l";
    case SIDE_RIGHT:   return "r";
    case SIDE_ILLEGAL:
    default:           return NULL;
  }
}
```

```
/*! This method returns the SideT from the string that is passed as the first
    argument.
    \param str pointer to a string that contains side info string at index 0
    \return SideT of string representation, SIDE_ILLEGAL if it is not known */
SideT SoccerTypes::getSideFromStr( char* str )
{
  if( str[0] == 'l' )      return SIDE_LEFT;
  else if( str[0] == 'r' ) return SIDE_RIGHT;
  else                     return SIDE_ILLEGAL;

}

/*! This method returns the string representation of the BallStatus as is used
    in the Robocup Soccer Simulation (in_field, goal_left, goal_right or
    out_of_field).
    \param bs BallStatus which should be converted
    \return pointer to the string (enough memory should be allocated) */
char* SoccerTypes::getBallStatusStr( BallStatusT bs )
{
  switch( bs )
  {
    case BS_IN_FIELD:      return "in_field";
    case BS_GOAL_LEFT:     return "goal_left";
    case BS_GOAL_RIGHT:    return "goal_right";
    case BS_OUT_OF_FIELD:  return "out_of_field";
    default:               return NULL;
  }
}

/*! This method returns the BallStatus from the string that is passed as the
    first argument.
   \param str pointer to a string that contains ball status info at index 0

   \return BallStatus of string representation, BS_ILLEGAL if it is
   not known */
BallStatusT SoccerTypes::getBallStatusFromStr( char* str )
{
  switch( str[0] )
  {
    case 'i': return BS_IN_FIELD;
    case 'o': return BS_OUT_OF_FIELD;
    case 'g': return (str[5]=='l') ? BS_GOAL_LEFT : BS_GOAL_RIGHT;
    default:
            cout << "(SoccerTypes::getBallStatusFromStr) illegal status " <<
                 str << "\n";
```

```
            return BS_ILLEGAL;
  }
}

/*! This method returns the angle value corresponding to the direction
    'dir'. */
AngDeg SoccerTypes::getAngleFromDirection( DirectionT dir )
{
  switch( dir )
  {
    case DIR_NORTH:      return    0.0;
    case DIR_NORTHWEST:  return  -45.0;
    case DIR_WEST:       return  -90.0;
    case DIR_SOUTHWEST:  return -135.0;
    case DIR_SOUTH:      return  180.0;
    case DIR_SOUTHEAST:  return  135.0;
    case DIR_EAST:       return   90.0;
    case DIR_NORTHEAST:  return   45.0;
    case DIR_ILLEGAL:
    default:
      break;

  }
  cerr << "(SoccerTypes::getAngleFromDirection) illegal dir " << dir << endl;
  return 0.0;
}




/*************************************************************************
/
/********************* TESTING PURPOSES
*************************************/
/*************************************************************************
/

/*
int main( void )
{
  char* s = "(p \"1\" 3)";

      "(see 0 ((g r) 56.3 0) ((f c) 3.8 0 -0 0) ((f r t) 66 -31) ((f r b)
      66 31) ((f p r t) 44.7 -26) ((f p r c) 39.6 0) ((f p r
      b) 44.7 26) ((f g r t) 56.8 -7) ((f g r b) 56.8 7) ((f t
```

```
    r 40) 58.6 -41) ((f t r 50) 66.7 -35) ((f b r 40) 58.6
    41) ((f b r 50) 66.7 35) ((f r t 30) 68 -26) ((f r t 20)
    64.7 -18) ((f r t 10) 62.2 -9) ((f r 0) 61.6 0) ((f r b
    10) 62.2 9) ((f r b 20) 64.7 18) ((f r b 30) 68 26) ((b)
    3.7 0 0 0) ((P) 1.2 179) ((P) 0.6 179) ((p "1" 3) nan 0
    0 0 0 0) ((p "1" 4) nan 0 0 0 0 0) ((l r) 56.3 -89))";
  char *ptr = s;
  char buf[1023];
  printf("%s\n", ptr );
  ObjectT o = SoccerTypes::getObjectFromStr(&ptr, "1");
  printf("%d, %s\n", o, SoccerTypes::getObjectStr(buf,o,"1") );
}


int main( void )
{
  Time t( 10, 0 );
  t.show();
  t.updateTime( 10 );
  t.show();
  t.updateTime( 10 );
  t.show();
  cout << t.isStopped();
  t.updateTime( 12 );
  t.show();
  t.updateTime( 12);
  t.setTimeStopped(2);
  Time t2( 12, 3 );
  cout << t.getTimeDifference( t2 ) << " " << t2.isStopped() << "\n";
  cout << (t < t2 ) << ( t > t2 ) << ( t <= t2 ) << (t == t2 ) << "\n";
}

*/
```

Formation.h

```
#ifndef _FORMATIONS_
#define _FORMATIONS_

#include "SoccerTypes.h"  // PlayerT

/*****************************************************************************
/
/********************** CLASS PLAYERTYPEINFO
**********************************/
/*****************************************************************************
/

/*! This class contains information for one individual player_type, defined in
SoccerTypes.h. A player_type should not be confused with the player_types
introduced in soccerserver 7.xx. A playerType PlayerT is defined as the kind of
a player. Different possibilities are PT_ATTACKER, PT_MIDFIELDER_WING, etc.
This class contains different characteristics of one playertype. This
information consists of the following values:
- dAttrX - x attraction to the ball for this player type.
- dAtttY - y attraction to the ball for this player type.
- dMinX  - minimal x coordinate for this player
- dMaxX  - maximal x coordinate for this player
- bBehindBall - indicating whether this player type should always stay behind
the ball or not.


This class contains different get and set methods to change the values
associated for this class, normally these are changed when the
Formations class reads in the formation file. */
class PlayerTypeInfo
{
  PlayerT playerType; /*!< This class gives information about this PlayerType*/
  double  dAttrX;     /*!< x attraction to the ball                    */
  double  dAttrY;     /*!< y attraction to the ball                    */
  double  dMinX;      /*!< minimal x coordinate for this player type       */
  double  dMaxX;      /*!< maximal x coordinate for this player type       */
  bool    bBehindBall;/*!< should player always stay behind the ball       */

public:
  PlayerTypeInfo( );
  PlayerTypeInfo( PlayerT, double, double, double, double, bool );

  // method to set all the values at once and displaying them all
  bool    setValues( PlayerT, double, double, double, double, bool  );
  void    show     ( ostream &os = cout                          );
```

```
  // standard get and set methods to individually set all the values
  bool    setPlayerType    ( PlayerT type  );
  PlayerT getPlayerType    (               ) const;
  bool    setAttrX         ( double  attrX );
  double  getAttrX         (               ) const;
  bool    setAttrY         ( double  attrY );
  double  getAttrY         (               ) const;
  bool    setMinX          ( double  minX  );
  double  getMinX          (               ) const;
  bool    setMaxX          ( double  maxX  );
  double  getMaxX          (               ) const;
  bool    setBehindBall    ( bool    b     );
  bool    getBehindBall    (               ) const;

};


/*************************************************************************
/
/********************** CLASS FORMATIONTYPEINFO
*******************************/
/*************************************************************************
/

/*!This class contains information about one specific formation. It
   contains the formation type (defined in SoccerTypes.h), the home
   position of all the roles (=specific player in a formation), the
   player types for all the roles and the information about the
   different player_types. Furthermore it contains methods to retrieve
   this information for a specific role. */
class FormationTypeInfo
{
  FormationT      formationType;              /*!< type of this formation   */
  VecPosition     posHome[ MAX_TEAMMATES ];    /*!< home position for roles  */
  PlayerT         playerType[ MAX_TEAMMATES ]; /*!< player_types for roles   */
  PlayerTypeInfo  playerTypeInfo[ MAX_PLAYER_TYPES ]; /*!< info for roles    */

public:
  FormationTypeInfo(                  );
  void show(         ostream &os = cout );

  // get and set methods to get information for a player in this formation
  bool           setFormationType        ( FormationT  type               );
  FormationT     getFormationType        (                         ) const;
  bool           setPosHome              ( VecPosition pos,     int atIndex);
  bool           setXPosHome             ( double      x,       int atIndex);
```

```
  bool           setYPosHome             ( double      y,       int atIndex);
  VecPosition    getPosHome              ( int         atIndex       ) const;
  bool           setPlayerType           ( PlayerT     type,    int atIndex);
  PlayerT        getPlayerType           ( int         atIndex       ) const;
  bool           setPlayerTypeInfo       ( PlayerTypeInfo info, int atIndex);
  PlayerTypeInfo* getPlayerTypeInfo      ( int         atIndex            );
  PlayerTypeInfo* getPlayerTypeInfoOfPlayer( int         iPlayerInFormation  );
};

/*****************************************************************************
/
/********************** CLASS FORMATIONS
************************************/
/*****************************************************************************
/

/*!This class is a container for all different Formation Types: it
   contains the information of all the formation types. Furthermore it
   contains two other values: the current formation type that is used
   by the agent and the role of the agent in the current
   formation. These two values fully specify the position of this
   player in the formation. */
class Formations
{
  FormationTypeInfo formations[ MAX_FORMATION_TYPES ]; /*!< stored formations*/
  FormationT       curFormation;      /*!< type of the current formation   */
  int              iPlayerInFormation; /*!< role agent in current formation */
public:
  Formations( const char *strFile = NULL, FormationT ft=FT_ILLEGAL, int iNr=1);
  void show ( ostream &os = cout );

  // method to get the strategic position depending on different factors
  VecPosition getStrategicPosition( int         iPlayer,
                                    VecPosition posBall,
                                    double      dMaxXInPlayMode,
                                    bool        bInBallPossession = false,
                                    double      dMaxYPercentage = 0.75,
                                                FormationT  ft = FT_ILLEGAL );

  // method to read the formations from a formation configuration file.
  bool       readFormations      ( const char *strFile         );

  // standard get and set methods for the different member variables
  bool       setFormation        ( FormationT formation           );
  FormationT  getFormation        (                              ) const;
  bool       setPlayerInFormation( int number                    );
```

```
  int        getPlayerInFormation( ObjectT    obj = OBJECT_ILLEGAL ) const;
  PlayerT    getPlayerType       ( ObjectT    obj,
                                                FormationT ft = FT_ILLEGAL      ) const;
  PlayerT    getPlayerType       ( int        iIndex = -1,
                                                FormationT ft = FT_ILLEGAL      ) const;
};

#endif
```

Formation.cpp

```
#include "Formations.h"
#include "Parse.h"       // Parse

#include <fstream>       // ifstream
#include <stdio.h>       // printf
#include <math.h>        // fabs

/*************************************************************************
/
/********************** CLASS PLAYERTYPEINFO
**********************************/
/*************************************************************************
/

/*! This method is the default constructor and sets all the values of this
    class to "illegal" values. This method is needed when an array of this
    class is initialized, since then the default constructor (without
    arguments) is called. Afterwards the actual values should be set using
    the method setValues. */
PlayerTypeInfo::PlayerTypeInfo()
{
  setValues( PT_ILLEGAL, UnknownDoubleValue,UnknownDoubleValue,
           UnknownDoubleValue,UnknownDoubleValue, false );
}

/*! This Constructor receives the values for all the member variables as
    arguments and initializes the member variables using the method setValues.
    \param pt PlayerType corresponding to the player type of this class
    \param dAttrX x attraction to the ball
    \param dAttrY y attraction to the ball
    \param dMinX minimal x coordinate for this player type
    \param dMaxX maximal x coordinate for this player type
    \param bBehindBall boolean indicating whether this player type should
                    always stay behind the ball. */
PlayerTypeInfo::PlayerTypeInfo( PlayerT pt, double dAttrX, double dAttrY,
           double dMinX, double dMaxX, bool bBehindBall  )
{
  setValues( pt, dAttrX, dAttrY, dMinX, dMaxX, bBehindBall );
}

/*! This method receives the values for all the member variables as arguments
    and sets these member variables.
    \param pt PlayerType corresponding to the player type of this class
    \param ax x attraction to the ball
    \param ay y attraction to the ball
    \param minx minimal x coordinate for this player type
```

```
    \param maxx maximal x coordinate for this player type
    \param bb boolean indicating whether this player type should always
                  stay behind the ball.
    \return bool indicating whether update was successful. */
bool PlayerTypeInfo::setValues( PlayerT pt,   double ax,   double ay,
                     double minx,  double maxx, bool bb )
{
  playerType      = pt;
  dAttrX          = ax;
  dAttrY          = ay;
  dMinX           = minx;
  dMaxX           = maxx;
  bBehindBall     = bb;

  return true;
}

/*! This method print the different member values separated by comma's to the
    specified output stream.
    \param os output stream to which member values are printed */
void PlayerTypeInfo::show( ostream &os )
{
  os << "(" << (int)playerType << ", " << dAttrX << ", " << dAttrY << ", "
           << dMinX           << ", " << dMaxX  << ", " << bBehindBall
           <<  ")" << endl;

}

/*! This method sets the player type associated with this class.
    \param type new player type
    \return bool indicating whether update was succesfull */
bool PlayerTypeInfo::setPlayerType( PlayerT type )
{
  playerType = type;
  return true;
}

/*! This method returns the player type associated with this class.
    \return player type of this class */
PlayerT PlayerTypeInfo::getPlayerType( ) const
{
  return playerType;
}

/*! This method sets the x attraction to the ball for this player type. The x
    attraction to the ball is a double in the range (0,1). This value is used
```

```
    to determine the x coordinate of the strategic position for this player
    type. The x attraction of the ball is multiplied with the x coordinate of
    the ball and added to the home position of the agent to determine the x
    coordinate of the strategic position.
    \param dAttractionX new x attraction for this player type
    \return bool indicating whether update was succesfull */
bool PlayerTypeInfo::setAttrX( double dAttractionX )
{
  dAttrX = dAttractionX;
  return true;
}

/*! This method returns the x attraction to the ball for this player type. The
    x attraction to the ball is a double in the range (0,1). This value is used
    to determine the x coordinate of the strategic position for this player
    type. The x attraction of the ball is multiplied with the x coordinate of
    the ball and added to the home position of the agent to determine the x
    coordinate of the strategic position.
    \return x attraction for this player type */
double PlayerTypeInfo::getAttrX( ) const
{
  return dAttrX;
}

/*! This method sets the y attraction to the ball for this player type. The y
    attraction to the ball is a double in the range (0,1). This value is used
    to determine the y coordinate of the strategic position for this player
    type. The y attraction of the ball is multiplied with the y coordinate of
    the ball and added to the home position of the agent to determine the y
    coordinate of the strategic position.
    \param dAttractionY new y attraction for this player type
    \return bool indicating whether update was succesfull */
bool PlayerTypeInfo::setAttrY( double dAttractionY )
{
  dAttrY = dAttractionY;
  return true;
}

/*! This method returns the y attraction to the ball for this player type. The
    y attraction to the ball is a double in the range (0,1). This value is used
    to determine the y coordinate of the strategic position for this player
    type. The y attraction of the ball is multiplied with the y coordinate of
    the ball and added to the home position of the agent to determine the y
    coordinate of the strategic position.
    \return y attraction for this player type */
double PlayerTypeInfo::getAttrY( ) const
```

```
{
 return dAttrY;
}

/*! This method sets the minimal x coordinate for this player type. When the
    calculated x coordinate for the strategic position is lower than this
    value, the x coordinate is set to this minimal x coordinate.
    \param dMinimalX new minimal x coordinate for this player type
    \return bool indicating whether update was succesfull. */
bool PlayerTypeInfo::setMinX( double dMinimalX )
{
 dMinX = dMinimalX;
 return true;
}

/*! This method returns the minimal x coordinate for this player type. When the
    calculated x coordinate for the strategic position is lower than this
    value, the x coordinate is set to this minimal x coordinate.
    \return minimal x coordinate for this player type */
double PlayerTypeInfo::getMinX( ) const
{
 return dMinX;
}

/*! This method sets the maximal x coordinate for this player type. When the
    calculated x coordinate for the strategic position is larger than this
    value, the x coordinate is set to this maximal x coordinate.
    \param dMaximalX new maximal x coordinate for this player type
    \return bool indicating whether update was succesfull. */
bool PlayerTypeInfo::setMaxX( double dMaximalX )
{
 dMaxX = dMaximalX;
 return true;
}

/*! This method returns the maximal x coordinate for this player type. When the
    calculated x coordinate for the strategic position is larger than this
    value, the x coordinate is set to this maximal x coordinate.
    \return maximal x coordinate for this player type */
double PlayerTypeInfo::getMaxX( ) const
{
 return dMaxX;
}

/*!This method sets the value that indicates whether this player type should
   stay behind the ball or not. When set to true and the strategic position for
```

```
   this player type is calculated to be in front of the ball. The x coordinate
   of the strategic position is set to the x coordinate of the ball.
   \param b boolean indicating whether this playertype should stay behind the
          ball
   \return bool indicating whether update was succesfull. */
bool PlayerTypeInfo::setBehindBall( bool b )
{
  bBehindBall = b;
  return true;
}

/*! This method returns the value that indicates whether this player type
    should stay behind the ball or not. When set to true and the strategic
    position for this player type is calculated to be in front of the ball.
    The x coordinate of the strategic position is set to the x coordinate
    of the ball.
    \return bool indicating whether to stay behind the ball or not */
bool PlayerTypeInfo::getBehindBall( ) const
{
  return bBehindBall;
}

/*************************************************************************
/
/********************** CLASS
FORMATIONTYPEINFO*******************************/
/*************************************************************************
/

/*! This is the default constructor and does nothing. */
FormationTypeInfo::FormationTypeInfo( )
{
}

/*!This method sets the current formation type for this class.
   \param type new formation type for this class.
   \return bool indicating whether update was successful. */
bool FormationTypeInfo::setFormationType( FormationT type )
{
  formationType = type;
  return true;
}

/*! This method return the current formation type for this class.
    \return formation type for this class. */
FormationT FormationTypeInfo::getFormationType( ) const
```

```
{
  return formationType;
}

/*! This method prints all the information about this formation to the
    specified output stream. The format is the following:
    - x coordinate of the home position for all the roles
    - y coordinate of the home position for all the roles
    - the player types for all the roles
    - the x attraction for all the player types
    - the y attraction for all the player types
    - indication whether to stay behind the ball for all the player types
    - minimal x coordinate for all the player types
    - maximal x coordinate for all the player types
    \param os output stream for output. */
void FormationTypeInfo::show( ostream &os )
{
  char str[128];
  for( int i = 0; i < MAX_TEAMMATES; i++ )
  {
    sprintf(str, "%3.2f ", posHome[i].getX() );
    os << str;
  }
  os << endl;
  for( int i = 0; i < MAX_TEAMMATES; i++ )
  {
    sprintf( str, "%3.2f ", posHome[i].getY() );
    os << str;
  }
  os << endl;
  for( int i = 0; i < MAX_TEAMMATES; i++ )
  {
    sprintf( str, "%5d ", (int)playerType[i] );
    os << str;
  }
  os << endl;
  for( int i = 0; i < MAX_PLAYER_TYPES; i++ )
  {
    sprintf( str, "%3.2f ", playerTypeInfo[i].getAttrX() );
    os << str;
  }
  os << endl;
  for( int i = 0; i < MAX_PLAYER_TYPES; i++ )
  {
    sprintf( str, "%3.2f ", playerTypeInfo[i].getAttrY() );
    os << str;
```

```
  }
  os << endl;
  for( int i = 0; i < MAX_PLAYER_TYPES; i++ )
  {
    sprintf( str, "%5d ", playerTypeInfo[i].getBehindBall() );
    os << str;
  }
  os << endl;
  for( int i = 0; i < MAX_PLAYER_TYPES; i++ )
  {
    sprintf( str, "%3.2f ", playerTypeInfo[i].getMinX() );
    os << str;
  }
  os << endl;
  for( int i = 0; i < MAX_PLAYER_TYPES; i++ )
  {
    sprintf( str, "%3.2f ", playerTypeInfo[i].getMaxX() );
    os << str;
  }
  os << endl;
}


/*! This method sets the home position of the role indicated by the number
    'atIndex' in this formation. The home position is the position from which
    the strategic position is calculated and could be interpreted as the
    position a player is located when the ball is at the position (0,0).
    \param pos new home position for the player with role number 'atIndex'
    \param atIndex index of the player with role for which the home
           position should be set.
    \return bool indicating whether update was succesfull. */
bool FormationTypeInfo::setPosHome( VecPosition pos, int atIndex )
{
  posHome[ atIndex ] = pos;
  return true;
}

/*! This method sets the x coordinate of the home position for the player
    with role number 'atIndex'.
    \param x x coordinate for the home position
    \param atIndex role in formation for which x coordinate should be set.
    \return bool indicating whether update was succesfull. */
bool FormationTypeInfo::setXPosHome( double x, int atIndex )
{
  posHome[ atIndex ].setX( x );
  return true;
```

```
}

/*! This method sets the y coordinate of the home position for the player
    with role number 'atIndex'.
    \param y y coordinate for the home position
    \param atIndex role number for which y coordinate should be set.
    \return bool indicating whether update was succesfull. */
bool FormationTypeInfo::setYPosHome( double y, int atIndex )
{
  posHome[ atIndex ].setY( y );
  return true;
}

/*! This method returns the home position for the player with role number
    atIndex in this formation. The home position is the position from which the
    strategic position is calculated and could be interpreted as the position a
    player is located when the ball is at the position (0,0).
    \return home position for player number at index 'atIndex' */
VecPosition FormationTypeInfo::getPosHome( int atIndex ) const
{
  return posHome[ atIndex ];
}

/*! This method sets the player type for the player with role number 'atIndex'
    in this formation.
    \param type new player type for role at position 'atIndex'
    \param atIndex role number for which player type should be set.
    \return bool indicating whether update was succesfull. */
bool FormationTypeInfo::setPlayerType( PlayerT type, int atIndex )
{
  playerType[ atIndex ] = type;
  return true;
}

/*! This method returns the player type for the player with role number
    'atIndex' in this formation.
    \return player type for player with role number 'atIndex' */
PlayerT FormationTypeInfo::getPlayerType( int atIndex ) const
{
  return playerType[ atIndex ];
}

/*! This method sets the information for a player type in this formation. Note
    that information is for a player TYPE and not for a player ROLE.
    \param info new player type information for the player type at 'atIndex'.
    \param atIndex number of player type for which information should be set.
```

```
   \return bool indicating whether update was succesfull. */
bool FormationTypeInfo::setPlayerTypeInfo( PlayerTypeInfo info, int atIndex )
{
 playerTypeInfo[ atIndex ] = info;
 return true;
}

/*! This method returns (a pointer to) the player type information for the
   player type at position 'atIndex'
   \param atIndex index of which player type information should be returned
   \return pointer to player type information located at index 'atIndex' */
PlayerTypeInfo* FormationTypeInfo::getPlayerTypeInfo( int atIndex )
{
 return &playerTypeInfo[ atIndex ];
}


/*! This method returns (a pointer to) the player type information for the
   player with role number 'iPlayerInFormation'.
   \param iPlayerInFormation role number for which info should be returned
   \return pointer to information for role 'iPlayerInFormation'*/
PlayerTypeInfo* FormationTypeInfo::getPlayerTypeInfoOfPlayer(
                                        int iPlayerInFormation )
{
 return &playerTypeInfo[ playerType[iPlayerInFormation] ];
}

/*****************************************************************************
/
/********************** CLASS FORMATIONS
*************************************/
/*****************************************************************************
/

/*! This is the constructor for the Formations class and needs as arguments
   a formation configuration file, the current formation and the number of the
   agent in this formation (normally at start-up this equals the player
   number).
   \param strFile string representation of the formation configuration file
   \param curFt current formation type (default FT_ILLEGAL)
   \param iNr number of the agent in this formation (default 1)*/
Formations::Formations( const char *strFile, FormationT curFt, int iNr )
{
 if( strFile[0] == '\0' )
 {
   cerr << "(Formations::Formations) No Filename given" << endl;
```

```
    return;
  }

  if( readFormations( strFile ) == false )
    cerr << "(Formations::Formations) Error reading file " << strFile << endl;
  curFormation = curFt;
  setPlayerInFormation( iNr );
}

/*! This methods prints all the information of the different formation types to
    the output stream os and furthermore prints the current formation and the
    role number of the agent in this formation.
    \param os output stream to which output is written. */
void Formations::show( ostream &os )
{
  for( int i = 0 ; i < MAX_FORMATION_TYPES; i ++ )
    formations[i].show( os );
  os << "Current formation: " << (int)curFormation << endl
       << "Player nr in formation: " << iPlayerInFormation ;
}

/*! This method returns the strategic position for a player. It calculates this
    information by taking the home position of the current role in the
    current formation and combines this with the position of the ball using the
    attraction values for the current player type. The attraction values
    defines the percentage of the ball coordinate that is added to the home
    position of the current player type. So when the x coordindate of the home
    position is 10.0, x coordinate ball is 20.0 and x attraction is 0.25. The
    x coordinate of the strategic position will become 10.0 + 0.25*20.0 = 15.0.
    When this value is smaller than the minimal x coordinate or larger than
    the maximal x coordinate, the coordinate is changed to this minimal or
    maximal coordinate respectively. Also when the behind ball value is set,
    the x coordinate of the strategic position is set to this ball coordinate.
    Furthermore when the strategic position is in front of the supplied
    argument dMaxXInPlayMode, the x coordinate is adjusted to this value.
    During normal play mode the supplied value is often the offside line.
    \param iPlayer player number in formation of which strategic position
                should be determined.
    \param posBall position of the ball
    \param dMaxXInPlayMode, max x coordinate allowed in current play mode. */
VecPosition Formations::getStrategicPosition( int iPlayer, VecPosition posBall,
     double dMaxXInPlayMode, bool bInBallPossession, double dMaxYPercentage,
     FormationT ft )
{
  if( ft == FT_ILLEGAL )
    ft = curFormation;
```

```
  VecPosition     posHome;
  PlayerTypeInfo* ptInfo = formations[ft].
                     getPlayerTypeInfoOfPlayer( iPlayer );
  double x, y;

  // get the home position and calculate the associated strategic position
  posHome = formations[ft].getPosHome( iPlayer );
  y = posHome.getY() + posBall.getY() * ptInfo->getAttrY();
  x = posHome.getX() + posBall.getX() * ptInfo->getAttrX();

  // do not move too much to the side
  if( fabs( y ) > 0.5*dMaxYPercentage*PITCH_WIDTH )
    y = sign(y)*0.5*dMaxYPercentage*PITCH_WIDTH;

  // when behind ball is set, do not move to point in front of ball
  if( ptInfo->getBehindBall() == true && x > posBall.getX() )
    x = posBall.getX();

  // do not move past maximal x or before minimal x
  if( x > ptInfo->getMaxX() )
    x = ptInfo->getMaxX();
  else if( x < ptInfo->getMinX() )
    x = ptInfo->getMinX();

  // when x coordinate is in front of allowed x value, change it
  if( x > dMaxXInPlayMode )
    x = dMaxXInPlayMode;

  return VecPosition( x, y );
}

/*! This method reads the formations from the file 'strFile' and has the
    following format:
       - x coordinate of the home position for all the roles
       - y coordinate of the home position for all the roles
       - the player types for all the roles
       - the x attraction for all the player types
       - the y attraction for all the player types
       - indication whether to stay behind the ball for all the player types
       - minimal x coordinate for all the player types
       - maximal x coordinate for all the player types
       - extra y distance added to strat. pos. when in ball possession
    \param strFile string representation of the file.
    \return bool when file was read in succesfully. */
bool Formations::readFormations( const char *strFile )
{
```

```
ifstream in( strFile );
if( !in )
{
  cerr << "(readValues::readValues) Could not open file '" <<
  strFile << "'" << endl;
  return false;
}

char strLine[256], *str;
int           iLineNr          = 0, i;
int           iForm            = 0; // current formation type that is parsed
int           iLineInFormation = 0; // current offset of line in formation
bool          bReturn          = true;
PlayerTypeInfo *pt_info;

// read all lines
while( bReturn && in.getline( strLine, sizeof(strLine) ) )
{
  str = &strLine[0];
  iLineNr++;
  // comment and empty lines should be skipped
  if( !(strLine[0] == '\n' || strLine[0] == '#' || strLine[0]=='\0' ||
        Parse::gotoFirstNonSpace( &str ) == '\0' ) )
  {
    // there are ten different lines in a formation (see comment above)
    // all values for each line are parsed in one iteration
    // after all 10 lines are parsed, the sequence it is resetted.
    switch( iLineInFormation )
    {
      case 0: // first line is the number of the formation
        iForm = Parse::parseFirstInt( &str );
        break;
      case 1: // the x coordinate of the home pos for all the players
        for( i = 0 ; i < MAX_TEAMMATES ; i ++ )
          formations[iForm].setXPosHome(Parse::parseFirstDouble(&str), i);
        break;
      case 2: // the y coordinate of the home pos for all the players
        for( i = 0 ; i < MAX_TEAMMATES ; i ++ )
          formations[iForm].setYPosHome(Parse::parseFirstDouble(&str), i);
        break;
      case 3: // the player types for all the players
        for( i = 0 ; i < MAX_TEAMMATES ; i ++ )
          formations[iForm].setPlayerType(
              (PlayerT) Parse::parseFirstInt(&str), i);
        break;
      case 4: // the x attraction for all the player types
```

```
    for( i = 0 ; i < MAX_PLAYER_TYPES ; i ++ )
    {
     pt_info = formations[iForm].getPlayerTypeInfo( i );
     pt_info->setAttrX( Parse::parseFirstDouble( &str ) );
    }
    break;
   case 5: // the y attraction for all the player types
    for( i = 0 ; i < MAX_PLAYER_TYPES ; i ++ )
    {
     pt_info = formations[iForm].getPlayerTypeInfo( i );
     pt_info->setAttrY( Parse::parseFirstDouble( &str ) );
    }
    break;
   case 6: // stay behind the ball for all the player types
    for( i = 0 ; i < MAX_PLAYER_TYPES ; i ++ )
    {
     pt_info = formations[iForm].getPlayerTypeInfo( i );
     if( Parse::parseFirstInt( &str ) == 1 )
      pt_info->setBehindBall( true );
     else
      pt_info->setBehindBall( false );          ;
    }
    break;
   case 7: // the minimal x coordinate for all the player types
    for( i = 0 ; i < MAX_PLAYER_TYPES ; i ++ )
    {
     pt_info = formations[iForm].getPlayerTypeInfo( i );
     pt_info->setMinX( Parse::parseFirstDouble( &str ));
    }
    break;
   case 8:// the maximal x coordinate for all the player types
    for( i = 0 ; i < MAX_PLAYER_TYPES ; i ++ )
    {
     pt_info = formations[iForm].getPlayerTypeInfo( i );
     pt_info->setMaxX( Parse::parseFirstDouble( &str ));
    }
    break;
   default:
    cerr << "(Formations::readFormations) error line " << iLineNr <<endl;
    return false;
  }

  iLineInFormation++;          // go one line further
  if( iLineInFormation == 9 ) // each formation consists of ten lines
   iLineInFormation = 0;
 }
```

```
  }
  return true;
}

/*! This method sets the current formation.
    \param formation new current formation
    \return bool indicating whether the update was successful */
bool Formations::setFormation( FormationT formation )
{
  curFormation = formation;
  return true;
}

/*! This method returns the current formation.
    \return current formation  */
FormationT Formations::getFormation( ) const
{
  return curFormation;
}

/*! This method sets the player number of the agent in the current formation to
    'iNumber'.
    \param iNumber new player number for this agent
    \return bool indicating whether the update was succesfull */
bool Formations::setPlayerInFormation( int iNumber )
{
  iPlayerInFormation = iNumber;
  return true;
}

/*! This method returns the role number of the agent in the current formation
    \return player number for this agent in the current formation */
int Formations::getPlayerInFormation( ObjectT obj  ) const
{
  int i;
  if( obj == OBJECT_ILLEGAL )
    i = iPlayerInFormation;
  else
    i = SoccerTypes::getIndex( obj );

  return i;
}

/*! This method returns the player type for the specified object
    \return player type for the agent in the current formation */
PlayerT Formations::getPlayerType( ObjectT obj, FormationT ft ) const
```

```
{
  if( ft == FT_ILLEGAL )
    ft = curFormation;
  return formations[ ft ].getPlayerType(
    SoccerTypes::getIndex( obj ) );
}

/*! This method returns the player type for the agent in the current formation
    \return player type for the agent in the current formation */
PlayerT Formations::getPlayerType( int iIndex, FormationT ft ) const
{
  if( ft == FT_ILLEGAL )
    ft = curFormation;
  if( iIndex == -1  )
    iIndex = iPlayerInFormation;
  return formations[ ft ].getPlayerType( iIndex );
}

/*************************** TESTING PURPOSES
*********************************/

/*
int main( void )
{
  Formations fs( "formations.conf" );
  fs.show( cout );
}
*/
```

GenericValues.h

```
#ifndef _GENERIC_VALUES_
#define _GENERIC_VALUES_

#include <iostream>    // needed for output stream

using namespace std;

/*! This enumeration contains all generic values used throughout the code. */
enum GenericValueKind
{
  GENERIC_VALUE_DOUBLE  = 0,
  GENERIC_VALUE_STRING  = 1,
  GENERIC_VALUE_BOOLEAN = 2,
  GENERIC_VALUE_INTEGER = 3,
};

/*************************************************************************
/
/*******************   CLASS GENERICVALUET
*********************************/
/*************************************************************************
/

/*! This class contains a pointer to a variable of a generic type (double,
    char*, bool, int) and this pointer is associated with a string by which the
    variable can be reached. Several methods are defined which enable one to
    access the name and value of the variable. */
class GenericValueT
{
  // private member data
private:

  const char*      m_strName;  /*!< the name associated with the variable to
                                   which the class pointer points          */
  void*            m_vAddress; /*!< a pointer to a variable of a generic type*/
  GenericValueKind m_type;     /*!< the (generic) type of the variable to which
                                   the class pointer points                */

  // public methods
public:

  // constructor for the GenericValueT class
  GenericValueT( const char *strName, void *vAddress, GenericValueKind type );

  // destructor for the GenericValueT class
  ~GenericValueT( );
```

```
  // get methods for private member variables
  const char* getName ( );

  // methods to set/get the value of this generic variable
  bool  setValue( const char *strValue );
  char* getValue(       char *strValue );

  // display method
  void show( ostream& out, const char *strSeparator );
};

/*****************************************************************************
/
/*******************   CLASS GENERICVALUES
**********************************/
/*****************************************************************************
/

/*! This class contains a collection of GenericValueT objects. This
    makes it possible to reference variables using string names. The
    class is an abstract class which should not be instantiated. It is
    the subclass of this class which contains the actual variables. In
    order to add a reference to a variable the method 'addSetting'
    must be used which associates the variables in the subclass with
    string names. The GenericValues class is used to read in
    configuration files. This now becomes very easy as long as one
    makes sure that the names in the configuration file match the
    string names associated with the corresponding variables */
class GenericValues
{
  // private member data
private:

  char  *m_strClassName;    /*!< the name associated with this group of generic
                            values; this is usually the name of the
                            subclass which contains the actual variables*/
  GenericValueT ** m_values;/*!< a pointer to an array containing all generic
                            value pointers (GenericValueT objects)      */
  int   m_iValuesTotal;     /*!< the total number of generic values stored in
                            the collection so far                       */
  int   m_iMaxGenericValues;/*!< the number of generic values in the current
                            collection, i.e. the maximum number of values
                            that can be stored                          */

  GenericValueT* getValuePtr( const char *strName                       );
```

```
public:

  // constructor for the GenericValues class
  GenericValues          ( char *strName,            int iMaxValues      );

  // destructor for the GenericValues class
  virtual ~GenericValues  (                                          );

  // get methods for private member variables
  char* getClassName     (                                           );
  int   getValuesTotal   (                                          );

  // method for adding a generic value to the collection
  bool addSetting( const char *strName, void *vAddress, GenericValueKind t );

  // methods for reading and writing generic values and collections of values
  virtual char* getValue  ( const char *strName,       char *strValue       );
  virtual bool  setValue  ( const char *strName, const char *strValue       );
  virtual bool  readValues( const char *strFile, const char *strSeparator = 0);
  virtual bool  saveValues( const char *strFile, const char *strSeparator = 0,
                                        bool bAppend = true   );

  // display method
  virtual void  show     ( ostream& out,       const char *strSeparator    );
};

#endif
```

GenericValues.cpp

```
#include "GenericValues.h"
#include "Parse.h"          // needed for 'gotoFirstNonSpace'
#include <stdio.h>          // needed for 'sprintf'
#include <stdlib.h>         // needed for 'free'
#include <string.h>         // needed for 'strdup'
#include <ctype.h>          // needed for 'isdigit'
#include <fstream>          // needed for 'ifstream'

/*****************************************************************************
/
/*******************   CLASS GENERICVALUET
**********************************/
/*****************************************************************************
/

/*! Constructor for the GenericValueT class. It creates a GenericValueT object
    by setting all the private member variables to the values passed to the
    constructor.
    \param strName a string denoting the name associated with the generic
    variable to which the class pointer will point (the variable can be reached
    through this name)
    \param vAdress a (void) pointer to the actual generic variable
    \param t the (generic) type of the variable associated with the class
    pointer */
GenericValueT::GenericValueT( const char *str, void *vAddr, GenericValueKind t)
{
  m_strName  = strdup( str ); // strdup needed to alloc right amount of memory
  m_vAddress = vAddr;
  m_type     = t;
}

/*! Destructor for the GenericValueT class. It destroys a GenericValueT object
    by freeing all memory allocated to it. */
GenericValueT::~GenericValueT( )
{
  if( m_strName )
    free( ( char * ) m_strName );
}

/*! Get method for the 'm_strName' member variable.
    \return the name associated with the variable to which the class pointer
    points */
const char* GenericValueT::getName ( )
{
  return ( m_strName );
}
```

```
/*! This method sets the variable associated with this GenericValueT
    object to the value indicated by the given string argument. The
    value is always supplied as a string which is then converted into
    the right type for this GenericValueT object.

    \param strValue a string denoting the value to which the variable
    associated with this GenericValueT object must be set

    \return a boolean indicating whether the update was successful */
bool GenericValueT::setValue( const char *strValue )
{
  bool bReturn = true, b;

  // determine the generic type associated with this GenericValueT
  // object and apply the correct conversion of the string argument
  switch( m_type )
  {
    case GENERIC_VALUE_DOUBLE:
      *( double * ) m_vAddress = atof( strValue ? strValue : 0 );
      break;
    case GENERIC_VALUE_STRING:
      strcpy( ( char * ) m_vAddress, strValue );
      break;
    case GENERIC_VALUE_BOOLEAN:
      b = false;
      if( !strValue )
        ;
      else if( isdigit( strValue[ 0 ] ) )
        b = atoi( strValue ? strValue : 0 );
#ifdef WIN32
      else if( strcmp( strValue, "on"  ) == 0 ||
               strcmp( strValue, "true") == 0 ||
               strcmp( strValue, "yes" ) == 0 )
#else
      else if( strcasecmp( strValue, "on"  ) == 0 ||
               strcasecmp( strValue, "true") == 0 ||
               strcasecmp( strValue, "yes" ) == 0 )
#endif
        b = true;

      *( bool * ) m_vAddress = ( b == true ) ? true : false;
      break;
    case GENERIC_VALUE_INTEGER:
      *( int * ) m_vAddress = atoi( strValue ? strValue : 0 );
      break;
```

```
    default:
      bReturn = false;
  }

  return ( bReturn );
}

/*! This method determines the value of the variable associated with
    this GenericValueT object. The result is converted to a string
    which is put into the argument to the method (note that enough
    memory must be allocated for this char*). This same string is also
    returned by the method. This enables you to use the method as an
    argument to a function such as 'strcpy'.

    \param strValue a string which after the method call will contain
    the value of the variable associated with this GenericValueT
    object

    \return a string containing the value of the variable associated
    with this GenericValueT object. */
char* GenericValueT::getValue( char *strValue )
{
  // determine the generic type associated with this GenericValueT
  // object and apply the correct conversion into a string
  switch( m_type )
  {
    case GENERIC_VALUE_DOUBLE:
      sprintf( strValue, "%2f", *( double * ) m_vAddress );
      break;
    case GENERIC_VALUE_STRING:
      sprintf( strValue, "%s", ( char * ) m_vAddress );
      break;
    case GENERIC_VALUE_BOOLEAN:
      sprintf( strValue, "%d", *( int * ) m_vAddress );
      break;
    case GENERIC_VALUE_INTEGER:
      sprintf( strValue, "%d", *( int * ) m_vAddress );
      break;
    default:
      *strValue = '\0';
  }

  return ( strValue );
}

/*! This display method writes the name associated with the variable
```

```
    in this GenericValueT object together with its value to the output
    stream indicated by the first argument to the method. Name and
    value are separated by the separator given as the second
    argument. Note that boolean values are written as 'true' and
    'false'.

    \param out the output stream to which the information should be
    written

    \param strSeparator a string representing the separator which
    should be written between the name associated with the variable
    and its value */
void GenericValueT::show( ostream& out, const char *strSeparator )
{
  // write the name associated with the variable in this GenericValueT
  // object followed by the separator to the specified output stream
  out << m_strName << strSeparator;

  // determine the generic type associated with this GenericValueT
  // object and write the correct value to the specified output stream
  switch( m_type )
  {
    case GENERIC_VALUE_DOUBLE:
      out << *( double * ) m_vAddress;
      break;
    case GENERIC_VALUE_STRING:
      out << ( char * ) m_vAddress;
      break;
    case GENERIC_VALUE_BOOLEAN:
      out << ( ( *( bool * ) m_vAddress == true ) ? "true" : "false");
      break;
    case GENERIC_VALUE_INTEGER:
      out << *( int * ) m_vAddress;
      break;
    default:
      break;
  }

  out << endl;
}


/*****************************************************************************
/
/*******************   CLASS GENERICVALUES
*********************************/
```

```
/*****************************************************************************
/

/*! Constructor for the GenericValues class. It creates a GenericValues object.

    \param strName a string denoting the name associated with this group of
    generic values (this is usually the name of the subclass which contains the
    actual generic variables)
    \param iMaxValues an integer denoting the number of generic values in the
    current collection, i.e. the maximum number that can be stored */
GenericValues::GenericValues( char *strName, int iMaxValues )
{
  m_iValuesTotal = 0;   // total number of values in collection is set to zero

  if( strName )       // set the name of the collection
    m_strClassName = strdup( strName );

  m_iMaxGenericValues = iMaxValues;

  // a GenericValues object is a collection of GenericValueT objects
  m_values = new GenericValueT*[ iMaxValues ];
}

/*! Destructor for the GenericValues class. It destroys a GenericValues object
    by freeing all memory allocated to it. */
GenericValues::~GenericValues( void )
{
  for( int i = 0 ; i < getValuesTotal( ) ; i++ )
    delete m_values[ i ];
  delete m_values;

  if( m_strClassName )
    free( m_strClassName );
}

/*! Get method for the 'm_strClassName' member variable.
    \return the name associated with this group of generic values (this is
    usually the name of the subclass which contains the actual variables) */
char* GenericValues::getClassName( )
{
  return ( m_strClassName );
}

/*! Get method for the 'm_iValuesTotal' member variable.
    \return the number of generic values stored in the collection so far */
int GenericValues::getValuesTotal( )
```

```
{
  return ( m_iValuesTotal );
}

/*! This method adds a new generic value to the collection and
    provides a connection between a string name and the variable name
    of the generic value.

    \param strName a string denoting the name associated with the
    variable of the generic value which is added (the variable can be
    reached through this name)

    \param vAddress a (void) pointer to the actual variable itself

    \param type the (generic) type of the variable which is added

    \return a boolean indicating whether the generic value has been succesfully
    added */
bool GenericValues::addSetting( const char *strName, void *vAddress,
                                GenericValueKind type )
{
  if( getValuePtr( strName ) )              // setting already installed
  {
    cerr << "Setting '" << strName << "' already installed." << endl;
    return false;
  }
  if( m_iValuesTotal == m_iMaxGenericValues )    // buffer is full
  {
    cerr << "GenericValues::addSetting buffer for " << m_strClassName <<
            " is full (cannot add '" << strName << "')" << endl;
    return false;
  }

  m_values[ m_iValuesTotal++ ] = new GenericValueT( strName, vAddress, type );

  return ( true );
}

/*! This (private) method returns a pointer to the GenericValueT
    object of which the name associated with the variable matches the
    argument passed to the method.

    \param strName a string denoting the name associated with the
    variable of the GenericValueT object to which a pointer should be
    returned
```

```
    \return a pointer to the GenericValueT object of which the name
    associated with the variable matches the argument; 0 when a
    GenericValueT object with a variable associated with the given
    name does not exist */
GenericValueT* GenericValues::getValuePtr( const char *strName )
{
  GenericValueT *ptr = 0;

  // search through the collection for a GenericValueT object of which the name
  // associated with the variable matches the argument passed to the method
  for( int i = 0 ; i < getValuesTotal( ) ; i++ )
  {
    if( strcmp( m_values[ i ]->getName( ), strName ) == 0 )
    {
      ptr = m_values[ i ];
      break;
    }
  }

  return ( ptr );
}

/*! This method determines the value of the variable of which the name
    associated with it matches the first argument to the method. The value is
    converted to a string which is put into the second argument (note that
    enough memory must be allocated for this char*). This same string is also
    returned by the method. This enables you to use the method as an argument
    to a function such as 'strcpy'.
    \param strName a string denoting the name associated with the variable of
    which the value should be returned
    \param strValue a string which after the method call will contain the value
    of the variable of which the name associated with it matches the first
    argument to the method
    \return a string containing the value of the variable of which the name
    associated with it matches the first argument to the method */
char* GenericValues::getValue( const char *strName, char *strValue )
{
  GenericValueT *parptr;

  parptr = getValuePtr( strName );

  if( parptr )
    strValue = parptr->getValue( strValue );   // this method returns a string
  else
    strValue[ 0 ] = '\0';
```

```
  return ( strValue );
}

/*! This method sets the variable of which the name associated with it
    matches the first argument to the method to the value indicated by
    the second argument. The value is always supplied as a string
    which is then converted into the right type for the generic value
    in question.

    \param strName a string denoting the name associated with the
    variable which must be set

    \param strValue a string denoting the value to which the variable
    indicated by the first argument should be set

    \return a boolean indicating whether the update was successful */
bool GenericValues::setValue( const char *strName, const char *strValue )
{
  bool bReturn = false;
  GenericValueT *parptr;

  parptr = getValuePtr( strName );

  if( parptr )
    bReturn = parptr->setValue( strValue ); // string is converted to its type

  return ( bReturn );
}

/*! This method reads generic values from a file indicated by the
    first argument to the method and stores them in the collection. In
    this file the names associated with each value must be listed
    first followed by a separator and then the value. The names
    associated with each value should match the string names with
    which the corresponding generic value is added using 'addSetting'.

    \param strFile a string denoting the name of the file from which the values
    must be read
    \param strSeparator a string representing the separator which is written
    between name and value in the configuration file
    \return a boolean indicating whether the values have been read correctly */
bool GenericValues::readValues( const char *strFile, const char *strSeparator )
{
  ifstream in( strFile );

  if( !in )
```

```
 {
  cerr << "(GenericValues::readValues) Could not open file '" <<
  strFile << "'" << endl;
  return ( false );
 }

 bool  bReturn = true;
 char  strLine[ 256 ], strName[ 100 ], strValue[ 100 ];
 char* c;
 int   iLineNr = 0;

 // read each line in the configuration file and store the values
 while( in.getline( strLine, sizeof( strLine ) ) )
 {
  iLineNr++;

  // skip comment lines and empty lines; " #" is for server.conf version 7.xx
  if( !( strLine[ 0 ] == '\n' ||
         strLine[ 0 ] == '#'  ||
         strLine[ 0 ] == '\0' ||
         ( strlen( strLine ) > 1 &&
           strLine[ 0 ] == ' ' &&
           strLine[ 1 ] == '#' ) ) )
  {
   // convert all characters belonging to the separator to spaces
   if( strSeparator && ( c = strstr( strLine, strSeparator ) ) != NULL )
     for( size_t i = 0; i < strlen( strSeparator ); i++ )
       *( c + i ) = ' ';

   // read the name and value on this line and store the value
   if( !( sscanf( strLine, "%s%s", strName, strValue ) == 2 &&
          setValue( strName, strValue ) ) )
   {
    bReturn = false;
    cerr << "(GenericValues::readValues) '" << strFile << "' linenr "
         << iLineNr << ",  error in '" << strLine << "'" << endl;
   }
  }
 }

 return ( bReturn );
}

/*! This method writes the generic values in the current collection to
    a file indicated by the first argument to the method. The string
    name associated with the value and the value itself are separated
```

```
    by a separator which is specified as the second argument.

    \param strFile a string denoting the name of the file to which the
    generic values should be written

    \param strSeparator a string representing the separator which
    should be written between the name associated with each value and
    the value itself

    \param bAppend a boolean indicating whether the values should be
    appended to the given file (=true) or if current contents of the
    file should be overwritten (=false)

    \return a boolean indicating whether the values in the current
    collection have been correctly written to the given file */
bool GenericValues::saveValues( const char *strFile, const char *strSeparator,
                                bool bAppend )
{
  ofstream outf( strFile, ( bAppend == false ? ( ios::out )
                                     : ( ios::out | ios::app ) ) );

  if( !outf )
  {
    cerr << "Could not open file '" << strFile << "'" << endl;
    return ( false );
  }

  // values are written to file using 'show' (note that
  // output stream to write to is a file in this case)
  show( outf, strSeparator );

  return ( true );
}

/*! This display method writes all the generic values in the current
    collection to the output stream indicated by the first argument to
    the method. The name associated with each value and the value
    itself are separated by a separator which is specified as the
    second argument.

    \param out the output stream to which the current collection of
    generic values should be written

    \param strSeparator a string representing the separator which
    should be written between the name associated with each value and
    the value itself */
```

```
void GenericValues::show( ostream& out, const char *strSeparator )
{
  for( int i = 0; i < getValuesTotal( ); i++ )
    m_values[ i ]->show( out, strSeparator );
}

/********************* TESTING PURPOSES
*************************************/
/*
int main( void )
{
  GenericValues g("ServerSettings", 6);

  int    i = 10;
  char   c[256];
  sprintf( c, "hallo" );
  bool   b = true;
  double d = 10.14;

  cout << g.addSetting( "var1", &i, GENERIC_VALUE_INTEGER ) << endl;
  cout << g.addSetting( "var2", c, GENERIC_VALUE_STRING  ) << endl;
  cout << g.addSetting( "var3", &d, GENERIC_VALUE_DOUBLE  ) << endl;
  cout << g.addSetting( "var4", &b, GENERIC_VALUE_BOOLEAN ) << endl;
  g.show( cout, ":"  );

  g.setValue( "var1", "11"      );
  g.setValue( "var2", "hoi"     );
  g.setValue( "var3", "20.2342" );
  g.setValue( "var4", "false"   );
  g.show( cout, ":"  );

//  g.setIntegerValue( "var1", 22      );
//  g.setStringValue ( "var2", "hoi2"  );
//  g.setDoubleValue ( "var3", 30.2342 );
//  g.setBooleanValue( "var4", true    );
//  g.show( cout, ":"  );

//  g.readValues( "server.conf", ":" );
//  g.show( cout, ":"  );

  return ( 0 );
}

*/
```

WorldModel.h

```
#ifndef _WORLD_MODEL_
#define _WORLD_MODEL_

#include "Objects.h"        // needed for PlayerObject
#include "PlayerSettings.h" // needed for getPlayerDistTolerance
#include "Logger.h"         // needed for Log
#include "Formations.h"     // needed for getStrategicPosition (prediction)
#include <list>

#ifdef WIN32
  #include <windows.h>      // needed for mutex
#else
  #include <pthread.h>      // needed for pthread_mutex
  #include <sys/time.h>     // needed for gettimeofday
#endif


extern Logger Log;          // defined in Logger.C
extern Logger LogDraw;      // defined in Logger.C

#ifdef WIN32
/*! This function shall return the integral value (represented as a double)
    nearest x in the direction of the current rounding mode. The current
    rounding mode is implementation-defined.

    If the current rounding mode rounds toward negative infinity, then rint()
    shall be equivalent to floor() . If the current rounding mode rounds
    toward positive infinity, then rint() shall be equivalent to ceil().
    url: http://www.opengroup.org/onlinepubs/007904975/functions/rint.html */
inline double rint(double x)
{
   return floor(x+0.5);
}

/*! The drand48() function shall return non-negative,
    double-precision, floating-point values, uniformly distributed over the
    interval [0.0,1.0).
    url: http://www.opengroup.org/onlinepubs/007904975/functions/drand48.html*/
inline double drand48()
{
   return ((double)(rand() % 100)) / 100;
}
#endif

/*****************************************************************************
/
```

```
/********************** CLASS WORLDMODEL
************************************/
/*****************************************************************************
/

/*! The Class WorlModel contains all the RoboCup information that is available
    on the field. It contains information about the players, ball, flags and
    lines. Furthermore it contains methods to extract useful information.
    The (large amount of) attributes can be separated into different groups:
    - Environmental information: specific information about the soccer server
    - Match information: general information about the current state of a match
    - Object information: all the objects on the soccer field
    - Action information: actions that the agent has performed

    The methods can also be divided into different groups:
    - Retrieval methods: directly retrieving information of objects
    - Update methods: update world based on new sensory information
    - Prediction methods: predict future states based on past perceptions
    - High-Level methods: deriving high-level conclusions from basic worldstate
*/
class WorldModel
{
private:

  /*****************************************************************************/
  /************************** ATTRIBUTES
************************************/
  /*****************************************************************************/

  /////////////////////////// ENVIRONMENTAL INFORMATION ////////////////////////

  ServerSettings *SS;                 /*!< Reference to all server params*/
  PlayerSettings *PS;                 /*!< Reference to all client params*/
  HeteroPlayerSettings pt[MAX_HETERO_PLAYERS]; /*!< info hetero player types */
  Formations     *formations;          /*!< Reference to formation used   */

  /////////////////////////// CURRENT MATCH INFORMATION ////////////////////////

  // time information
  Time         timeLastSeeMessage;     /*!< server time of last see msg    */
  Time         timeLastRecvSeeMessage;  /*!< server time received see msg   */
  Time         timeLastSenseMessage;    /*!< server time of last sense msg  */
  Time         timeLastRecvSenseMessage;/*!< server time received sense msg */
  Time         timeLastHearMessage;     /*!< server time of last hear msg   */
  bool         bNewInfo;               /*!< indicates new info from server */
  Time         timeLastCatch;          /*!< time of last catch by goalie   */
```

```
Time        timeLastRefMessage;     /*!< time of last referee message   */

// player information
char        strTeamName[MAX_TEAM_NAME_LENGTH]; /*!< Team name          */
int         iPlayerNumber;          /*!< player number in soccerserver  */
SideT       sideSide;               /*!< side where the agent started   */

// match information
PlayModeT   playMode;               /*!< current play mode in the game  */
int         iGoalDiff;              /*!< goal difference                */


/////////////////////////// OBJECTS ////////////////////////////////////////

// dynamic objects
BallObject    Ball;                 /*!< information of the ball        */
AgentObject   agentObject;          /*!< information of the agent itself*/
PlayerObject  Teammates[MAX_TEAMMATES];/*!< information of all teammates   */
PlayerObject  Opponents[MAX_OPPONENTS];/*!< information of all opponents   */
PlayerObject  UnknownPlayers[MAX_TEAMMATES+MAX_OPPONENTS];
                            /*!< info unknown players are stored
                                 here til mapped to known player */
int         iNrUnknownPlayers;      /*!< number of unknown players      */

// fixed objects
FixedObject   Flags[MAX_FLAGS];     /*!< all flag information           */
FixedObject   Lines[MAX_LINES];     /*!< all line information           */

/////////////////////////// LOCALIZATION INFORMATION ////////////////////////

static const int iNrParticlesAgent = 100; /*!<nr of particles used to store
                                    agent position              */
static const int iNrParticlesBall  = 100; /*! nr of particles used to store
                                    ball position and velocity   */
VecPosition   particlesPosAgent[iNrParticlesAgent]; /*!< particles to store
                                        agent position      */
VecPosition   particlesPosBall[iNrParticlesBall];   /*! particles to store
                                        ball position       */
VecPosition   particlesVelBall[iNrParticlesBall];   /*! particles to store
                                        ball velocity       */
double        dTotalVarVel;
double        dTotalVarPos;
/////////////////////////// PREVIOUS ACTION INFORMATION /////////////////////

// arrays needed to keep track of actually performed actions.
```

```
  SoccerCommand queuedCommands[CMD_MAX_COMMANDS];   /*!<all performed
commands,
                                        set by ActHandler    */
  bool         performedCommands[CMD_MAX_COMMANDS];/*!< commands performed in
                                        last cycle, index is
                                        CommandT             */
  int          iCommandCounters[CMD_MAX_COMMANDS]; /*!< counters for all
                                        performed commands   */


  /////////////////////////// VARIOUS ////////////////////////////////////////

  // attributes only applicable to the coach
  Time         timeCheckBall;          /*!< time bsCheckBall applies to    */
  BallStatusT  bsCheckBall;            /*!< state of the ball              */

  // synchronization
#ifdef WIN32
  CRITICAL_SECTION mutex_newInfo;      /*!< mutex to protect bNewInfo       */
  HANDLE           event_newInfo;      /*!< event for bNewInfo              */
#else
  pthread_mutex_t  mutex_newInfo;      /*!< mutex to protect bNewInfo       */
  pthread_cond_t   cond_newInfo;       /*!< cond variable for bNewInfo      */
#endif
  bool         m_bRecvThink;           /*!< think received in sync. mode    */

  // communication
  char         m_strPlayerMsg[MAX_MSG];/*!< message communicated by player  */
  int          m_iCycleInMsg;          /*!< cycle contained in message      */
  Time         m_timePlayerMsg;        /*!< time corresponding to player msg*/
  int          m_iMessageSender;       /*!< player who send message         */
  char         m_strCommunicate[MAX_SAY_MSG];/*!< string for communicating  */

  // attention to
  ObjectT      m_objFocus;             /*!< object to which is focused.     */

  // offside line
  double       m_dCommOffsideX;        /*!< communicated offside line       */
  Time         m_timeCommOffsideX;     /*!< time Communicated offside line  */

  // feature information
  Feature      m_features[MAX_FEATURES];/*!< features applied to cur. cycle.*/

  // other
  bool         m_bPerformedKick;       /*!<Indicates whether ball was kicked*/
```

```
  // heterogeneous player information
  set<ObjectT>  m_setSubstitutedOpp;    /*!< Set of substituted opp players. */

  // upcoming view angle/quality
  SoccerCommand m_changeViewCommand;    /*!< last sent change_view command   */

  // side of penalty shootout
  SideT         m_sidePenalty;

public:

  bool          m_bWasCollision;        /*!<Indicates whether it is collision*/
  Time          m_timeLastCollision;    /*!< Last collision time            */

  // statistics
  int           iNrHoles;               /*!< nr of holes recorded           */
  int           iNrOpponentsSeen;       /*!< total nr of opponents seen     */
  int           iNrTeammatesSeen;       /*!< total nr of teammates seen     */

  // last received messages
  char          strLastSeeMessage  [MAX_MSG];  /*!< Last see message        */
  char          strLastSenseMessage[MAX_MSG];  /*!< Last sense_body message  */
  char          strLastHearMessage [MAX_MSG];  /*!< Last hear message       */


  /*************************************************************************/
  /************************** OPERATIONS
**************************************/
  /*************************************************************************/

  ////////////////////////// DIRECT RETRIEVAL (WorldModel.C) ////////////////////

private:
  // private methods
  Object*       getObjectPtrFromType      ( ObjectT       o          );

public:

  // get and set methods of attributes in WorldModel itself
  void          setTimeLastCatch          ( Time          time       );
  int           getTimeSinceLastCatch     (                          );
  bool          setTimeLastRefereeMessage ( Time          time       );
  Time          getTimeLastRefereeMessage (                          );
  Time          getCurrentTime            (                          );
  int           getCurrentCycle           (                          );
  bool          isTimeStopped             (                          );
```

```
bool        isLastMessageSee        (                       )const;
Time        getTimeLastSeeGlobalMessage(                    )const;
bool        setTimeLastSeeGlobalMessage( Time       time    );
Time        getTimeLastSeeMessage   (                       )const;
Time        getTimeLastRecvSeeMessage (                     )const;
bool        setTimeLastSeeMessage   ( Time       time       );
Time        getTimeLastSenseMessage (                       )const;
Time        getTimeLastRecvSenseMessage(                    )const;
bool        setTimeLastSenseMessage ( Time       time       );
Time        getTimeLastHearMessage  (                       )const;
bool        setTimeLastHearMessage  ( Time       time       );
int         getPlayerNumber         (                       )const;
bool        setPlayerNumber         ( int        i          );
SideT       getSide                 (                       )const;
bool        setSide                 ( SideT      s          );
const char* getTeamName             (                       )const;
bool        setTeamName             ( char       *str       );
PlayModeT   getPlayMode             (                       )const;
bool        setPlayMode             ( PlayModeT  pm         );
int         getGoalDiff             (                       )const;
int         addOneToGoalDiff        (                       );
int         subtractOneFromGoalDiff (                       );
int         getNrOfCommands         ( CommandT   c          )const;
bool        setNrOfCommands         ( CommandT   c,
                                      int        i          );
Time        getTimeCheckBall        (                       )const;
bool        setTimeCheckBall        ( Time       time       );
BallStatusT getCheckBallStatus      (                       )const;
bool        setCheckBallStatus      ( BallStatusT bs        );
bool        getRecvThink            (                       );
char*       getCommunicationString  (                       );
bool        setCommunicationString  ( char*      srt        );
ObjectT     getObjectFocus          (                       );
bool        setObjectFocus          ( ObjectT    obj        );

// iterate over a specific object set
ObjectT     iterateObjectStart      ( int        &iIndex,
                                      ObjectSetT g,
                                      double     dConf = -1.0,
                                                           bool       bForward = false );
ObjectT     iterateObjectNext       ( int        &iIndex,
                                      ObjectSetT g,
                                      double     dConf = -1.0,
                                                           bool       bForward = false );
void        iterateObjectDone       ( int        &iIndex    );
```

```
// get and set methods for agent information
ObjectT      getAgentObjectType       (                        )const;
int          getAgentIndex            (                        )const;
bool         setAgentObjectType       ( ObjectT       o        );
AngDeg       getAgentBodyAngleRelToNeck (                      )const;
AngDeg       getAgentGlobalNeckAngle  (                        )const;
AngDeg       getAgentGlobalBodyAngle  (                        );
Stamina      getAgentStamina          (                        )const;
TiredNessT   getAgentTiredNess        (                        )const;
double       getAgentEffort           (                        )const;
VecPosition  getAgentGlobalVelocity   (                        )const;
double       getAgentSpeed            (                        )const;
VecPosition  getAgentGlobalPosition   (                        )const;
bool         setAgentViewAngle        ( ViewAngleT    va       ) ;
ViewAngleT   getAgentViewAngle        (                        )const;
bool         setAgentViewQuality      ( ViewQualityT  vq       ) ;
ViewQualityT getAgentViewQuality      (                        )const;
double       getAgentViewFrequency    ( ViewAngleT    va = VA_ILLEGAL,
                                        ViewQualityT  vq = VQ_ILLEGAL  );
bool         getAgentArmMovable       (                        );
VecPosition  getAgentArmPosition      (                        );
int          getAgentArmExpires       (                        );

// get methods for ball information
VecPosition  getBallPos               (                        );
double       getBallSpeed             (                        );
AngDeg       getBallDirection         (                        );

// get method for different information about a specific object
Time         getTimeGlobalPosition    ( ObjectT       o        );
VecPosition  getGlobalPosition        ( ObjectT       o        );
Time         getTimeGlobalVelocity    ( ObjectT       o        );
VecPosition  getGlobalVelocity        ( ObjectT       o        );
double       getRelativeDistance      ( ObjectT       o        );
VecPosition  getRelativePosition      ( ObjectT       o        );
AngDeg       getRelativeAngle         ( ObjectT       o,
                                        bool         bWithBody = false);
Time         getTimeGlobalAngles      ( ObjectT       o        );
AngDeg       getGlobalBodyAngle       ( ObjectT       o        );
AngDeg       getGlobalNeckAngle       ( ObjectT       o        );
AngDeg       getGlobalAngle           ( ObjectT       o        );
double       getConfidence            ( ObjectT       o        );
bool         isKnownPlayer            ( ObjectT       o        );
ObjectT      getOppGoalieType         (                        );
ObjectT      getOwnGoalieType         (                        );
Time         getTimeLastSeen          ( ObjectT       o        );
```

```
Time          getTimeChangeInformation   ( ObjectT       o              );
VecPosition   getGlobalPositionLastSee   ( ObjectT       o              );
Time          getTimeGlobalPositionLastSee( ObjectT      o              );
VecPosition   getGlobalVelocityLastSee   ( ObjectT       o              );
AngDeg        getGlobalBodyAngleLastSee  ( ObjectT       o              );
int           getTackleExpires           ( ObjectT       o =OBJECT_ILLEGAL);
AngDeg        getGlobalArmDirection      ( ObjectT       o              );
Time          getTimeGlobalArmDirection  ( ObjectT       o              );
double        getProbTackleSucceeds      ( ObjectT       o = OBJECT_ILLEGAL,
                                                              int          iExtraCycles = 0,
                                           VecPosition    *pos = NULL    );
double        getProbTackleClosestOpp    ( int           iExtraCycles = 0 );
list<ObjectT> getListCloseOpponents      ( VecPosition    pos,
                                           double         dDist = 15     );

// set methods for objects
bool          setIsKnownPlayer           ( ObjectT       o,
                                           bool           isKnownPlayer  );
bool          setTimeLastSeen            ( ObjectT       o,
                                           Time           time           );
bool          setHeteroPlayerType        ( ObjectT       o,
                                           int            iPlayer        );

// formation information
PlayerT       getPlayerType              ( ObjectT       o =OBJECT_ILLEGAL);
bool          isInPlayerSet              ( ObjectT       o,
                                                              PlayerSetT   ps             );

// get method for information about goals
VecPosition   getPosOpponentGoal         (                              );
VecPosition   getPosOwnGoal              (                              );
double        getRelDistanceOpponentGoal (                              );
AngDeg        getRelAngleOpponentGoal    (                              );
ObjectT       getLastOpponentDefender    ( double        *dX = NULL     );

// get and methods about fixed specifications (heterogeneous players)
HeteroPlayerSettings getInfoHeteroPlayer ( int           iIndex         );
HeteroPlayerSettings getHeteroInfoPlayer ( ObjectT       obj            );
int           getHeteroPlayerType        ( ObjectT       obj            );
bool          setSubstitutedOpp          ( ObjectT       obj            );
ObjectT       getSubstitutedOpp          (                              );
double        getDashPowerRate           ( ObjectT       obj            );
double        getPlayerSpeedMax          ( ObjectT       obj            );
double        getPlayerDecay             ( ObjectT       obj            );
double        getMaximalKickDist         ( ObjectT       obj            );
double        getStaminaIncMax           ( ObjectT       obj            );
```

```
double      getPlayerSize           ( ObjectT       obj             );
double      getInertiaMoment        ( ObjectT       obj             );
double      getEffortMax            ( ObjectT       obj             );
double      getEffectiveMaxSpeed    ( ObjectT       obj,
                                                    bool        bWithNoise =false);

// get method about previous commands
bool        isQueuedActionPerformed (                           );

// methods that return truth values about current play mode
bool        isFreeKickUs            ( PlayModeT pm = PM_ILLEGAL     );
bool        isFreeKickThem          ( PlayModeT pm = PM_ILLEGAL     );
bool        isCornerKickUs          ( PlayModeT pm = PM_ILLEGAL     );
bool        isCornerKickThem        ( PlayModeT pm = PM_ILLEGAL     );
bool        isOffsideUs             ( PlayModeT pm = PM_ILLEGAL     );
bool        isOffsideThem           ( PlayModeT pm = PM_ILLEGAL     );
bool        isKickInUs              ( PlayModeT pm = PM_ILLEGAL     );
bool        isKickInThem            ( PlayModeT pm = PM_ILLEGAL     );
bool        isFreeKickFaultUs       ( PlayModeT pm = PM_ILLEGAL     );
bool        isFreeKickFaultThem     ( PlayModeT pm = PM_ILLEGAL     );
bool        isKickOffUs             ( PlayModeT pm = PM_ILLEGAL     );
bool        isKickOffThem           ( PlayModeT pm = PM_ILLEGAL     );
bool        isBackPassUs            ( PlayModeT pm = PM_ILLEGAL     );
bool        isBackPassThem          ( PlayModeT pm = PM_ILLEGAL     );
bool        isGoalKickUs            ( PlayModeT pm = PM_ILLEGAL     );
bool        isGoalKickThem          ( PlayModeT pm = PM_ILLEGAL     );
bool        isBeforeKickOff         ( PlayModeT pm = PM_ILLEGAL     );
bool        isDeadBallUs            ( PlayModeT pm = PM_ILLEGAL     );
bool        isDeadBallThem          ( PlayModeT pm = PM_ILLEGAL     );
bool        isPenaltyUs             ( PlayModeT pm = PM_ILLEGAL     );
bool        isPenaltyThem           ( PlayModeT pm = PM_ILLEGAL     );
bool        isFullStateOn           ( SideT     s  = SIDE_ILLEGAL   );

// various methods
bool        setChangeViewCommand    ( SoccerCommand  soc            );
SoccerCommand getChangeViewCommand  (                               );

SideT       getSidePenalty          (                               );
bool        setSidePenalty          ( SideT         side            );

//////////////////////////// UPDATE METHODS (WorldModelUpdate.C)////////////////

// processing new information about the objects (filling World Model).
void        processSeeGlobalInfo    ( ObjectT       o,
                                      Time          time,
                                      VecPosition   pos,
```

```
                        VecPosition    vel,
                        AngDeg         angBody,
                        AngDeg         angNeck        );
bool        processNewAgentInfo       ( ViewQualityT   vq,
                        ViewAngleT     va,
                        double         dStamina,
                        double         dEffort,
                        double         dSpeed,
                        AngDeg         angSpeed,
                        AngDeg         angHeadAngle,
                        int            iTackleExpires,
                        int            iArmMovable,
                        int            iArmExpires,
                        VecPosition    posArm         );
void        processNewObjectInfo      ( ObjectT        o,
                        Time           time,
                        double         dDist,
                        int            iDir,
                        double         dDistChange,
                        double         dDirChange,
                        AngDeg         angRelBodyAng,
                        AngDeg         angRelNeckAng,
                        bool           isGoalie,
                        ObjectT        objMin,
                        ObjectT        objMax,
                        double         dPointDir,
                        bool           isTackling      );
bool        processPerfectHearInfo    ( ObjectT        o,
                        VecPosition    pos,
                        double         dConf,
                        bool           bIsGoalie=0     );
bool        processPerfectHearInfoBall ( VecPosition    pos,
                        VecPosition    vel,
                        double         dConf          );
bool        processUnsureHearInfo     ( ObjectT        o,
                        VecPosition    pos,
                        double         dConf          );
bool        processNewHeteroPlayer    ( int            iIndex,
                        double         dPlayerSpeedMax,
                        double         dStaminaIncMax,
                        double         dPlayerDecay,
                        double         dInertiaMoment,
                        double         dDashPowerRate,
                        double         dPlayerSize,
                        double         dKickableMargin,
                        double         dKickRand,
```

```
                                double      dExtraStamina,
                                double      dEffortMax,
                                double      dEffortMin      );
  void      processCatchedBall      ( RefereeMessageT rm,
                                Time        time        );
  void      processQueuedCommands   ( SoccerCommand  commands[],
                                int         iCommands       );
  bool      storePlayerMessage      ( int       iPlayer,
                                char        *strMsg,
                                int         iCycle      );
  bool      processPlayerMessage    (                       );
  bool      processRecvThink        ( bool      b           );

  // update methods (defined in WorldModelUpdate.C)
  bool      updateAll               (                       );

  // update with prediction for new cycle
  bool      updateAfterSenseMessage (                       );

private:
  // update with new visual information
  void      processLastSeeMessage   (                       );
  bool      updateAfterSeeMessage   (                       );
  bool      updateAgentObjectAfterSee (                     );
  bool      updateDynamicObjectAfterSee( ObjectT    o           );

  // update new cycle
  void      processLastSenseMessage (                       );
  bool      updateAgentAndBallAfterSense(                   );
  bool      updateBallAfterKick     ( SoccerCommand  soc        );
  bool      updateDynamicObjectForNextCycle( ObjectT    o,
                                int         iCycle      );
  bool      updateBallForCollision  ( VecPosition   posAgent    );

  // update from relative to global and vice versa
  bool      updateRelativeFromGlobal (                      );
  bool      updateObjectRelativeFromGlobal( ObjectT     o           );

  // methods to determine agent state from perceived information
  bool      calculateStateAgent     ( VecPosition   *posGlobal,
                                VecPosition   *velGlobal,
                                AngDeg      *angGlobal      );
  void      initParticlesAgent      ( AngDeg        angGlobal       );
  void      initParticlesAgent      ( VecPosition   posInitital     );
  int       checkParticlesAgent     ( AngDeg        angGlobalNeck   );
  void      updateParticlesAgent    ( VecPosition   vel,
```

```
                        bool        bAfterSense     );
VecPosition  averageParticles       ( VecPosition   posArray[],
                        int         iLength         );
void        resampleParticlesAgent  ( int           iLeft           );
bool        calculateStateAgent2    ( VecPosition   *posGlobal,
                        VecPosition   *velGlobal,
                        AngDeg        *angGlobal    );
bool        calculateStateAgent3    ( VecPosition   *posGlobal,
                        VecPosition   *velGlobal,
                        AngDeg        *angGlobal    );
VecPosition  calculatePosAgentWith2Flags( ObjectT       objFlag1,
                        ObjectT       objFlag2      );
AngDeg      calculateAngleAgentWithPos ( VecPosition    pos             );


// methods to determine ball state from perceived information
bool        calculateStateBall      ( VecPosition   *posGlobal,
                        VecPosition   *velGlobal    );
void        initParticlesBall       ( VecPosition   posArray[],
                        VecPosition   velArray[],
                        int         iLength         );
void        checkParticlesBall      ( VecPosition   posArray[],
                        VecPosition   velArray[],
                        int         iLength,
                        int         *iNrLeft        );
void        updateParticlesBall     ( VecPosition   posArray[],
                        VecPosition   velArray[],
                        int         iLength,
                        double      dPower,
                        AngDeg      ang             );
void        resampleParticlesBall   ( VecPosition   posArray[],
                        VecPosition   velArray[],
                        int         iLength,
                        int         iLeft           );
ObjectT     getMaxRangeUnknownPlayer  ( ObjectT     obj,
                        char*       strMsg          );
VecPosition  calculateVelocityDynamicObject( ObjectT    o               );


// methods to determine player state from perceived information
bool        calculateStatePlayer    ( ObjectT       o,
                        VecPosition   *posGlobal,
                        VecPosition   *velGlobal    );

// methods to calculate noise range (quantized noise)
bool        getMinMaxDistQuantizeValue ( double     dInput,
```

```
                                double      *dMin,
                                double      *dMax,
                                double      x1,
                                double      x2          );
  bool      getMinMaxDirChange      ( double      dOutput,
                                double      *dMin,
                                double      *dMax,
                                double      x1          );
  bool      getMinMaxDistChange     ( double      dOutput,
                                double      dDist,
                                double      *dMin,
                                double      *dMax,
                                double      x1,
                                double      xDist1,
                                double      xDist2      );
  double    invQuantizeMin          ( double      dOutput,
                                double      dQuantizeStep   );
  double    invQuantizeMax          ( double      dOutput,
                                double      dQuantizeStep   );


public:
  // various update methods
  void      mapUnknownPlayers       ( Time        time        );
  bool      updateSSToHeteroPlayerType ( int        iPlayerType     );
  bool      resetTimeObjects        (                       );
  void      removeGhosts            (                       );


  /////////////////////////// PREDICTIONS (WorldModelPredict.C) /////////////////

  // predictions of worldmodel about future states of different objects
  bool      predictStateAfterCommand  ( SoccerCommand  com,
                                VecPosition   *pos,
                                VecPosition   *vel,
                                AngDeg        *angGlobalBody,
                                AngDeg        *angGlobalNeck,
                                ObjectT       obj=OBJECT_ILLEGAL,
                                Stamina       *sta = NULL     );
  bool      predictAgentStateAfterCommand(SoccerCommand com,
                                VecPosition   *pos,
                                VecPosition   *vel,
                                AngDeg        *angBody,
                                AngDeg        *angNeck,
                                Stamina       *sta        );
  bool      predictObjectStateAfterCommand( ObjectT     obj,
```

```
                        SoccerCommand com,
                        VecPosition    *pos,
                        VecPosition    *vel,
                        AngDeg         *angBody,
                        AngDeg         *angNeck,
                        Stamina        *sta          );
VecPosition  predictAgentPosAfterCommand( SoccerCommand  com          );
void         predictStateAfterDash     ( double         dActualPower,
                        VecPosition    *pos,
                        VecPosition    *vel,
                        Stamina        *sta,
                        double         dDirection,
                        ObjectT        o=OBJECT_ILLEGAL);
void         predictStateAfterTurn     ( AngDeg         dSendAngle,
                        VecPosition    *pos,
                        VecPosition    *vel,
                        AngDeg         *angBody,
                        AngDeg         *angNeck,
                        ObjectT        obj=OBJECT_ILLEGAL,
                        Stamina        *sta = NULL      );
void         predictBallInfoAfterCommand( SoccerCommand  soc,
                        VecPosition    *pos = NULL,
                        VecPosition    *vel = NULL      );
VecPosition  predictPosAfterNrCycles    ( ObjectT        o,
                        double         dCycles,
                        int            iDashPower = 100,
                        VecPosition    *posIn = NULL,
                        VecPosition    *velIn = NULL,
                        bool           bUpdate = true   );
VecPosition  predictAgentPos            ( int            iCycles,
                        int            iDashPower = 0   );
VecPosition  predictFinalAgentPos       ( VecPosition    *pos = NULL,
                        VecPosition    *vel = NULL      );
int          predictNrCyclesForDistance ( ObjectT        o,
                        double         dDist,
                        double         dCurSpeed        );
int          predictNrCyclesToPoint     ( ObjectT        o,
                        VecPosition    posTo            );
int          predictNrCyclesToObject    ( ObjectT        objFrom,
                        ObjectT        objTo            );
void         predictStaminaAfterDash    ( double         dPower,
                        Stamina        *sta             );
SoccerCommand predictCommandTurnTowards ( ObjectT        obj,
                        VecPosition    posTo,
                        int            iCycles,
                        double         dDistBack,
```

```
                                bool          bMoveBack,
                                VecPosition   *pos = NULL,
                                VecPosition   *vel = NULL,
                                AngDeg        *angBody = NULL  );
SoccerCommand predictCommandToMoveToPos  ( ObjectT       obj,
                                VecPosition   posTo,
                                int           iCycles,
                                double        dDistBack = 2.5,
                                bool          bMoveBack = false,
                                VecPosition   *pos = NULL,
                                VecPosition   *vel = NULL,
                                AngDeg        *angBody = NULL  );
SoccerCommand predictCommandToInterceptBall( ObjectT      obj,
                                SoccerCommand  soc,
                                int           *iCycles = NULL ,
                                VecPosition   *posIntercept=NULL,
                                VecPosition   *pos = NULL,
                                VecPosition   *vel = NULL,
                                AngDeg        *angBody = NULL  );
bool        isCollisionAfterCommand   ( SoccerCommand  soc            );


////////////////////////// HIGH-LEVEL METHODS (WorldModelHighLevel.C) ///////

// methods that return the number of players in a certain area of the field
int         getNrInSetInRectangle     ( ObjectSetT     objectSet,
                                Rect          *rect = NULL     );
int         getNrInSetInCircle        ( ObjectSetT     objectSet,
                                Circle        c              );
int         getNrInSetInCone          ( ObjectSetT     objectSet,
                                double        dWidth,
                                VecPosition   start,
                                VecPosition   end            );
bool        isEmptySpace              ( ObjectT        obj,
                                AngDeg        ang,
                                double        dDist = 4.0      );
bool        coordinateWith            ( ObjectT        obj            );

// method that return the closest or fastest player to a certain pos orobject
ObjectT     getClosestInSetTo         ( ObjectSetT     objectSet,
                                ObjectT       o ,
                                double        *dDist = NULL,
                                double        dConfThr = -1.0  );
ObjectT     getClosestInSetTo         ( ObjectSetT     objectSet,
                                VecPosition   pos,
                                double        *dDist = NULL,
```

```
                              double        dConfThr = -1.0  );
ObjectT      getClosestInSetTo         ( ObjectSetT     objectSet,
                              Line          l,
                              VecPosition    pos1,
                              VecPosition    pos2,
                              double        *dDistToLine= NULL,
                              double        *dDistPos1To=NULL);
ObjectT      getClosestRelativeInSet    ( ObjectSetT     set,
                              double        *dDist = NULL    );
ObjectT      getSecondClosestInSetTo    ( ObjectSetT     objectSet,
                              ObjectT       o,
                              double        *dDist = NULL,
                              double        dConfThr = -1.0  );
ObjectT      getSecondClosestRelativeInSet( ObjectSetT   set,
                              double        *dDist = NULL    );
void         createInterceptFeatures    (                          );
ObjectT      getFastestInSetTo         ( ObjectSetT     objectSet,
                              ObjectT       o,
                              int           *iCycles = NULL  );
ObjectT      getFastestInSetTo         ( ObjectSetT     objectSet,
                              VecPosition    pos,
                              VecPosition    vel,
                              double        dDecay,
                              int           *iCycles = NULL  );
ObjectT      getFurthestInSetTo        ( ObjectSetT     objectSet,
                              ObjectT       o ,
                              double        *dDist = NULL,
                              double        dConfThr = -1.0  );
ObjectT      getFurthestRelativeInSet   ( ObjectSetT     set,
                              double        *dDist = NULL    );
VecPosition   getPosClosestOpponentTo    ( double        *dDist = NULL,
                              ObjectT       o =OBJECT_ILLEGAL);
double       getMaxTraveledDistance     ( ObjectT       o              );
// various methods dealing with sets of objects
ObjectT      getFirstEmptySpotInSet     ( ObjectSetT     objectSet,
                              int           iUnknownPlayer=-1);

// boolean methods that indicate whether an object fulfills a  constraint
bool         isVisible                 ( ObjectT       o              );
bool         isBallKickable            (                          );
bool         isBallCatchable           (                          );
bool         isBallHeadingToGoal       (                          );
bool         isBallInOurPossesion      (                          );
bool         isBallInOwnPenaltyArea     (                          );
bool         isInOwnPenaltyArea        ( VecPosition    pos           );
bool         isInTheirPenaltyArea      ( VecPosition    pos           );
```

```
  bool        isConfidenceGood        ( ObjectT                    );
  bool        isConfidenceVeryGood    ( ObjectT                    );
  bool        isOnside                ( ObjectT                    );
  bool        isOpponentAtAngle       ( AngDeg       ang,
                                        double       dDist          );

  // various methods that supply specific information about the field
  Time        getTimeFromConfidence   ( double       dConf          );
  double      getOffsideX             ( bool         bIncludeComm=true);
  VecPosition getOuterPositionInField ( VecPosition  pos,
                                        AngDeg       ang,
                                        double       dDist = 3.0,
                                        bool         bWithPenalty=true);
  AngDeg      getDirectionOfWidestAngle ( VecPosition  posOrg,
                                        AngDeg       angMin,
                                        AngDeg       angMax,
                                        AngDeg       *ang,
                                        double       dDist          );
  bool        isInField               ( VecPosition  pos,
                                        double       dMargin = 1    );
  bool        isBeforeGoal            ( VecPosition  pos            );

  // strategic positioning
  VecPosition getStrategicPosition    ( ObjectT      obj,
                                                     FormationT   ft = FT_ILLEGAL
);
  VecPosition getStrategicPosition    ( int          iPlayer = -1,
                                                     FormationT   ft = FT_ILLEGAL
);
  VecPosition getMarkingPosition      ( VecPosition  pos,
                                                     double       dDist,
                                                     MarkT        mark          );
  int         getClosestPlayerInFormationTo( VecPosition pos,
                                        bool         bWithGoalie = 1,
                                        ObjectT      obj=OBJECT_ILLEGAL,
                                                     PlayerSetT   ps = PS_ALL,
                                                     FormationT   ft = FT_ILLEGAL
);


  // methods  for computing the actual argument for a soccer command.
  double      getActualKickPowerRate  (                            );
  double      getKickPowerForSpeed    ( double       dDesiredSpeed  );
  double      getKickSpeedToTravel    ( double       dDistance,
                                        double       dEndSpeed      );
  double      getFirstSpeedFromEndSpeed ( double       dEndSpeed,
```

```
                                double        dCycles,
                                double        dDecay = -1.0    );
double        getFirstSpeedFromDist      ( double        dDist,
                                double        dCycles,
                                double        dDecay = -1.0    );
double        getEndSpeedFromFirstSpeed  ( double        dFirstSpeed,
                                double        dCycles          );
AngDeg        getAngleForTurn            ( AngDeg        angDesiredAngle,
                                double        dSpeed,
                                ObjectT       o=OBJECT_ILLEGAL);
AngDeg        getActualTurnAngle         ( AngDeg        angActualAngle,
                                double        dSpeed,
                                ObjectT       o=OBJECT_ILLEGAL);
double        getPowerForDash            ( VecPosition   posRelTo,
                                AngDeg        angBody,
                                VecPosition   vel,
                                double        dEffort,
                                int           iCycles = 1      );


/////////////////////////// VARIOUS (WorldModel.C) ////////////////////////////

// constructor
WorldModel                      ( ServerSettings *ss,
                                PlayerSettings *ps,
                                Formations     *fs             );
~WorldModel                     (                              );

// print information about WorldModel
void          show              ( ostream       &os = cout      );
void          show              ( ObjectSetT    set,
                                ostream       &os = cout      );
void          showQueuedCommands        ( ostream       &os = cout      );
void          show              ( ObjectT       o,
                                ostream       &os = cout      );

// methods that deal with timing information
bool          waitForNewInformation     (                              );

// methods that deal with debugging
void          logObjectInformation      ( int           iLogLevel,
                                ObjectT       o               );
void          logDrawInfo               ( int           iLogLevel       );
void          logCoordInfo              ( int           iLogLevel       );
bool          logCircle                 ( int           iLogLevel,
                                VecPosition   pos,
```

```
                                double        dRadius,
                                bool          bAll = false     );
  bool        logLine                ( int           iLogLevel,
                                VecPosition   pos1,
                                VecPosition   pos2,
                                bool          bAll = false     );
  bool        logDrawBallInfo          ( int           iLogLevel       );
  void        drawCoordinationGraph    (                               );

  // specific static variables
  char m_colorPlayers[11][8];      /*!< color information with which each
                             player should draw its relevant info*/
  int m_iMultX;                    /*!< This variable denotes with which value
                             the x coordinates of the draw
                             information should be multiplied in
                             order to convert the coordinates to
                             the coordinate system of the monitor*/
  int m_iMultY;                    /*!< This variable denotes with which value
                             the y coordinates of the draw
                             information should be multiplied in
                             order to convert the coordinates to
                             the coordinate system of the monitor*/

  bool        isFeatureRelevant       ( FeatureT       type           );
  Feature     getFeature              ( FeatureT       type           );
  bool        setFeature              ( FeatureT       type,
                                  Feature        feature        );
};

#endif
```

```
#include<stdio.h>    // needed for printf
#include<errno.h>    // needed for ETIMEDOUT
#ifdef WIN32
  #include<windows.h>
#else
  #include<strings.h>  // needed for strcpy
  #include<pthread.h>  // needed for pthread_mutex_init
#endif
#include<string.h>   // needed for strcpy
#include<math.h>     // needed for erf
#include<map>        // needed for map
#include"WorldModel.h"

/*****************************************************************************
/
/********************** CLASS WORLDMODEL
************************************/
/*****************************************************************************
/

/*! This constructor creates the worldmodel, all variables are initialized by
    default values
    \param ss reference to class in which all server parameters are stored
    \param ps reference to class in which all client parameters are stored
    \param fs reference to class in which all formation information is stored*/
WorldModel::WorldModel( ServerSettings *ss, PlayerSettings *ps,
                        Formations *fs):agentObject(  )
{
  dTotalVarVel   = 0.0;
  dTotalVarPos   = 0.0;
  SS             = ss;
  PS             = ps;
  formations     = fs;
  bNewInfo       = false;

  setSide        ( SIDE_ILLEGAL ); // is set by init message
  strTeamName[0] = '\0';
  setPlayMode    ( PM_BEFORE_KICK_OFF );
  iGoalDiff      = 0;
  m_sidePenalty = SIDE_ILLEGAL;

  int i;
  for( i = 0; i < MAX_TEAMMATES ; i ++ )
    Teammates[i].setType( SoccerTypes::getTeammateObjectFromIndex( i ) );
  for( i = 0; i < MAX_OPPONENTS ; i ++ )
    Opponents[i].setType( SoccerTypes::getOpponentObjectFromIndex( i ) );
```

```
  for( i = 0; i < MAX_OPPONENTS + MAX_TEAMMATES ; i ++ )
    UnknownPlayers[i].setType( OBJECT_ILLEGAL );
  for( i = 0; i < MAX_FLAGS; i ++ )
    Flags[i].setType( OBJECT_ILLEGAL );
  for( i = 0; i < MAX_LINES; i ++ )
    Lines[i].setType( OBJECT_ILLEGAL );

  iNrUnknownPlayers     = 0;

  Ball.setType            ( OBJECT_BALL    );
  agentObject.setType     ( OBJECT_ILLEGAL );
  agentObject.setStamina  ( Stamina(SS->getStaminaMax(),1.0,1.0) );

  for( i = 0 ; i < CMD_MAX_COMMANDS ; i ++ )
  {
    queuedCommands[i].commandType = (CommandT)i;
    performedCommands[i]          = false;
    iCommandCounters[i]           = 0;
  }

  iNrHoles              = 0;
  iNrOpponentsSeen      = 0;
  iNrTeammatesSeen      = 0;
  bsCheckBall           = BS_ILLEGAL;

  // initialize the mutex for bNewInfo
#ifdef WIN32
  InitializeCriticalSection( &mutex_newInfo );
  event_newInfo = CreateEvent( NULL, TRUE, FALSE, NULL );
#else
  pthread_mutex_init( &mutex_newInfo, NULL );
  pthread_cond_init ( &cond_newInfo,  NULL );
#endif
  m_bRecvThink          = false;
  timeLastSenseMessage  = Time( 0, 1 );

  strcpy( m_colorPlayers[0], "ffb0b0" );   // pink
  strcpy( m_colorPlayers[1], "ffb000" );   // orange
  strcpy( m_colorPlayers[2], "00b000" );   // darkgreen
  strcpy( m_colorPlayers[3], "00ffff" );   // lightblue
  strcpy( m_colorPlayers[4], "ff00ff" );   // purple
  strcpy( m_colorPlayers[5], "ffff00" );   // yellow
  strcpy( m_colorPlayers[6], "ffffff" );   // white
  strcpy( m_colorPlayers[7], "0000ff" );   // blue
  strcpy( m_colorPlayers[8], "00ff00" );   // green
  strcpy( m_colorPlayers[9], "ff0000" );   // red
```

```
  strcpy( m_colorPlayers[10],"aaaaaa" );   // gray

  for( i = 0 ; i < MAX_FEATURES ; i ++ )
  {
    m_features[i].setTimeSee( Time( UnknownTime, 0 ) );
    m_features[i].setTimeSense( Time( UnknownTime, 0 ) );
  }
}

/*! Destructor */
WorldModel::~WorldModel()
{
#ifdef WIN32
  DeleteCriticalSection( &mutex_newInfo );
#endif
}

/*! This method returns a pointer to the Object information of the object
    type that is passed as the first argument.
    \param o ObjectType of which information should be returned
    \return pointer to object information of supplied ObjectT argument */
Object* WorldModel::getObjectPtrFromType( ObjectT o )
{
  Object *object = NULL;
  if( o == OBJECT_ILLEGAL )
    return NULL;

  if( SoccerTypes::isKnownPlayer( o ) )
  {
    if( o == agentObject.getType() )
      object = &agentObject;
    else if( SoccerTypes::isTeammate( o ) )
      object = &Teammates[SoccerTypes::getIndex(o)];
    else
      object = &Opponents[SoccerTypes::getIndex(o)];
  }
  else if( SoccerTypes::isFlag( o ) )
    object = &Flags[SoccerTypes::getIndex(o)];
  else if( SoccerTypes::isLine( o ) )
    object = &Lines[SoccerTypes::getIndex(o)];
  else if( SoccerTypes::isBall( o ) )
    object = &Ball;
  else if( o == OBJECT_OPPONENT_GOALIE )
    return getObjectPtrFromType( getOppGoalieType() );
  else if( o == OBJECT_TEAMMATE_GOALIE )
    return getObjectPtrFromType( getOwnGoalieType() );
```

```
 return object;
}

/*! This method sets the time of the last catch cycle. This information is
    received by the SenseHandler when the referee has sent this message.
    After a catch, the goalie is not allowed to catch the ball for
    catch_ban_cycles (defined in ServerSettings).
    \param iTime time the ball was catched. */
void WorldModel::setTimeLastCatch( Time time )
{
  timeLastCatch = time;
}

/*! This method returns the number of cycles since the last catch.
    \return cycles since last catch. */
int WorldModel::getTimeSinceLastCatch()
{
  if( timeLastCatch.getTime() == -1 )
    return 1000;
  return timeLastSenseMessage - timeLastCatch;
}

/*! This method sets the time of the last received referee message. This
    information is received by the SenseHandler.
    \param iTime time the referee sent the last message. */
bool WorldModel::setTimeLastRefereeMessage( Time time )
{
  timeLastRefMessage = time;
  return true;
}

/*! This method returns the time of the last received referee message.
    \return time of last received referee message. */
Time WorldModel::getTimeLastRefereeMessage( )
{
  return timeLastRefMessage;
}

/*! This method returns the current time. In case of a player this is the
    time of the last sense message, in case of the coach this is the time of
    the last see_global message.
    \return actual time */
Time WorldModel::getCurrentTime()
{
  if( getPlayerNumber() == 0 )
    return getTimeLastSeeGlobalMessage();
```

```
  else
    return getTimeLastRecvSenseMessage();
}

/*! This method returns the current cycle number. In case of a player this is
    the cycle of the last sense message, in case of the coach this is the cycle
    of the last see_global message.
    \return actual time */
int WorldModel::getCurrentCycle()
{
  return getCurrentTime().getTime();
}


/*! This method returns whether the time of the server stands
    still. This occurs during non play-on modes (kick_in, kick_off,
    etc.).

    \return bool indicating whether time of the server stands still. */
bool WorldModel::isTimeStopped()
{
  return getCurrentTime().isStopped();
}

/*! This method returns whether the last received message was a see or not.
    \return bool indicating whether the last received message was a see.*/
bool WorldModel::isLastMessageSee() const
{
  return getTimeLastSeeMessage() == getTimeLastSenseMessage() ;
}

/*! This method returns the time of the last see global message.
    This message can only be received by the coach.
    \return time of last see_global message */
Time WorldModel::getTimeLastSeeGlobalMessage( ) const
{
  return getTimeLastRecvSeeMessage();
}

/*! This method sets the time of the last see_global message.
    \param time see message has arrived
    \return true when update was succesful */
bool WorldModel::setTimeLastSeeGlobalMessage( Time time )
{
  updateRelativeFromGlobal( );
  return setTimeLastSeeMessage( time ); // set see message
```

```
}

/*! This method returns the time of the last see message
    \return time of last see message */
Time WorldModel::getTimeLastSeeMessage( ) const
{
  return timeLastSeeMessage;
}

/*! This method returns the time of the last received see message.
    The difference with getTimeLastSeeMessage is that that method returns
    the last see message that has been updated in the world model. In most
    cases these are equal.
    \return time of last received sense message */
Time WorldModel::getTimeLastRecvSeeMessage( ) const
{
  return timeLastRecvSeeMessage ;
}
/*! This method sets the time of the last see message. It also sends a
    condition signal to indicate that variable bNewInfo has
    changed. When main thread is stopped in waitForNewInformation, it
    is unblocked.

    \param time see message has arrived
    \return true when update was succesful */
bool WorldModel::setTimeLastSeeMessage( Time time )
{
  timeLastRecvSeeMessage  = time;
  if( SS->getSynchMode() == false )
  {
#ifdef WIN32
    //EnterCriticalSection( &mutex_newInfo );
    bNewInfo            = true;
    SetEvent            (  event_newInfo );
    //LeaveCriticalSection( &mutex_newInfo );
#else
    pthread_mutex_lock  ( &mutex_newInfo );
    bNewInfo            = true;
    pthread_cond_signal ( &cond_newInfo );
    pthread_mutex_unlock( &mutex_newInfo );
#endif
  }

  return true;
}
```

```
/*! This method returns the time of the last sense message
    \return time of last sense message */
Time WorldModel::getTimeLastSenseMessage( ) const
{
  return timeLastSenseMessage ;
}

/*! This method returns the time of the last received sense message.
    The difference with getTimeLastSenseMessage is that that method returns
    the last sense message that has been updated in the world model. In most
    cases these are equal.
    \return time of last received sense message */
Time WorldModel::getTimeLastRecvSenseMessage( ) const
{
  return timeLastRecvSenseMessage ;
}


/*! This method sets the time of the last sense message. It also send a
    condition signal to indicate that variable bNewInfo has changed.
    When main thread is stopped in waitForNewInformation, it is
    unblocked.
    \param time sense message has arrived
    \return true when update was succesful */
bool WorldModel::setTimeLastSenseMessage( Time time )
{
  timeLastRecvSenseMessage = time;
  if( SS->getSynchMode() == false )
  {
#ifdef WIN32
    //EnterCriticalSection( &mutex_newInfo );
    bNewInfo = true;
    SetEvent           (  event_newInfo );
  //LeaveCriticalSection( &mutex_newInfo );
#else
    pthread_mutex_lock  ( &mutex_newInfo );
    bNewInfo = true;
    pthread_cond_signal ( &cond_newInfo );
    pthread_mutex_unlock( &mutex_newInfo );
#endif
  }

  return true;
}

/*! This method returns the time of the last hear message
```

```
    \return time of last hear message */
Time WorldModel::getTimeLastHearMessage( ) const
{
  return timeLastHearMessage ;
}

/*! This method sets the time of the last hear message.
    \param time hear message has arrived
    \return true when update was succesful */
bool WorldModel::setTimeLastHearMessage( Time time )
{
  timeLastHearMessage = time;
  return true;
}

/*! This method returns the player number of the agent. The player number is
    the fixed number which is given by the server after initialization.
    \return player number of this player. */
int WorldModel::getPlayerNumber( ) const
{
  return iPlayerNumber;
}

/*! This method sets the player number of the agent. This value is available in
    the conformation message sent by the soccerserver after the initialization.
    \param i new player number of the agent.
    \return bool indicating whether player number was set. */
bool WorldModel::setPlayerNumber( int i )
{
  iPlayerNumber = i;
  return true;
}

/*! This method returns the side of the agent. Note that
    the side of the agent does not change after half time.
    \return side (SIDE_LEFT or SIDE_RIGHT) for agent */
SideT WorldModel::getSide( ) const
{
  return sideSide;
}

/*! This method sets the side of the agent
    \param s (SIDE_LEFT or SIDE_RIGHT) for agent
    \return bool indicating whether update was succesful.*/
bool WorldModel::setSide( SideT s )
{
```

```
  sideSide = s;
  m_iMultX = (getSide() == SIDE_LEFT ) ?  1 : -1 ;    // set the draw info
  m_iMultY = (getSide() == SIDE_LEFT ) ? -1 :  1 ;    // from Logger.C
  return true;
}

/*! This method returns the teamname of the agent in this worldmodel
    \return teamname for the agent in this worldmodel */
const char* WorldModel::getTeamName( ) const
{
  return strTeamName ;
}

/*! This method sets the teamname of the agent. The maximum team name is
    MAX_TEAM_NAME_LENGTH as defined in Soccertypes.h.
    \param str teamname for the agent in this worldmodel
    \return bool indicating whether update was succesful.*/
bool WorldModel::setTeamName( char * str )
{
  strcpy( strTeamName, str );
  return true;
}

/*! This method returns the current playmode. This playmode is passed
    through by the referee.

    \return current playmode (see SoccerTypes for possibilities) */
PlayModeT WorldModel::getPlayMode( ) const
{
  return playMode ;
}

/*! This method sets the play mode of the agent in this worldmodel
    \param pm for the agent in this worldmodel
    \return bool indicating whether update was succesful.*/
bool WorldModel::setPlayMode( PlayModeT pm )
{
  playMode = pm;
  if( ( pm == PM_GOAL_KICK_LEFT  && getSide() == SIDE_LEFT  ) ||
      ( pm == PM_GOAL_KICK_RIGHT && getSide() == SIDE_RIGHT )    )
    setTimeLastCatch( getTimeLastSenseMessage() );
  return true;
}

/*! This method returns the goal difference. When this value is below zero,
    the team of agent is behind, 0 means that the score is currently the same
```

```
    for both teams and a value higher than zero means that you're winning!.
    \return goal difference */
int WorldModel::getGoalDiff( ) const
{
  return iGoalDiff;
}

/*!This method adds one goal to the goal difference. Call this method when your
   team has scored a goal
   \return new goal difference */
int WorldModel::addOneToGoalDiff( )
{
  return ++iGoalDiff;
}

/*!This method subtracts one from the goal difference. Call this method when
   you're team has conceided a goal
   \return new goal difference */
int WorldModel::subtractOneFromGoalDiff()
{
  return --iGoalDiff;
}

/*! This method returns the amount of commands c performed by the agent.
    This is supplied in the sense_body message.
    \param c CommandT of which number of commands should be returned.
    \return amount of commands c performed by the soccerserver. */
int WorldModel::getNrOfCommands( CommandT c ) const
{
  return iCommandCounters[ (int) c ];
}

/*! This method sets the number of commands c that were performed by the agent.
    This is supplied in the sense_body message and can be used to check
    whether an action is actually performed by the soccer server, since the
    corresponding counter should be one higher than the previous
    sense_body message. When this is the case the corresponding index of
    the PerformedCommands array is set to true.
    \param c CommandT of which the number of commands should be set.
    \param i number of commands that are performed of this command
    \return bool indicating whether update was performed. */
bool WorldModel::setNrOfCommands( CommandT c, int i )
{
  int iIndex = (int) c;

  // if counter is the same as before, no command is performed, otherwise it is
```

```
  performedCommands[iIndex] = ( iCommandCounters[iIndex] == i ) ? false : true;
  iCommandCounters [iIndex] = i;
  return true;
}

/*! This method returns the time the status of the ball was last checked
    (coach only).
    \return server cycle the status of the ball was last checked. */
Time WorldModel::getTimeCheckBall( ) const
{
  return timeCheckBall;
}

/*! This method sets the time the ball was checked for the last time (coach
    only).
    \param time server time ball was checked.
    \return bool indicating whether update was succesfull. */
bool WorldModel::setTimeCheckBall( Time time )
{
  timeCheckBall = time;
  return true;
}

/*! This method returns the status of the ball. This value is derived from
    the check_ball command that can only be used by the coach. The status of
    the ball corresponds to the server time returned by getTimeCheckBall.
    \return BallStatus status of the ball. */
BallStatusT WorldModel::getCheckBallStatus( ) const
{
  return bsCheckBall;
}

/*! This method sets the status of the ball. The status of the ball
    corresponds to the server time returned by getTimeCheckBall.  This method
    is only useful for the coach, since only he can sent a check_ball message.
    \return BallStatus status of the ball.  */
bool WorldModel::setCheckBallStatus( BallStatusT bs )
{
  bsCheckBall = bs;
  return true;
}

/*! This method returns a boolean indicating whether the synchronization method
    indicated we are ready. */
bool WorldModel::getRecvThink( )
{
```

```
  return m_bRecvThink;
}

/*! This method returns the string that we want communicate. It can be set
    during the creation of the action. When the ActHandler sends its actions,
    it sends this string to the server.
    \return communication string. */
char* WorldModel::getCommunicationString( )
{
  return m_strCommunicate;
}


/*! This methods returns the object type of the object that is currently
    listened to. This information is gathered from a sense message.
    When this agent says a message, we will definitely hear it. */
ObjectT WorldModel::getObjectFocus( )
{
  return m_objFocus;
}


/*! This methods sets the object type of the object that is currently listened
    to. This information is gathered from a sense message. When this agent says
    a message, we will definitely hear it. */
bool WorldModel::setObjectFocus( ObjectT obj )
{
  m_objFocus = obj;
  return true;
}


/*! This method sets the string that we want communicate. It can be set
    during the creation of the action. When the ActHandler sends its actions,
    it sends this string to the server.
    \return communication string. */
bool WorldModel::setCommunicationString( char *str )
{
  strncpy( m_strCommunicate, str, MAX_SAY_MSG );
  return true;
}


/*!This method starts an iteration over an object set g. This method will
   return the first ObjectT in an ObjectSetT that has a confidence higher than
   dConf. After this call use the method iterateObjectNext with the same
```

```
   argument to get the other objects that belong to this set.
   \param iIndex index that should be used in consecutive cycles.
   \param g ObjectSetT of which the ObjecT should be returned.
   \param dConf minimum confidence needed for ObjectT to be returned.
   \return ObjectT that is first in the set g. */
ObjectT WorldModel::iterateObjectStart(int& iIndex,ObjectSetT g,double dConf,
                                                                  bool bForward)
{
  iIndex = -1;
  return iterateObjectNext( iIndex, g, dConf, bForward );
}

/*!This method gets the next object in the iteration over an object set g.
   iterateObjectStart should have been called first. Only ObjectT with a
   confidence higher than dConf will be returned. When no more objects are
   available OBJECT_ILLEGAL is returned.
   \param iIndex same argument as was supplied to iterateObjectStart.
   \param g ObjectSetT of which the ObjecT should be returned.
   \param dConf minimum confidence needed for ObjectT to be returned.
   \return ObjectT that is next in the set g. */
ObjectT WorldModel::iterateObjectNext(int& iIndex,ObjectSetT g, double dConf,
                                                                  bool bForward)
{
  ObjectT o, objGoalie = OBJECT_TEAMMATE_1;
  bool bContinue = true;

  if( g == OBJECT_SET_TEAMMATES_NO_GOALIE )
    objGoalie = getOwnGoalieType();

  if( iIndex < 0 )
    iIndex = (bForward==false) ? OBJECT_MAX_OBJECTS : -1 ;

  // when dConf is not specified it has the default value of -1.0, in this
  // case set it to the confidence threshold defined in PlayerSetting, but
  // only do this for dynamic objexts, not for flags, lines, etc. since all
  // should be returend.
  if( dConf == -1.0 && (g==OBJECT_SET_OPPONENTS || g==OBJECT_SET_PLAYERS ||
                g==OBJECT_SET_TEAMMATES) )
    dConf = PS->getPlayerConfThr();

  int i = 0;
  if( bForward == true )
    i = iIndex + 1;
  else
    i = iIndex - 1;
  bContinue = ( bForward == false ) ? ( i >= 0 ) : ( i < OBJECT_MAX_OBJECTS );
```

```
  while( bContinue )
  {
    o = (ObjectT) i;
    if( SoccerTypes::isInSet( o, g, objGoalie ) )
    {
      if( getConfidence( o ) >= dConf )
      {
        iIndex = i;
        return o;
      }
      else if( dConf == 1.0 && getTimeLastSeeMessage() == getTimeLastSeen( o ))
      {
        iIndex = i; // confidence of 1.0 can only be in same cycle as see
        return o;   // message. Therefore first test should succeed normally;
      }             // in cases where this method is called after see message,
                    // but new sense has already arrived, confidence is lowered
    }               // but we want to return object that was seen in last see
                    // message; this compensates for those cases.
    if( bForward == true )
      i++;
    else
      i--;
    bContinue = ( bForward == false ) ? ( i >= 0 ) :( i < OBJECT_MAX_OBJECTS);
  }

  return OBJECT_ILLEGAL;
}

/*!This method finsished the iteration. It should be called after
   iterateObjectNext has returned OBJECT_ILLEGAL indicating no more objects are
   available that satisfy the constraints.
   \param iIndex index of iteration */
void WorldModel::iterateObjectDone( int &iIndex )
{
  iIndex = -1;
}

/*! This method returns the ObjectType of the agent. This
    is an ObjectT between OBJECT_TEAMMATE_1 and OBJECT_TEAMMATE_11
    \return ObjectT that represent the type of the agent  */
ObjectT WorldModel::getAgentObjectType( ) const
{
  return agentObject.getType();
}
```

```
int WorldModel::getAgentIndex( ) const
{
  return SoccerTypes::getIndex( getAgentObjectType() );
}

/*! This method sets the ObjectType of the agent. This
    is an objectT between OBJECT_TEAMMATE_1 and OBJECT_TEAMMATE_11
    \param ObjectT that represent the type of the agent
    \return bool indicating whether the update was succesful */
bool WorldModel::setAgentObjectType( ObjectT o )
{
  agentObject.setType( o );
  return true;
}

/*! This method returns the body angle relative to the neck of the agent.
    \return AngDeg representing the body angle relative to the neck */
AngDeg WorldModel::getAgentBodyAngleRelToNeck( ) const
{
  return agentObject.getBodyAngleRelToNeck();
}

/*! This method returns the global neck angle of the agent in the world.
    \return global neck angle agent */
AngDeg WorldModel::getAgentGlobalNeckAngle(  ) const
{
  return agentObject.getGlobalNeckAngle(  );
}

/*! This method returns the global body angle of the agent in the world.
    \return global body angle agent */
AngDeg WorldModel::getAgentGlobalBodyAngle(  )
{
  return agentObject.getGlobalBodyAngle(  );
}

/*! This method returns the stamina information of the agent
    \return Stamina stamina of the agent */
Stamina WorldModel::getAgentStamina( ) const
{
  return agentObject.getStamina();
}

/*! This method returns a TiredNessT value that indicates how tired the agent
   is.
                                    \return TiredNessT value that indicates how tired the agent is. */
```

```
TiredNessT WorldModel::getAgentTiredNess( ) const
{
  return getAgentStamina().getTiredNess(
                                                SS->getRecoverDecThr(), SS->getStaminaMax() );
}

/*! This method returns the effort information of the agent
    \return double effort of the agent */
double WorldModel::getAgentEffort( ) const
{
  return agentObject.getStamina().getEffort();
}

/*! This method returns the global velocity information of the agent
    \return global velocity of the agent */
VecPosition WorldModel::getAgentGlobalVelocity( ) const
{
  return agentObject.getGlobalVelocity();
}

/*! This method returns the speed of the agent
    \return speed of the agent */
double WorldModel::getAgentSpeed( ) const
{
  return agentObject.getSpeed();
}

/*! This method returns the global position of the agent
    \return global position of the agent */
VecPosition WorldModel::getAgentGlobalPosition( ) const
{
  return agentObject.getGlobalPosition();
}

/*! This method sets the view angle of the agent.
    \return bool indicating whether update was successful.  */
bool WorldModel::setAgentViewAngle( ViewAngleT va )
{
  agentObject.setViewAngle( va );
  return true;
}


/*! This method returns the view angle of the agent.

    \return ViewAngleT view angle of the agent (VA_NARROW, VA_NORMAL,
```

```
   VA_WIDE)*/
ViewAngleT WorldModel::getAgentViewAngle( ) const
{
 return agentObject.getViewAngle();
}

/*! This method sets the view quality of the agent.
   \return bool indicating whether update was successful.  */
bool WorldModel::setAgentViewQuality( ViewQualityT vq )
{
 agentObject.setViewQuality( vq );
 return true;
}
/*! This method returns the view quality of the agent
   \return ViewQualityT of the agent (VA_LOW, VA_HIGH). */
ViewQualityT WorldModel::getAgentViewQuality( ) const
{
 return agentObject.getViewQuality();
}

/*! This method returns the view frequency of see messages for the agent
   relative to the time of the sense_body interval. So 0.5 means a see
   message arrives twice in every simulation cycle.
   \return double representing the view frequency */
double WorldModel::getAgentViewFrequency( ViewAngleT va, ViewQualityT vq )
{
 double dViewQualityFactor ;
 double dViewWidthFactor   ;

 if( va == VA_ILLEGAL )
   va = getAgentViewAngle();
 if( vq == VQ_ILLEGAL )
   vq = getAgentViewQuality();

 switch( va )
 {
   case VA_NARROW:  dViewWidthFactor   = 0.5; break;
   case VA_NORMAL:  dViewWidthFactor   = 1.0; break;
   case VA_WIDE:    dViewWidthFactor   = 2.0; break;
   case VA_ILLEGAL:
   default:         dViewWidthFactor   = 0.0; break;
 }

 switch( vq )
 {
   case VQ_LOW:     dViewQualityFactor = 0.5; break;
```

```
    case VQ_HIGH:    dViewQualityFactor = 1.0; break;
    case VQ_ILLEGAL:
    default:         dViewQualityFactor = 0.0; break;
   }

  return dViewQualityFactor*dViewWidthFactor;
}

/*! This method returns whether the arm of the agent can be moved. */
bool WorldModel::getAgentArmMovable( )
{
  return agentObject.getArmMovable();
}

/*! This method returns the current position the arm of the agent is
    pointing towards. */
VecPosition WorldModel::getAgentArmPosition( )
{
  return agentObject.getGlobalArmPosition();
}

/*! This method returns how many cycles it will last before the arm
    of the agent stops pointing. */
int WorldModel::getAgentArmExpires( )
{
  return agentObject.getArmExpires();
}

/*! This method returns the global position of the ball. The method
    getConfidence with as argument OBJECT_BALL should be called to check the
    confidence of this global position.
    \return global position bal. */
VecPosition  WorldModel::getBallPos()
{
  return getGlobalPosition( OBJECT_BALL );
}

/*! This method returns the current estimate of the speed of the ball.
    \return speed of the ball (magnitude of the global velocity vector). */
double WorldModel::getBallSpeed()
{
  return Ball.getGlobalVelocity().getMagnitude();
}

/*! This method returns the global direction of the ball velocity.
    \return global direction of the ball velocity */
```

```
AngDeg WorldModel::getBallDirection()
{
 return Ball.getGlobalVelocity().getDirection();
}

/*! This method returns the time of the global position
    of the specified object.
    \param ObjectT that represent the type of the object to check
    \return time corresponding to the global position of 'o' */
Time WorldModel::getTimeGlobalPosition( ObjectT o )
{
 PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
 if( object != NULL )
   return object->getTimeGlobalPosition();
 return UnknownTime;
}


/*! This method returns the global position of an objectType. This
    method is normally used for the objects on the field (player,
    opponents and the ball).  When the global position cannot be
    determined, a VecPosition with both the x and y coordinate are set
    to 'UnknownDoubleValue'.

    \param ObjectT that represent the type of the object to check
    \return VecPosition containing the global position. */
VecPosition WorldModel::getGlobalPosition( ObjectT o )
{
 Object *object = getObjectPtrFromType( o );
 if( object != NULL )
 {
   if( SoccerTypes::isFlag( o ) || SoccerTypes::isGoal( o ) )
     return SoccerTypes::getGlobalPositionFlag( o, getSide(),
                                        SS->getGoalWidth() );
   else
     return object->getGlobalPosition();
 }
 return VecPosition( UnknownDoubleValue, UnknownDoubleValue);
}

/*! This method returns the time of the global velocity
    of the specified object.
    \param ObjectT that represent the type of the object to check
    \return time corresponding to the global velocity of 'o' */
Time WorldModel::getTimeGlobalVelocity( ObjectT o )
{
```

```
  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
  if( object != NULL )
    return object->getTimeGlobalVelocity();
  return UnknownTime;
}


/*! This method returns the global velocity of an objectType. When the
    global position cannot be determined, a VecPosition is returned with
    both the x and y coordinate set to 'UnknownDoubleValue'.
    \param ObjectT that represent the type of the object to check
    \return VecPosition containing the global position. */
VecPosition WorldModel::getGlobalVelocity( ObjectT o )
{
  DynamicObject *object = (DynamicObject*)getObjectPtrFromType( o );
  if( object != NULL )
    return object->getGlobalVelocity(  );
  return VecPosition( UnknownDoubleValue, UnknownDoubleValue );

}

/*! This method returns the relative distance between the agent and the object
    supplied as the first argument. No check is made whether this information
    is up to date (use isVisible or getConfidence for that).
    \param ObjectT that represent the type of the object to check
    \return relative distance to this object */
double WorldModel::getRelativeDistance( ObjectT o )
{
  Object *object = getObjectPtrFromType( o );
  if( object != NULL )
    return object->getRelativeDistance();
  return UnknownDoubleValue;
}

/*! This method returns the relative position of the object to the agent.
    No check is made whether this information is up to date (use isVisible or
    getConfidence for that).
    \param ObjectT that represent the type of the object to check
    \return relative position to this object */
VecPosition  WorldModel::getRelativePosition( ObjectT o )
{
  Object *object = getObjectPtrFromType( o );
  if( object != NULL )
    return object->getRelativePosition();
  return VecPosition(UnknownDoubleValue, UnknownDoubleValue);
}
```

```
/*! This method returns the relative angle between the agent and the object
    supplied as the first argument. No check is made whether this information
    is up to date (use isVisible or getConfidence for that).
    By default the returned angle is relative to the neck
    of the agent. When the second argument 'bWithBody' is set to true, the
    returned angle is relative to the body of the agent.
    \param ObjectT that represent the type of the object to check
    \param bWithBody when true angle is relative to body, otherwise to neck
          (default false)
    \return relative angle to this object */
AngDeg WorldModel::getRelativeAngle( ObjectT o, bool bWithBody )
{
  Object *object = getObjectPtrFromType( o );
  double dBody   = 0.0;

  if( object != NULL )
  {
    if( bWithBody == true )
      dBody = getAgentBodyAngleRelToNeck();
    return VecPosition::normalizeAngle( object->getRelativeAngle() - dBody );
  }
  return UnknownDoubleValue;
}

/*! This method returns the time of the global angles (both body and
    neck angle) of the specified object.

    \param ObjectT that represent the type of the object to check
    \return time corresponding to both the stored body and neck angle */
Time WorldModel::getTimeGlobalAngles( ObjectT o )
{
  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
  if( object != NULL )
    return object->getTimeGlobalAngles();
  return Time( -1, 0);
}

/*! This method returns the global body angle of the specified object.
    No check is made whether this information is up to date (use
    getTimeGlobalAngles).
    \param ObjectT that represent the type of the object to check
    \return last known global body angle of this object */
AngDeg WorldModel::getGlobalBodyAngle( ObjectT o )
{
  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
```

```
  if( object != NULL )
    return object->getGlobalBodyAngle();
  return UnknownAngleValue;
}

/*! This method returns the global neck angle of the specified object.
    No check is made whether this information is up to date (use
    getTimeGlobalAngles).
    \param ObjectT that represent the type of the object to check
    \return last known global neck angle of this object */
AngDeg WorldModel::getGlobalNeckAngle( ObjectT o )
{
  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
  if( object != NULL )
    return object->getGlobalNeckAngle();
  return UnknownAngleValue;
}

/*! This method returns the global angle of the specified object (this object
    is normally a line).
    \param ObjectT that represent the type of the object to check
    \return global angle of this object in the field */
AngDeg WorldModel::getGlobalAngle( ObjectT o )
{
  if( SoccerTypes::isLine( o ) )
    return SoccerTypes::getGlobalAngleLine( o, getSide() );
  return UnknownAngleValue;
}


/*! This method returns the confidence value of the object
    supplied as the first argument. The confidence is calculated using the
    current server cycle and the time the object was last seen.
    \param ObjectT that represent the type of the object to check
    \return confidence value [0.0, 1.0] that indicates the confidence value */
double WorldModel::getConfidence( ObjectT o)
{
  Object *object = getObjectPtrFromType( o );
  if( object != NULL )
    return object->getConfidence( getCurrentTime() );

  return 0.0;
}

/*! This method returns wheter the specified object type is a known player.
    A known player is a player of which we know for certain that the player
```

```
    number is correct. If a player is seen without a number and it cannot be
    mapped to a player, it is put on the first empty position in the
    player list and the status of known player is set to false.
    \param o object type of player that should be checked
    \return bool indicating whether we are certain of number of player 'o'. */
bool WorldModel::isKnownPlayer( ObjectT o )
{
  PlayerObject *object = (PlayerObject *)getObjectPtrFromType( o );
  if( object != NULL  )
    return object->getIsKnownPlayer();
  return false;

}

/*! This method returns the object type of the opponent
    goalkeeper. Which object type is the actual goalkeeper is checked
    in different ways. First of all this information is can be
    available in a see message.  When no player is stored in the world
    model of which this is is perceived, the opponent goalkeeper is
    assumed the player with the highest x coordinate, but only if this
    player stands very close in front of the goal

    \return ObjectT that represents the opponent goalkeeper, OBJECT_ILLEGAL if
    it cannot be determined which object type is the opponent goalkeeper. */
ObjectT WorldModel::getOppGoalieType()
{
  static ObjectT objGoalieType = OBJECT_ILLEGAL;

  if( objGoalieType != OBJECT_ILLEGAL &&
      isConfidenceGood( objGoalieType ) &&
      isKnownPlayer( objGoalieType ) )
    return objGoalieType;

  ObjectT objOppMaxX = OBJECT_ILLEGAL;
  double  x = -100.0, y = UnknownDoubleValue;

  for( int i = 0; i < MAX_OPPONENTS; i++ )
  {
    if( isConfidenceGood( Opponents[i].getType( ) ) )
    {
      if( Opponents[i].getIsGoalie() == true ) // &&
//          Opponents[i].getGlobalPosition().getX() > PENALTY_X - 2.0 )
      {
        objGoalieType = Opponents[i].getType();
        return Opponents[i].getType();
      }
```

```
      if( Opponents[i].getGlobalPosition().getX() > x )
      {
        x         = Opponents[i].getGlobalPosition().getX();
        y         = Opponents[i].getGlobalPosition().getY();
        objOppMaxX = Opponents[i].getType();
      }
    }
  }

  // if opponent with highest x is nr 1, assume it is goalkeeper when standing
  // in own penalty area, otherwise assume goalkeeper closest player to goal.
  if( (objOppMaxX == OBJECT_OPPONENT_1 && x > PENALTY_X + 4.0 ) ||
      (objOppMaxX != OBJECT_ILLEGAL    && x > PITCH_LENGTH/2.0 - 6.0 &&
       fabs( y ) < SS->getGoalWidth()/2.0 ))
    return objOppMaxX;
  return OBJECT_ILLEGAL;
}


/*! This method returns the object type of the own goalkeeper. Which object
    type is the actual goalkeeper is checked in different ways. First of all
    this information is available in the see, when (goalie) is behind the
    perceived object. When no player is stored in the world model of which
    this is perceived, the own goalkeeper is assumed the player with the lowest
    x coordinate, but only if this player stands close in front of the goal.
    \return ObjectT that represents the own goalkeeper, OBJECT_ILLEGAL if
    it cannot be determined which object type is the own goalkeeper. */
ObjectT WorldModel::getOwnGoalieType()
{
  ObjectT objOwnMinX = OBJECT_ILLEGAL;
  double x = 100.0, y = UnknownDoubleValue;
  for( int i = 0; i < MAX_TEAMMATES; i++ )
  {
    if( isConfidenceGood( Teammates[i].getType( ) ) )
    {
      if( Teammates[i].getIsGoalie() == true )
        return Teammates[i].getType();
      if( Teammates[i].getGlobalPosition().getX() < x )
      {
        x         = Teammates[i].getGlobalPosition().getX();
        y         = Teammates[i].getGlobalPosition().getY();
        objOwnMinX = Teammates[i].getType();
      }
    }
  }
  if( ( objOwnMinX == OBJECT_TEAMMATE_1 && x < - ( PENALTY_X + 4.0 ) ) ||
```

```
     (objOwnMinX != OBJECT_ILLEGAL    && x < -  PITCH_LENGTH/2.0 + 6.0  &&
      fabs( y ) < SS->getGoalWidth()/2.0 ))
    return objOwnMinX;
  return OBJECT_ILLEGAL;
}


/*! This method returns the last server cycle the specified object has been
    seen.
    \param object type of object that should be checked
    \return server time this object was last seen (in a see message). */
Time WorldModel::getTimeLastSeen( ObjectT o )
{
  Object *object = getObjectPtrFromType( o );
  if( object != NULL )
    return object->getTimeLastSeen(  );
  return Time( -1, 0);

}

/*! This method returns the last server cycle the relative distance change of
    the specified object has been reported.
    \param object type of object that should be checked
    \return server time relative distance change of this object was last
          seen (in a see message). */
Time WorldModel::getTimeChangeInformation( ObjectT o )
{
  DynamicObject *object = (DynamicObject*)getObjectPtrFromType( o );
  if( object != NULL )
    return object->getTimeChangeInformation(  );
  return Time( -1, 0);

}

/*! This method returns the last global position derived from a see message.
    The time corresponds to the method 'getTimeGlobalPositionLastSee'.
    \param object type of object that should be checked
    \return global position related to the last see message. */
VecPosition WorldModel::getGlobalPositionLastSee( ObjectT o )
{
  DynamicObject *object = (DynamicObject*)getObjectPtrFromType( o );
  if( object != NULL )
    return object->getGlobalPositionLastSee(  );
  return VecPosition( UnknownDoubleValue, UnknownDoubleValue);
}
```

```
/*! This method returns the last server cycle the global position
    the specified object has been calculated.
    \param object type of object that should be checked
    \return server time global position of this object was last
          derived (from a see message). */
Time WorldModel::getTimeGlobalPositionLastSee( ObjectT o )
{
  DynamicObject *object = (DynamicObject*)getObjectPtrFromType( o );
  if( object != NULL )
    return object->getTimeGlobalPosDerivedFromSee(  );
  return UnknownTime;
}

/*! This method returns the last global velocity derived from a see message.
    The time corresponds to the method 'getTimeChangeInformation'.
    \param object type of object that should be checked
    \return global velocity related to the last see message. */
VecPosition WorldModel::getGlobalVelocityLastSee( ObjectT o )
{
  DynamicObject *object = (DynamicObject*)getObjectPtrFromType( o );
  if( object != NULL )
    return object->getGlobalVelocityLastSee(  );
  return VecPosition( UnknownDoubleValue, UnknownDoubleValue);
}

/*! This method returns the last global body angle derived from a see message.
    The time corresponds to the method 'getTimeChangeInformation'.
    \param object type of object that should be checked
    \return global body angle related to the last see message. */
AngDeg WorldModel::getGlobalBodyAngleLastSee( ObjectT o )
{
  PlayerObject *object = (PlayerObject*)getObjectPtrFromType( o );
  if( object != NULL )
    return object->getGlobalBodyAngleLastSee(  );
  return UnknownAngleValue;
}

/*! This method returns the number of cycles it will take the tackle to expire.
    In case of the argument OBJECT_ILLEGAL, the number of cycles for the
    agentObject is returned. */
int WorldModel::getTackleExpires( ObjectT o )
{
  if( o == OBJECT_ILLEGAL || o == getAgentObjectType() )
    return agentObject.getTackleExpires();

  PlayerObject *object = (PlayerObject*)getObjectPtrFromType( o );
```

```
  if( object == NULL )
    return 0;
  return max(0, object->getTimeTackle( ) -
             getCurrentTime()        +
             SS->getTackleCycles());
}

/*! This method returns the arm direction of object 'o'. It does not keep track
    of how relevant this information is. When the pointing agent stops pointing
    the angle is set to UnknownAngleValue. */
AngDeg WorldModel::getGlobalArmDirection( ObjectT o )
{
  PlayerObject *object = (PlayerObject*)getObjectPtrFromType( o );
  if( object == NULL )
    return UnknownAngleValue;

  return object->getGlobalArm();
}

/*! This method returns the time related to the global arm direction of object
    'o'. When the pointing agent stops pointing the angle is set to
    UnknownAngleValue and the returned time is not relevant. */
Time  WorldModel::getTimeGlobalArmDirection( ObjectT o )
{
  PlayerObject *object = (PlayerObject*)getObjectPtrFromType( o );
  if( object == NULL )
    return Time(-1,0);

  return object->getTimeGlobalArm();
}

/*! This method returns the probability that a tackle performed by object o
    will succeed. This probability depends on the relative distance to the
    ball in both the x and y direction and various server parameters. In the
    case that o equals OBJECT_ILLEGAL, the returned probability corresponds
    to that of the agent object. */
double WorldModel::getProbTackleSucceeds( ObjectT o, int iExtraCycles,
                                          VecPosition *pos)
{
  if( o == OBJECT_ILLEGAL )
    o = getAgentObjectType();

  VecPosition posObject   = getGlobalPosition( o );
  VecPosition posBall     = (pos == NULL ) ? getBallPos() : *pos ;
  AngDeg      angBody     = getGlobalBodyAngle( o );
  int         iExtra      = 0;
```

```
double      dTackleDist, dDist=0;

// if opponent goalie is within 3 metres he can probably catch in next cycle
// RC2003 HACK
if( o == getOppGoalieType() &&
    posBall.getDistanceTo( o ) < 3.0  )
  return 1.0;

if( o != getAgentObjectType() )
{
  // get the number of cycles object was not seen and assume he moves 0.6
  // in every cycle. Only in case of bad body direction subtract one.
  // then move object position closer to the ball
  dDist  = posBall.getDistanceTo( posObject );
  iExtra = getCurrentTime() - getTimeLastSeen( o ) + iExtraCycles;
  AngDeg ang    = (posBall - posObject).getDirection();

  // if body angle ok,
  if( getCurrentTime() - getTimeGlobalAngles( o ) < 2 )
  {
    if( fabs( VecPosition::normalizeAngle( ang - angBody ) ) > 35 )
                                        iExtra --;
    if( getGlobalVelocity( o ).getMagnitude() < 0.2 )
                                        iExtra --;
  }

  double dExtra = 0.7*max( 0, iExtra );

  // if object was not seen in last see message, he stood further away
  // then the visible_distance.
  if( getTimeLastSeen( o ) != getTimeLastSeeMessage() &&
      getCurrentTime() == getTimeLastSeeMessage() &&
      dDist - dExtra < SS->getVisibleDistance() )
  {
    Log.log( 560, "prob tackle succeeds: opp not seen raise dExtra" );
    dExtra = dDist - SS->getVisibleDistance();
  }

  // now incorporate that to kick the ball we may need more cycles
  //    dExtra = max( 0, dExtra + iExtraCycles );

  // do not move object more than 4.0 metres.
  posObject += VecPosition( min(4.0,min(dDist - 0.2, dExtra )), ang, POLAR );

  // object is directed towards ball
  angBody = ang;
```

```
  }

  VecPosition posBallRel  = posBall - posObject;
  posBallRel.rotate( - angBody );
  if ( posBallRel.getX() > 0.0 )      // ball in front -> tackle_dist parameter
    dTackleDist = SS->getTackleDist();
  else
    dTackleDist = SS->getTackleBackDist();

  double dProb =
     pow(fabs(posBallRel.getX())/dTackleDist         ,SS->getTackleExponent())+
     pow(fabs(posBallRel.getY())/SS->getTackleWidth(),SS->getTackleExponent());

  Log.log( 556,
           "tackle relpos o %d: (%f,%f) dist %f body %f extra %d %d: prob %f",
           o, posBallRel.getX(),posBallRel.getY(), dDist, angBody, iExtra,
           iExtraCycles, max(0,1-dProb) );

  return max( 0, 1 - dProb );
}

/*! This method returns a list with all the opponents that are location within
   distance of 'dDist' of position 'pos'. */
list<ObjectT> WorldModel::getListCloseOpponents( VecPosition pos,double dDist )
{
  int iIndex;
  list<ObjectT> listOpp;

  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_OPPONENTS );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_OPPONENTS  ) )
  {
    if( getGlobalPosition( o ).getDistanceTo( pos ) < dDist )
      listOpp.push_back( o );
  }
  iterateObjectDone( iIndex );
  return listOpp;
}

/*! This method returns the tackle probability of the closest opponent to the
    ball. */
double WorldModel::getProbTackleClosestOpp( int iExtraCycles )
{
  ObjectT obj = getClosestInSetTo( OBJECT_SET_OPPONENTS, OBJECT_BALL );
  if( obj == OBJECT_ILLEGAL )
    return -1.0;
```

```
  return getProbTackleSucceeds( obj, iExtraCycles );
}

/*! This method sets the value of the specified object to a known
    player or not.  A known player is a player of which the exact team
    name and player number are known. If the player number is not
    known, information about the object is stored at an empty position
    in the player array and the value of isKnownPlayer is set to
    'false'.

    \param o object type of which known player information should be set
    \param isKnownPlayer new known player value
    \return boolean indicating whether update was successful */
bool WorldModel::setIsKnownPlayer( ObjectT o, bool isKnownPlayer )
{
  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
  return object->setIsKnownPlayer( isKnownPlayer );
}

/*! This method sets the time the object 'o' has last been seen.
    \param o object of which the time should be changed
    \param time new time for this object
    \return bool indicating whether update was successful. */
bool WorldModel::setTimeLastSeen( ObjectT o, Time time )
{
  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
  return object->setTimeLastSeen( time );
}

/*! This method sets the heterogeneous player type of the object that is
    passed as the first argument.
    \param o object type of which the heterogeneous player type should be set
    \param iPlayer new heterogeneous player type of this object. */
bool WorldModel::setHeteroPlayerType( ObjectT  o, int iPlayerType )
{
  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( o );
  if( o == getAgentObjectType() )
    updateSSToHeteroPlayerType( iPlayerType );

  return object->setHeteroPlayerType( iPlayerType );
}

/*! This method returns the player type of the object 'o' in the
  current formation. */
PlayerT WorldModel::getPlayerType ( ObjectT o )
```

```
{
 if( o == OBJECT_ILLEGAL )
   o = getAgentObjectType();
 return formations->getPlayerType( SoccerTypes::getIndex( o ) );
}

/*! This method returns whether the object 'o' is located in the set of
   player types 'ps'. */
bool WorldModel::isInPlayerSet( ObjectT o, PlayerSetT ps )
{
 return SoccerTypes::isPlayerTypeInSet( getPlayerType( o ), ps );
}


/*! This methods returns the heterogeneous player type of object 'obj'.
   Initially, the player types of all players are set to the type of the agent
   itself. After new information from the coach, they are set correctly. */
int WorldModel::getHeteroPlayerType( ObjectT obj )
{
 PlayerObject *object = (PlayerObject*) getObjectPtrFromType( obj );

 return object->getHeteroPlayerType( );

}


/*! This method returns the global position of the opponent goal.
   \return VecPosition containing the position of the opponent goal. */
VecPosition WorldModel::getPosOpponentGoal( )
{
 ObjectT objGoal = SoccerTypes::getGoalOpponent( getSide() );
 if( isPenaltyUs() || isPenaltyThem() )
   objGoal = ( getSidePenalty() == SIDE_LEFT ) ?OBJECT_GOAL_L:OBJECT_GOAL_R ;
 return SoccerTypes::getGlobalPositionFlag(
                    objGoal,
                    getSide( ),
                    SS->getGoalWidth() );
}

/*! This method returns the global position of the own goal.
   \return VecPosition containing the position of the own goal. */
VecPosition WorldModel::getPosOwnGoal( )
{
 SideT sideGoal = getSide();
 ObjectT objGoal = SoccerTypes::getOwnGoal( sideGoal );
 if( isPenaltyUs() || isPenaltyThem() )
```

```
    objGoal = (getSidePenalty() == SIDE_LEFT ) ? OBJECT_GOAL_L : OBJECT_GOAL_R;

  return SoccerTypes::getGlobalPositionFlag(
                    objGoal,
                    getSide( ),
                    SS->getGoalWidth() );
}

/*! This method returns the relative distance to the opponent goal
    \return relative distance from the agent to the opponent goal. */
double  WorldModel::getRelDistanceOpponentGoal()
{
  VecPosition posGoal = getPosOpponentGoal();
  return getAgentGlobalPosition().getDistanceTo( posGoal );

}

/*! This method returns the relative angle to the opponent goal. This
    relative angle is the relative angle between the opponent goal
    position and the agent position. The neck and body angle of the
    agent are NOT taken into account.

    \return relative angle between goal and agent position. */
double  WorldModel::getRelAngleOpponentGoal()
{
  VecPosition posGoal;
  if( sideSide == SIDE_LEFT )
    posGoal = SoccerTypes::getGlobalPositionFlag( OBJECT_GOAL_R, sideSide );
  else
    posGoal = SoccerTypes::getGlobalPositionFlag( OBJECT_GOAL_L, sideSide );

  return ( posGoal - getAgentGlobalPosition()).getDirection() ;

}

/*! This method returns information about the heterogeneous player at index
    'iIndex'. This information consists of a subset of the ServerSettings
    values that fully specify the information of the heterogeneous player. */
HeteroPlayerSettings WorldModel::getInfoHeteroPlayer( int iIndex )
{
  return pt[iIndex];
}

/*! This method returns the heterogeneous player information of the object
    'obj'. This information consists of a subset of the ServerSettings
    values that fully specify the information of the heterogeneous player. */
```

```
HeteroPlayerSettings WorldModel::getHeteroInfoPlayer( ObjectT obj )
{
//  if( obj == getAgentObjectType() || SoccerTypes::isOpponent( obj ) )
//    return getInfoHeteroPlayer( agentObject.getHeteroPlayerType() );
  if( ! SoccerTypes::isKnownPlayer( obj ) )
    obj = getAgentObjectType();

  PlayerObject *object = (PlayerObject*) getObjectPtrFromType( obj );
  int          iType   = object->getHeteroPlayerType();
  // default iType in Object = 0
  return getInfoHeteroPlayer( iType );
}


/*! This method inserts a substituted opponent player in the set of substituted
    opponent players. This can then be detected by the coach, who then can
    figure out to which new type this opponent player is changed. */
bool WorldModel::setSubstitutedOpp( ObjectT obj )
{
  m_setSubstitutedOpp.insert( obj );
  return true;
}

/*! This method returns the first substituted opponent player in the set of
    substituted opponent players and then erases this opponent from the set. */
ObjectT WorldModel::getSubstitutedOpp( )
{
  if( m_setSubstitutedOpp.empty() == true )
    return OBJECT_ILLEGAL;

  ObjectT obj = *m_setSubstitutedOpp.begin();
  m_setSubstitutedOpp.erase( obj );
  return obj;
}


/*! This method returns the dash power rate of the object 'obj'. */
double WorldModel::getDashPowerRate( ObjectT obj )
{
  return getHeteroInfoPlayer( obj ).dDashPowerRate ;
}

/*! This method returns the maximum speed of the object 'obj'. */
double WorldModel::getPlayerSpeedMax( ObjectT obj )
{
  return getHeteroInfoPlayer( obj ).dPlayerSpeedMax ;
```

```
}

/*! This method returns the decay of the object 'obj'. */
double WorldModel::getPlayerDecay( ObjectT obj )
{
 return getHeteroInfoPlayer( obj ).dPlayerDecay ;
}

/*! This method returns the maximal kick distance of the object 'obj'. */
double WorldModel::getMaximalKickDist( ObjectT obj )
{
 return getHeteroInfoPlayer( obj ).dMaximalKickDist ;
}

/*! This method returns the stamina increase of the object 'obj'. */
double WorldModel::getStaminaIncMax( ObjectT obj )
{
 return getHeteroInfoPlayer( obj ).dStaminaIncMax ;
}

/*! This method returns the size of the object 'obj'. */
double WorldModel::getPlayerSize( ObjectT obj )
{
 return getHeteroInfoPlayer( obj ).dPlayerSize ;
}

/*! This method returns the inertia moment of the object 'obj'. */
double WorldModel::getInertiaMoment( ObjectT obj )
{
 return getHeteroInfoPlayer( obj ).dInertiaMoment;
}

/*! This method returns the maximum effort of the object 'obj'. */
double WorldModel::getEffortMax( ObjectT obj )
{
 return getHeteroInfoPlayer( obj ).dEffortMax;
}

/*! This method returns the effective max speed of the object 'obj'. */
double WorldModel::getEffectiveMaxSpeed( ObjectT obj, bool bWithNoise )
{
 double dSpeed = 0.0;
 HeteroPlayerSettings pt = getHeteroInfoPlayer( obj );

 for( int j = 0; j < 15 ; j ++ )
 {
```

```
    dSpeed *= pt.dPlayerDecay;
    dSpeed += SS->getMaxPower()*pt.dEffortMax*pt.dDashPowerRate;
    if( dSpeed > pt.dPlayerSpeedMax )
      dSpeed = pt.dPlayerSpeedMax;
    if( bWithNoise )
      dSpeed += sqrt(2*(pow(dSpeed * SS->getPlayerRand(), 2)));
  }
  return dSpeed;
}

/*! This method checks whether a queued action is performed. The commands in
    QueuedCommands are the commands that are sent to the server by the
    ActHandler. The performedCommands array contains the commands that are
    performed in the last cycle (using the count information in the sense_body
    message). Using these two array it is possible to check whether a command
    is actually performed by the server. When there is an action that is sent
    to the server and not performed, this method returns false, true otherwise.
    \return true when all commands sent to the server are performed. */
bool WorldModel::isQueuedActionPerformed()
{
  // for all possible commands check if it is sent in previous cycle,
  // but not performed
  for( int i = 0 ; i < CMD_MAX_COMMANDS ; i++ )
    if( queuedCommands[i].time   == getTimeLastSenseMessage() - 1 &&
        performedCommands[i]     == false )
      return false;

  return true;
}

/*! This method checks whether the play mode indicates that we have
    a free kick. When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether we have a free kick. */
bool WorldModel::isFreeKickUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  bool bLeftKick = (pm==PM_FREE_KICK_LEFT  ||
pm==PM_INDIRECT_FREE_KICK_LEFT );
  bool bRightKick= (pm==PM_FREE_KICK_RIGHT ||
pm==PM_INDIRECT_FREE_KICK_RIGHT);
```

```
  return ( bLeftKick   && getSide() == SIDE_LEFT  ) ||
         ( bRightKick  && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the play mode indicates that the other
    team has a free kick. When the specified PlayModeT equals
    PM_ILLEGAL (default), the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether the other team has a free kick. */
bool WorldModel::isFreeKickThem( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  bool bLeftKick = (pm==PM_FREE_KICK_LEFT  ||
pm==PM_INDIRECT_FREE_KICK_LEFT );
  bool bRightKick= (pm==PM_FREE_KICK_RIGHT ||
pm==PM_INDIRECT_FREE_KICK_RIGHT);

  return ( bRightKick  && getSide() == SIDE_LEFT  ) ||
         ( bLeftKick   && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the play mode indicates that there is (or
    will be) a before kick off situation. This is the case when the play mode
    equals PM_BEFORE_KICK_OFF or either PM_GOAL_LEFT or PM_GOAL_RIGHT since
    after the goal the play mode will go to PM_BEFORE_KICK_OFF.
    When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether there is a before kick off situation. */
bool WorldModel::isBeforeKickOff( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return pm == PM_BEFORE_KICK_OFF  || pm == PM_GOAL_LEFT  ||
         pm == PM_GOAL_RIGHT || isKickOffUs( pm ) || isKickOffThem( pm );
}

/*! This method checks whether the play mode indicates that there is
    a dead ball situation and our team is allowed to perform the action.
    That is our team has either a free kick, kick in, corner kick or a kick in.
    When the specified PlayModeT equals PM_ILLEGAL (default), the
```

```
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether we have a dead ball situation. */
bool WorldModel::isDeadBallUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return isKickInUs  ( pm ) || isFreeKickUs  ( pm ) || isCornerKickUs     ( pm)
      || isKickOffUs ( pm ) || isOffsideThem ( pm ) || isFreeKickFaultThem( pm)
      || isGoalKickUs( pm ) || isBackPassThem( pm ) ;
}


/*! This method checks whether the current play mode indicates that there is
    a dead ball situation and their team is allowed to perform the action.
    That is their team has either a free kick, kick in, corner kick or a kick
    in. When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether they have a dead ball situation. */
bool WorldModel::isDeadBallThem( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return isFreeKickThem( pm ) || isKickInThem  ( pm ) || isCornerKickThem ( pm)
      ||  isKickOffThem ( pm ) || isGoalKickThem( pm ) || isFreeKickFaultUs( pm)
      || isOffsideUs    ( pm ) || isBackPassUs  ( pm ) ;
}

bool WorldModel::setChangeViewCommand( SoccerCommand soc )
{
  m_changeViewCommand = soc;
  return true;
}

SoccerCommand WorldModel::getChangeViewCommand( )
{
  return m_changeViewCommand;
}

/*! This method returns the side the penalty kick is taken. */
SideT WorldModel::getSidePenalty( )
```

```
{
  return m_sidePenalty;
}

/*! This method sets the side the penalty kick is taken. */
bool WorldModel::setSidePenalty( SideT side )
{
  m_sidePenalty = side;
  return true;
}


/*! This method checks whether the current play mode indicates that we have
    a corner kick. When the specified PlayModeT equals PM_ILLEGAL (default),
    the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether we have a corner kick. */
bool WorldModel::isCornerKickUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_CORNER_KICK_LEFT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_CORNER_KICK_RIGHT && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that the other
    team has a corner kick. When the specified PlayModeT equals PM_ILLEGAL
    (default), the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether the other team has a corner kick. */
bool WorldModel::isCornerKickThem( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_CORNER_KICK_RIGHT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_CORNER_KICK_LEFT   && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that we stood
    offside. When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
```

```
        mode is used.
    \return bool indicating whether we stood offside. */
bool WorldModel::isOffsideUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_OFFSIDE_RIGHT  && getSide() == SIDE_RIGHT ) ||
         ( pm == PM_OFFSIDE_LEFT   && getSide() == SIDE_LEFT );
}

/*! This method checks whether the current play mode indicates that the other
    team stood offside. When the specified PlayModeT equals PM_ILLEGAL
    (default), the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
        mode is used.
    \return bool indicating whether the other team stood offside. */
bool WorldModel::isOffsideThem( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_OFFSIDE_LEFT  && getSide() == SIDE_RIGHT ) ||
         ( pm == PM_OFFSIDE_RIGHT && getSide() == SIDE_LEFT );
}

/*! This method checks whether the current play mode indicates that we have
    a kick in. When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
        mode is used.
    \return bool indicating whether we have a kick in. */
bool WorldModel::isKickInUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_KICK_IN_LEFT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_KICK_IN_RIGHT && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that the other
    team has a kick in. When the specified PlayModeT equals PM_ILLEGAL
    (default), the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
        mode is used.
```

```
   \return bool indicating whether the other team has a kick in. */
bool WorldModel::isKickInThem( PlayModeT pm )
{
 if( pm == PM_ILLEGAL )
   pm = getPlayMode();

 return ( pm == PM_KICK_IN_RIGHT  && getSide() == SIDE_LEFT  ) ||
        ( pm == PM_KICK_IN_LEFT   && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that we have
   made a free kick fault. This happens when a player kicks the ball twice
                                          after a free kick or a kick in.
                                          When the specified PlayModeT equals PM_ILLEGAL
(default), the
   current play mode is used.
   \param pm play mode to check. In default case (PM_ILLEGAL) the current play
         mode is used.
   \return bool indicating whether we have made a free kick fault. */
bool WorldModel::isFreeKickFaultUs( PlayModeT pm )
{
 if( pm == PM_ILLEGAL )
   pm = getPlayMode();

 return ( pm == PM_FREE_KICK_FAULT_LEFT  && getSide() == SIDE_LEFT  ) ||
        ( pm == PM_FREE_KICK_FAULT_RIGHT && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that the other
   team has made a free kick fault. This happens when a player kicks the ball
                                          twice after a free kick or a kick in.
                                          When the specified PlayModeT equals PM_ILLEGAL
(default), the
   current play mode is used.
   \param pm play mode to check. In default case (PM_ILLEGAL) the current play
         mode is used.

   \return bool indicating whether the other team has made a free
   kick fault.*/
bool WorldModel::isFreeKickFaultThem( PlayModeT pm )
{
 if( pm == PM_ILLEGAL )
   pm = getPlayMode();

 return ( pm == PM_FREE_KICK_FAULT_RIGHT  && getSide() == SIDE_LEFT  ) ||
        ( pm == PM_FREE_KICK_FAULT_LEFT   && getSide() == SIDE_RIGHT ) ;
```

```
}

/*! This method checks whether the current play mode indicates that we have
    a kick off. When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether we have a kick off. */
bool WorldModel::isKickOffUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_KICK_OFF_LEFT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_KICK_OFF_RIGHT && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that the other
    team has a kick off. When the specified PlayModeT equals PM_ILLEGAL
    (default), the current play mode is used. When the specified PlayModeT
    equals PM_ILLEGAL (default), the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether the other team has a kick off. */
bool WorldModel::isKickOffThem( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_KICK_OFF_RIGHT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_KICK_OFF_LEFT   && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that we have
    made a back pass (which is not allowed). This occurs when a teamamte has
                                          passed the ball back to the goalkeeper and he catched it.
                                          When the specified PlayModeT equals PM_ILLEGAL
(default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether we have made a back pass. */
bool WorldModel::isBackPassUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();
```

```
  return ( pm == PM_BACK_PASS_LEFT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_BACK_PASS_RIGHT && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that
    the other team has made a back pass (which is not allowed). This
    occurs when an opponent has passed the ball back to the goalkeeper
    and he catched it.  When the specified PlayModeT equals PM_ILLEGAL
    (default), the current play mode is used.

    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether the other team has made a back pass. */
bool WorldModel::isBackPassThem( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_BACK_PASS_RIGHT && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_BACK_PASS_LEFT  && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that we have
    a goal kick. When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether we have a goal kick. */
bool WorldModel::isGoalKickUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_GOAL_KICK_LEFT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_GOAL_KICK_RIGHT && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that the other
    team has a kick off. When the specified PlayModeT equals PM_ILLEGAL
    (default), the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether the other team has a kick off. */
bool WorldModel::isGoalKickThem( PlayModeT pm )
{
```

```
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( pm == PM_GOAL_KICK_RIGHT  && getSide() == SIDE_LEFT  ) ||
         ( pm == PM_GOAL_KICK_LEFT   && getSide() == SIDE_RIGHT ) ;
}

/*! This method checks whether the current play mode indicates that we have
    a penalty. When the specified PlayModeT equals PM_ILLEGAL (default), the
    current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether we have a penalty. */
bool WorldModel::isPenaltyUs( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( (
            ( pm == PM_PENALTY_SETUP_LEFT ||
              pm == PM_PENALTY_READY_LEFT ||
              pm == PM_PENALTY_TAKEN_LEFT ) && getSide() == SIDE_LEFT  )   ||
           (
            ( pm == PM_PENALTY_SETUP_RIGHT ||
              pm == PM_PENALTY_READY_RIGHT ||
              pm == PM_PENALTY_TAKEN_RIGHT ) && getSide() == SIDE_RIGHT  ) );
}


/*! This method checks whether the current play mode indicates that the other
    team takes a penalty. When the specified PlayModeT equals PM_ILLEGAL
    (default), the current play mode is used.
    \param pm play mode to check. In default case (PM_ILLEGAL) the current play
          mode is used.
    \return bool indicating whether the other team has a penalty. */
bool WorldModel::isPenaltyThem( PlayModeT pm )
{
  if( pm == PM_ILLEGAL )
    pm = getPlayMode();

  return ( (
            ( pm == PM_PENALTY_SETUP_LEFT ||
              pm == PM_PENALTY_READY_LEFT ||
              pm == PM_PENALTY_TAKEN_LEFT ) && getSide() == SIDE_RIGHT  )   ||
           (
            ( pm == PM_PENALTY_SETUP_RIGHT ||
```

```
            pm == PM_PENALTY_READY_RIGHT ||
            pm == PM_PENALTY_TAKEN_RIGHT ) && getSide() == SIDE_LEFT  ) );
}

bool WorldModel::isFullStateOn( SideT s )
{
 if( s == SIDE_ILLEGAL )
   s = getSide();

 if( s == SIDE_LEFT )
   return SS->getFullStateLeft();
 else if( s == SIDE_RIGHT )
   return SS->getFullStateRight();
 else
   return false;
}

/*! This method prints all the objects and information of the agent to the
   specified outputstream. Only the information of the objects that are seen
   recently are printed.
   \param os output stream to which output is written (default cout). */
void WorldModel::show( ostream & os )
{
 int i;
 os << "Worldmodel (" << getCurrentTime() << ")\n" <<
       "=======================\n";
 os << "Teamname: " << getTeamName() << "\n";
 if( Ball.getTimeLastSeen( ).getTime() != -1 )
   Ball.show();
 os << "Teammates: " << "\n";
 for( i = 0; i < MAX_TEAMMATES ; i++ )
   if( isConfidenceGood( Teammates[i].getType() ) )
     Teammates[i].show( getTeamName() );
 os << "Opponents: " << "\n";
 for( i = 0; i < MAX_OPPONENTS ; i++ )
   if( isConfidenceGood( Opponents[i].getType() ) )
     Opponents[i].show( DEFAULT_OPPONENT_NAME );
 os << "Agent: " << "\n";
 agentObject.show( getTeamName() );

 os << "General Info: " << "\n" <<
       "side: "      << SoccerTypes::getSideStr( getSide() )    << "\n" <<
       "kicks: "     << getNrOfCommands( CMD_KICK )            << "\n" <<
       "turns: "     << getNrOfCommands( CMD_TURN )             << "\n" <<
       "dashes: "    << getNrOfCommands( CMD_DASH )              << "\n" <<
       "turnnecks: " << getNrOfCommands( CMD_TURNNECK )        << "\n" <<
```

```
        "says: "      << getNrOfCommands( CMD_SAY )             << "\n" <<
        "playmode: "  << SoccerTypes::getPlayModeStr( playMode ) << "\n" <<
        "=================================="                    << "\n";

}

/*! This method prints all the objects and information contained in the object
    set 'set' to specified outputstream.
    \param os output stream to which output is written (default cout). */
void WorldModel::show( ObjectSetT set, ostream & os )
{
 int iIndex;
 for( ObjectT o = iterateObjectStart( iIndex, set );
      o != OBJECT_ILLEGAL;
      o = iterateObjectNext( iIndex, set ) )
   show( o, os );
 os << "\n";
}

/*! This method prints the queued commands - commands that are sent by the
    ActHandler to the server - to the specified output stream.
    \param os output stream to which information is printed (default cout)*/
void WorldModel::showQueuedCommands( ostream & os )
{
 os << "Commands in queue:\n" ;
 for( int i = 0; i < CMD_MAX_COMMANDS; i++ )
   if( queuedCommands[i].commandType != CMD_ILLEGAL )
     queuedCommands[i].show( os );
}

/*! This method prints the information about the Object o to the output stream
    os.
    \param o object of which information should be printed
    \param os output stream to which information is printed. */
void WorldModel::show( ObjectT o, ostream &os )
{
 Object *object = getObjectPtrFromType( o );
 if( object != NULL )
 {
   if( SoccerTypes::isPlayer( o ) )
   {
     PlayerObject *pobj = (PlayerObject*) object ;
     if( SoccerTypes::isTeammate( o ) )
       pobj->show( getTeamName(), os );
     else
       pobj->show( "Opponent", os );
```

```
    }
    else
      object->show( os );
  }
}

/*! This method blocks till new information has arrived. Information is either
    a sense_body message or a see message. If there isn't received information
    from the server for longer than 3 seconds, server is assumed dead and
    false is returned.
    \return true when new info has arrived, false if server is dead */
bool WorldModel::waitForNewInformation( )
{
  bool bReturn = true;
  if( bNewInfo == false ) // there hasn't arrived any information yet
  {
#ifdef WIN32
    DWORD waittime = PS->getServerTimeOut() * 1000;
    EnterCriticalSection( &mutex_newInfo );
    int ret;
    Log.logWithTime( 2, "go into conditional wait" );
    while( (ret = WaitForSingleObject( event_newInfo,
                                waittime) ) == WAIT_ABANDONED )
      printf("(WorldModel::waitForNewInformation) failure in loop!!\n");
    Log.logWithTime( 2, "go out of conditional wait" );
    if( ret == WAIT_TIMEOUT ) // if no info was received but timer timed out
      bReturn = false;
    ResetEvent( event_newInfo );
    LeaveCriticalSection( &mutex_newInfo );
#else
    struct timeval now;
    struct timespec timeout;
    gettimeofday(&now, NULL);
    timeout.tv_sec = now.tv_sec + PS->getServerTimeOut();
    timeout.tv_nsec = now.tv_usec*1000;

    // lock mutex and wait till it is unlocked by Sense thread
    // this happens in setTimeLastSeeMessage, setTimeLastSenseMessage
    // or setTimeLastSeeGlobalMessage
    pthread_mutex_lock( &mutex_newInfo );
    int ret;
    Log.logWithTime( 2, "go into conditional wait" );
    while( (ret = pthread_cond_timedwait( &cond_newInfo,
                &mutex_newInfo, &timeout) ) == EINTR )
     printf("(WorldModel::waitForNewInformation) failure in loop!!\n");
    Log.logWithTime( 2, "go out of conditional wait" );
```

```
    if( ret == ETIMEDOUT ) // if no info was received but timer timed out
      bReturn = false;
    pthread_mutex_unlock( &mutex_newInfo );
#endif
  }
  else
    Log.logWithTime( 2, "already new info waiting" );

  // update the time of the see and the sense messages not yet with the time
  // from the last received messages. This is done to circumvent that a time
  // is updated while the main thread is still determining a new action and
  // the last see message is not yet updated in the world model. This is
  // therefore then in updateAll;

  // reset the indication of new visual information
  bNewInfo = false;

  return bReturn;
}

/*! This method logs all object information that is currently stored in the
    World Model. The output is formatted as follows. First the current time
    (cycle_nr,cycle_stopped) is printed, followed by the object information
    of the specified object 'obj'. The global x and global y position are first
    printed, followed by the global x and global y velocity. Finally the body
    and the neck angle are printed. Then the information of the ball
    "pos_x pos_y vel_x vel_y conf" is printed, followed by the information of
    all eleven teammates "nr pos_x pos_y vel_x vel_y conf" and the same
    information about the eleven opponents. Values that are currently not known
    are replaced by the value -10.0000. This method is normally used by a
    player to log every cycle all its information contained in the world model
    to a file. If the coach (with perfect information) does the same, these two
    files can be analyzed to calculate the error for the different values.
    \param iLogLevel loglevel for which information should be printed
    \param obj object of which information should be printed at start line
              of this object also the body and neck angle are printed. */
void WorldModel::logObjectInformation( int iLogLevel, ObjectT obj )
{

  static char   str[2048];
  double dConf     = PS->getPlayerConfThr();
  sprintf( str, "(%4d,%3d) ", getCurrentTime().getTime(),
                                                    getCurrentTime().getTimeStopped() );
  if( obj != OBJECT_ILLEGAL )
    sprintf( str, "%12.6f %12.6f %12.6f %12.6f %12.6f %12.6f",
                          getGlobalPosition(obj).getX(),
```

```
                        getGlobalPosition(obj).getY(),
                        getGlobalVelocity(obj).getX(),
                        getGlobalVelocity(obj).getY(),
                        getGlobalBodyAngle(obj),
                        getGlobalNeckAngle(obj) );

if( getConfidence      ( OBJECT_BALL ) > dConf &&
   getRelativeDistance( OBJECT_BALL ) < 20.0 )
 sprintf( &str[strlen(str)], " %12.6f %12.6f",
                        getGlobalPosition(OBJECT_BALL).getX(),
                        getGlobalPosition(OBJECT_BALL).getY() );
else
 sprintf( &str[strlen(str)], " %12.6f %12.6f", -10.0, -10.0 );

if( getTimeGlobalVelocity( OBJECT_BALL ) > getTimeFromConfidence( dConf ) &&
   getRelativeDistance  ( OBJECT_BALL ) < 20.0 )
 sprintf( &str[strlen(str)], " %12.6f %12.6f",
                        getGlobalVelocity(OBJECT_BALL).getX(),
                        getGlobalVelocity(OBJECT_BALL).getY() );
else
 sprintf( &str[strlen(str)], " %12.6f %12.6f", -10.0, -10.0 );
sprintf( &str[strlen(str)], " %12.6f", getConfidence(OBJECT_BALL) );

int    iIndex=-1;
int    iIndexPlayer;
for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_PLAYERS, 0.0 );
    o != OBJECT_ILLEGAL;
    o = iterateObjectNext( iIndex, OBJECT_SET_PLAYERS, 0.0 ) )
{
                                    bool bPrint = false;

 iIndexPlayer = (SoccerTypes::isTeammate(o))
              ? SoccerTypes::getIndex(o) + 1
              : SoccerTypes::getIndex(o) + 12;
 sprintf( &str[strlen(str)], " %d", iIndexPlayer );
 if( getConfidence      ( o ) > dConf && isKnownPlayer(o) &&
    getRelativeDistance( o ) < 20.0 )
                                        {
   sprintf( &str[strlen(str)], " %12.6f %12.6f",
                        getGlobalPosition(o).getX(),
                        getGlobalPosition(o).getY() );
   bPrint = true;
                                        }
 else
    sprintf( &str[strlen(str)], " %12.6f %12.6f", -10.0, -10.0 );
```

```
    if( getTimeGlobalVelocity( o ) > getTimeFromConfidence( dConf ) &&
        getRelativeDistance  ( o ) < 20.0 && isKnownPlayer(o)  )
                                              {
      sprintf( &str[strlen(str)], " %12.6f %12.6f",
                                getGlobalVelocity(o).getX(),
                                getGlobalVelocity(o).getY() );
      bPrint = true;
                                              }
    else
      sprintf( &str[strlen(str)], " %12.6f %12.6f", -10.0, -10.0 );

    if( bPrint )
      sprintf( &str[strlen(str)], " %12.6f", getConfidence(o) );
    else
      sprintf( &str[strlen(str)], " %12.6f", -10.0 );
  }
  if( getCurrentCycle() != 3000  )
      Log.log( iLogLevel, str );
}

/*! This method logs drawing information to the log file LogDraw. The drawing
    information is written in the syntax that is understood by the soccer
    monitor. The contents of the created file can afterwards be read by an
    (adapted) logplayer to show the logged information. The information logged
    in this method is all the global position information of the players on the
    field.
    \param iLogLevel loglevel that has to be passed to the Logger.
*/
void WorldModel::logDrawInfo( int iLogLevel  )
{
  if( getCurrentCycle() % ( SS->getHalfTime() * SS->getSimulatorStep()) == 0 )
    return ;

  bool   bReturn1 = false, bReturn2 = false;
  double dConfThr = PS->getPlayerConfThr();
  int    iIndex;
  char   strMsg1[MAX_MSG], strMsg2[MAX_MSG];
  char   strColorLeft[]  = "ffff00" ;
  char   strColorRight[] = "0000ff" ;
  if( getSide() != SIDE_LEFT )
  {
    sprintf( strColorLeft, strColorRight );
    sprintf( strColorRight, "ffff00" );
  }
```

```
  sprintf( strMsg1, "_2D_ CIRCLE col=%s ", strColorLeft ) ;
  // first each players logs the positions of all the teammates
  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_TEAMMATES, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_TEAMMATES, dConfThr ) )
  {
    bReturn1 = true;

    // add all the global positions to the string
    sprintf( &strMsg1[strlen(strMsg1)], "(%1.2f,%1.2f,1.3) ",
     m_iMultX*getGlobalPosition( o ).getX(),
     m_iMultY*getGlobalPosition( o ).getY() );
  }
  iterateObjectDone( iIndex );

  if( bReturn1 == false )
    sprintf( strMsg1, "_2D_ " );
  else
    strcat( strMsg1, "; " );

  sprintf( strMsg2, "CIRCLE col=%s ", strColorRight ) ;
  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_OPPONENTS, dConfThr);
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_OPPONENTS, dConfThr ) )
  {
    bReturn2 = true;

    // add all the global positions to the string
    sprintf( &strMsg2[strlen(strMsg2)], "(%1.2f,%1.2f,1.3) ",
     m_iMultX*getGlobalPosition( o ).getX(),
     m_iMultY*getGlobalPosition( o ).getY() );
  }
  iterateObjectDone( iIndex );
  strcat( strMsg2, ";" );

  if( bReturn2 == true )
    strcat( strMsg1, strMsg2 );
  if( bReturn1 == true || bReturn2 == true)
    LogDraw.log( iLogLevel, strMsg1 );

  return ;
}

/*! This method logs drawing information to the log file LogDraw. The drawing
    information is written in the syntax that is understood by the soccer
    monitor. The contents of the created file can afterwards be read by an
```

```
   (adapted) logplayer to show the logged information. This method logs all
   the information about the coordination graphs.
   \param iLogLevel loglevel that has to be passed to the Logger.
*/
void WorldModel::logCoordInfo( int iLogLevel  )
{
 if( getCurrentCycle() % ( SS->getHalfTime() * SS->getSimulatorStep()) == 0 )
   return ;

 bool   bReturn = false;
 int    iIndex;
 double dConfThr = PS->getPlayerConfThr();
 char   strMsg[MAX_MSG];

 map<double, ObjectT > mapDistToBall;
 VecPosition posBall = getGlobalPosition( OBJECT_BALL ), posFirst;
 int i = 0;

 for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_TEAMMATES, dConfThr);
     o != OBJECT_ILLEGAL;
     o = iterateObjectNext ( iIndex, OBJECT_SET_TEAMMATES, dConfThr ) )
 {
   // add all the global positions to the string
   mapDistToBall.insert(
     make_pair( getGlobalPosition( o ).getDistanceTo( posBall ), o ) );
 }
 iterateObjectDone( iIndex );

 sprintf( strMsg, "all_2D_ ");
 for(  map<double, ObjectT >::iterator iter = mapDistToBall.begin();
       iter != mapDistToBall.end()  ;
       iter ++ )
 {
  if( i == 0 )
    posFirst = getGlobalPosition( iter->second );
  else if( iter->second == getAgentObjectType() && i < 4 &&
           iter->first < 20.0 )
  {
    bReturn = true;
    sprintf( &strMsg[strlen(strMsg)],
                  "LINE col=%s (%1.2f,%1.2f,%1.2f,%1.2f);",
      m_colorPlayers[ SoccerTypes::getIndex( getAgentObjectType() ) ],
      m_iMultX*posFirst.getX(),
      m_iMultY*posFirst.getY(),
      m_iMultX*getGlobalPosition( iter->second ).getX(),
      m_iMultY*getGlobalPosition( iter->second ).getY() );
```

```
    }

    i++;
  }

  if( bReturn == true )
    LogDraw.log( iLogLevel, strMsg );
}


/* This method logs a circle to the specified log level.
   \param iLogLevel log level to which the circle should be printed
   \param pos center position of the circle
   \param dRadius radius of the circle
   \param bAll indication whether it should be printed to specific drawlevel
         or not (default false). */
bool WorldModel::logCircle( int iLogLevel, VecPosition pos, double  dRadius,
                            bool  bAll                              )
{
  if( getCurrentCycle() % ( SS->getHalfTime() * SS->getSimulatorStep()) == 0 )
    return false;

  return LogDraw.log( iLogLevel, "%s_2D_ CIRCLE col=%s (%1.2f,%1.2f,%f) ;",
          ( bAll == true ) ? "all" : "" ,
          m_colorPlayers[ getAgentIndex() ],
          m_iMultX*pos.getX(), m_iMultY*pos.getY(), dRadius );
}

/* This method logs a line to the specified log level.
   \param iLogLevel log level to which the circle should be printed
   \param pos1 first point of line
   \param pos2 second point of line
   \param bAll indication whether it should be printed to specific drawlevel
         or not (default false). */
bool WorldModel::logLine( int  iLogLevel, VecPosition pos1, VecPosition pos2,
                          bool bAll              )
{
  if( getCurrentCycle() % ( SS->getHalfTime() * SS->getSimulatorStep()) == 0 )
    return false;

  return LogDraw.log( iLogLevel,
                      "%s_2D_ LINE col=%s (%1.2f,%1.2f,%1.2f,%1.2f) ;",
                      ( bAll == true ) ? "all" : "" ,
                      m_colorPlayers[ getAgentIndex() ],
                      m_iMultX*pos1.getX(), m_iMultY*pos1.getY(),
                      m_iMultX*pos2.getX(), m_iMultY*pos2.getY() );
```

```
}

/* This method logs the ball position and its velocity.

   \param iLogLevel log level to which the ball information should be
   printed.*/
bool WorldModel::logDrawBallInfo( int  iLogLevel )
{
  int  iPoints = 20;
  char strMsg[MAX_MSG];
  char *strColor = "ff0000";

  VecPosition vel     = getGlobalVelocity( OBJECT_BALL );
  VecPosition pos     = getBallPos(), pos1, pos2;
  AngDeg      angOrth = VecPosition::normalizeAngle( vel.getDirection( )+90 );
  VecPosition posBall = getBallPos();
  sprintf( strMsg, "_2D_ CIRCLE col=%s (%1.2f,%1.2f,0.3) ; ",
     strColor, m_iMultX*posBall.getX(), m_iMultY*posBall.getY() );

  for( int i = 0; i < iPoints; i ++ )
  {
    // create all the orthogonal lines
    pos += vel;
    pos1 = pos + VecPosition( -0.5*vel.getMagnitude(), angOrth, POLAR );
    pos2 = pos + VecPosition(  0.5*vel.getMagnitude(), angOrth, POLAR );
    sprintf( &strMsg[strlen(strMsg)], "LINE col=%s (%1.2f,%1.2f,%1.2f,%1.2f);",
           strColor, m_iMultX*pos1.getX(), m_iMultY*pos1.getY(),
           m_iMultX*pos2.getX(), m_iMultY*pos2.getY() );
    vel *= SS->getBallDecay();
  }

  // finally create the line through all orthogonal lines
  sprintf( &strMsg[strlen(strMsg)], "LINE col=%s (%1.2f,%1.2f,%1.2f,%1.2f) ;",
         strColor, m_iMultX*getBallPos().getX(), m_iMultY*getBallPos().getY(),
         m_iMultX*pos.getX(), m_iMultY*pos.getY() );

  return LogDraw.log( iLogLevel, strMsg );
}

/*! This method checks whether the feature of type 'type' is relevant. This is
    done by comparing the time of the feature with the current time. When it
    is equal, the feature is assumed relevant, otherwise irrelevant.
    \param type feature that should be checked on relevance
    \return bool indicating whether the feature is relevant. */
bool WorldModel::isFeatureRelevant( FeatureT type )
```

```
{
  int  iIndex  = (int)type;
  bool bReturn = true;
  static Time  timeKickedUsed = -1;

#if 0
  Log.log( 460, "check feature (%d,%d,%d) relevance now (%d,%d,%d)",
         m_features[iIndex].getTimeSee().getTime(),
         m_features[iIndex].getTimeSense().getTime(),
         m_features[iIndex].getTimeHear().getTime(),
         getTimeLastSeeMessage().getTime(),
         getTimeLastSenseMessage().getTime(),
         getTimeLastHearMessage().getTime() );
#endif
  // feature is relevant when see and hear time is larger or equal than the
  // current time
  bReturn  = m_features[iIndex].getTimeSee()  >= getTimeLastSeeMessage() &&
             m_features[iIndex].getTimeHear() >= getTimeLastHearMessage();

  // if ball kicked, also recheck
  if( timeKickedUsed != getTimeLastSenseMessage() )
  {
    bReturn &= ( m_bPerformedKick == false );
    timeKickedUsed = getTimeLastSenseMessage();
  }

  // in case of interception, also recheck after sense message
  if( type ==  FEATURE_INTERCEPTION_POINT_BALL ||
      type ==  FEATURE_INTERCEPT_CLOSE        )
    bReturn &= m_features[iIndex].getTimeSense() >= getTimeLastSenseMessage();
  return bReturn;
}

/*! This method return the feature corresponding to the type 'type'.
    \param type type of this feature, e.g., FEATURE_FASTEST_PLAYER_TO_BALL.
    \return corresponding feature.*/
Feature WorldModel::getFeature( FeatureT type )
{
  return m_features[(int)type];
}

/*! This method updates the feature 'type' with the information stored in
    'feature'.
    \param type type of this feature, e.g., FEATURE_FASTEST_PLAYER_TO_BALL.
    \param feature information regarding this feature
    \return boolean indicating whether the update was successful.*/
```

```
bool WorldModel::setFeature( FeatureT type, Feature feature )
{
  m_features[(int)type] = feature;
  return true;
}

/*! This method draws a coordination graph between the relevant players
    and logs this information. */
void WorldModel::drawCoordinationGraph( )
{
  bool partOfGraph = false; // is the player part of the coordination graph?
  static bool bFirst = true;
  static ofstream ofile;
  char strTmp[128];
  if( bFirst == true )
  {
    sprintf( strTmp, "logs/coord%d.txt", getPlayerNumber() );
    ofile.open( strTmp );
    bFirst = false;
  }

  ObjectT objAgent = getAgentObjectType();
  ObjectT objClosestToBall = getFastestInSetTo( OBJECT_SET_TEAMMATES,
                                      OBJECT_BALL );
  ObjectT objTeam = OBJECT_ILLEGAL;

  VecPosition vpAgent = getAgentGlobalPosition();
  VecPosition vpTeam = getGlobalPosition( objTeam ); // -1000.0, 1000.0

#if 0
  if ( objClosestToBall == objAgent ) // Agent is closest to ball
    // find the closest player that has a x-coordinate greater than the agent's
    {
      partOfGraph = true;

      int iIndex;
      ObjectSetT set = OBJECT_SET_TEAMMATES;
      for( ObjectT o = iterateObjectStart( iIndex, set );
           o != OBJECT_ILLEGAL;
           o = iterateObjectNext ( iIndex, set ) )
      {
        VecPosition vpNew = getGlobalPosition( o );
        Log.log( 701, "i am fastest, check %d, last_x %f, his_x %f  \
                       last_rel %f, his_rel  %f",
                 o, vpTeam.getX(), vpNew.getX(),
                 getRelativeDistance( objTeam ),
```

```
            getRelativeDistance( o ) );
      if( objTeam == OBJECT_ILLEGAL)
      {
       if( vpNew.getX() > vpAgent.getX() - 5)
       {
        objTeam = o;
        vpTeam = vpNew;
       }
      }
      else if ( vpNew.getX() > vpTeam.getX() && o != objAgent &&
            vpNew.getX() > vpAgent.getX() - 5 )
      {
       if ( getRelativeDistance( o ) < getRelativeDistance( objTeam ) )
       {
        objTeam = o;
        vpTeam = vpNew;
       }
      }
     }
     iterateObjectDone( iIndex );
    }
   else // Agent is not closest to ball
   {
   }
#else // Jelle
   objAgent = getAgentObjectType();

   objClosestToBall = getFastestInSetTo( OBJECT_SET_TEAMMATES, OBJECT_BALL );
   VecPosition posFastest   = getGlobalPosition( objClosestToBall );
   objTeam = OBJECT_ILLEGAL;

   vpAgent = getAgentGlobalPosition();
   vpTeam  = getGlobalPosition( objTeam ); // -1000.0, 1000.0

   // serach for closest teammate with x higher than 5 of guy with ball
   int iIndex;
   ObjectSetT set = OBJECT_SET_TEAMMATES;
   for( ObjectT o = iterateObjectStart( iIndex, set );
                                        o != OBJECT_ILLEGAL;
                                        o = iterateObjectNext ( iIndex, set ) )
   {
    VecPosition vpNew = getGlobalPosition( o );
    Log.log( 701, "fastest %d, check %d, last_x %f, his_x %f dist_old %f dist_new %f last_rel
%f, his_rel  %f",
          objClosestToBall, o, vpTeam.getX(), vpNew.getX(),
          posFastest.getDistanceTo( vpTeam ),
```

```
            posFastest.getDistanceTo( vpNew ),
            getRelativeDistance( objTeam ),
            getRelativeDistance( o ) );
    if( objTeam == OBJECT_ILLEGAL)
    {
      if( vpNew.getX() > posFastest.getX() - 5 && o != objClosestToBall )
        {
          objTeam = o;
          vpTeam = vpNew;
        }
    }
    else if ( /*vpNew.getX() > vpTeam.getX() && */o != objClosestToBall &&
             vpNew.getX() > posFastest.getX() - 5)
    {
      if( posFastest.getDistanceTo( vpNew ) <
          posFastest.getDistanceTo( vpTeam ) )
      {
        objTeam = o;
        vpTeam = vpNew;
      }
    }
  }
  iterateObjectDone( iIndex );
  Log.log( 701, "team %d fastest %d me %d", objTeam, objClosestToBall, objAgent  );
  if( objAgent == objTeam || objAgent == objClosestToBall )
    partOfGraph = true;
#endif
  if( partOfGraph == true && getCurrentTime().isStopped() == false )
  {
    Log.log( 701,
             "create line with me (%d) (%f,%f) and closest team %d (%f,%f)",
             SoccerTypes::getIndex(getAgentObjectType()) +1,
             posFastest.getX(),posFastest.getY(),
             SoccerTypes::getIndex(objTeam) +1,
             vpTeam.getX(),vpTeam.getY()          );
    logLine( 701, posFastest, vpTeam, true );
    ofile << getCurrentCycle() << ": " <<
      SoccerTypes::getObjectStr( strTmp, objTeam, getTeamName() ) << " "
        << SoccerTypes::getIndex( objTeam ) <<
      endl;
  }
}
```

```
#include<list>          // needed for list<double>
#include<stdio.h>       // needed for printf
#include "WorldModel.h"

/*! This method returns the number of objects that are within the circle 'c'
    Only objects are taken into account that are part of the set 'set' and
    have a confidence higher than the threshold defined in PlayerSettings.
    \param c circle in which objects should be located to be counted
    \return number of objects from 'set' in circle 'c'*/
int WorldModel::getNrInSetInCircle( ObjectSetT set, Circle c )
{
  double dConfThr = PS->getPlayerConfThr();
  int    iNr      = 0;
  int    iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, dConfThr ) )
  {
    if( c.isInside( getGlobalPosition( o ) ) )
      iNr++;
  }
  iterateObjectDone( iIndex );

  return iNr;
}

/*! This method returns the number of visible objects that are part of
    the object set 'set' and located in the rectangle 'rect'. When no
    rectangle is defined (rect=NULL) the whole field is taken into
    account. Only objects with a confidence value higher than the
    threshold defined in PlayerSettings are taken into consideration.

    \param set ObjectSetT from which objects are taken into consideration
    \param rect Rectangle in which objects are counted (default NULL)
    \return number of objects in Rectangle 'rect'.*/
int WorldModel::getNrInSetInRectangle( ObjectSetT set, Rect *rect  )
{
  double dConfThr = PS->getPlayerConfThr();
  int    iNr      = 0;
  int    iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, dConfThr ) )
  {
```

```
    if( rect == NULL || rect->isInside( getGlobalPosition( o ) ) )
      iNr++;
  }
  iterateObjectDone( iIndex );
  return iNr;
}

/*! This method returns the number of objects in the specified cone.
    A cone is like a piece of a pie, in which 'start' is
    the center of the pie, 'end' is the edge of the pie and 'dWidth' is the
    half width of the piece after distance 1. Only objects are taken into
    consideration that are within the set 'set' and have a confidence higher
    than the threshold defined in PlayerSettings.
    \param set ObjectSetT of which objects are taken into consideration
    \param dWidth half width of the cone after distance 1.0
    \param start center of the cone
    \param end position that is the end of the cone.
    \return number of objects part of 'set' and located in this cone. */
int WorldModel::getNrInSetInCone( ObjectSetT set, double dWidth,
                                  VecPosition start , VecPosition end )
{
  double     dConfThr   = PS->getPlayerConfThr();
  int        iNr        = 0;
  int        iIndex;
  Line       line       = Line::makeLineFromTwoPoints( start, end );
  VecPosition posOnLine;
  VecPosition posObj;

  for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, dConfThr ) )
  {
    posObj    = getGlobalPosition( o );
    posOnLine = line.getPointOnLineClosestTo( posObj );
    // whether posOnLine lies in cone is checked by three constraints
    // - does it lie in triangle (to infinity)
    // - lies between start and end (and thus not behind me)
    // - does it lie in circle
    if(posOnLine.getDistanceTo(posObj) < dWidth*posOnLine.getDistanceTo(start)
       && line.isInBetween( posOnLine, start, end )
       && start.getDistanceTo( posObj ) < start.getDistanceTo( end ) )
        iNr++;
  }
  iterateObjectDone( iIndex );
  return iNr;
}
```

```
/*! This method returns whether the space in direction 'ang" of
    object 'obj' is occupied by any opponents. */
bool WorldModel::isEmptySpace( ObjectT obj, AngDeg ang, double dDist )
{
  if( obj == OBJECT_ILLEGAL )
    return false;

  VecPosition pos = getGlobalPosition( obj );
   pos += VecPosition( dDist, ang, POLAR );

  if( getNrInSetInCircle( OBJECT_SET_OPPONENTS, Circle( pos, dDist ) ) == 0 )
    return true;

  return false;
}


bool WorldModel::coordinateWith( ObjectT obj )
{
  VecPosition pos = getGlobalPosition( obj );
  if( pos.getDistanceTo( getBallPos() ) < 30.0 &&
      pos.getX() > getBallPos().getX() - 5.0 )
  {

    if( getFastestInSetTo( OBJECT_SET_TEAMMATES, OBJECT_BALL ) ==
        getAgentObjectType() )
    {
      logCircle( 700, pos, 2.5 );
    }
    Log.log( 700, "coordinate with %d %f %f (%f %f)",
             obj, pos.getDistanceTo( getBallPos() ),
             pos.getX(), getBallPos().getX(), getBallPos().getY() );
    return true;
  }

  return false;
}

/*! This method returns the object type of the closest object to the
    ObjectT that is supplied as the second argument. Only objects are
    taken into account that are part of the set 'set' and have a
    confidence higher than the supplied threshold. If no threshold is
    supplied, the threshold defined in PlayerSettings is used.
```

```
    \param set ObjectSetT which denotes objects taken into consideration

    \param objTarget ObjectT that represent the type of the object to
    compare to

    \param dDist will be filled with the closest distance

    \param dConfThr minimum confidence threshold for the objects in 'set'
    \return ObjectType that is closest to o */
ObjectT WorldModel::getClosestInSetTo( ObjectSetT set, ObjectT objTarget,
                                       double *dDist, double dConfThr )
{
  if( dConfThr == -1.0 ) dConfThr      = PS->getPlayerConfThr();
  ObjectT     closestObject = OBJECT_ILLEGAL;
  double      dMinMag       = 1000.0;
  VecPosition v;
  int         iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, dConfThr ) )
  {
    if( o != objTarget )              // do not include target object
    {
      v = getGlobalPosition( objTarget ) - getGlobalPosition( o );
      if( v.getMagnitude() < dMinMag )
      {
        dMinMag       = v.getMagnitude();
        closestObject = o;
      }
    }
  }

  iterateObjectDone( iIndex );
  if( dDist != NULL )
    *dDist = dMinMag;
  return closestObject;
}

/*! This method returns the ojbect type of the closest object to the specified
    position and that is part of the object set 'set' with a confidence
    higher than the
    supplied threshold. If no threshold is supplied, the threshold defined in
    PlayerSettings is used.
    \param set ObjectSetT which denotes objects taken into consideration
```

```
    \param pos position to which player should be compared

    \param dDist will be filled with the distance between pos and
    closest object

    \param dConfThr minimum confidence threshold for the objects in 'set'
    \return ObjectT representing object that is closest to pos */
ObjectT WorldModel::getClosestInSetTo( ObjectSetT set, VecPosition pos,
                                       double *dDist,  double dConfThr )
{
  ObjectT     closestObject   = OBJECT_ILLEGAL;
  double      dMinMag         = 1000.0;
  VecPosition v;
  int         iIndex;

  if( dConfThr == -1.0 ) dConfThr      = PS->getPlayerConfThr();
  for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, dConfThr ) )
  {
    v = pos - getGlobalPosition( o );
    if( v.getMagnitude() < dMinMag )
    {
      dMinMag         = v.getMagnitude();
      closestObject   = o;
    }
  }
  iterateObjectDone( iIndex );
  if( dDist != NULL )
    *dDist = dMinMag;
  return closestObject;
}

/*! This method returns the closest object in 'set' to the line
    'l'. The projection p of the global position of this object on the
    line 'l' should lie between pos1 and pos2. After the method is
    finished, it returns this object and the last two arguments of
    this method are set to the the distance between the object and p
    and the distance from pos1 to p respectively.

    \param set ObjectSetT which denotes objects taken into consideration
    \param l line to which opponents should be projected
    \param pos1 minimum allowed projection point
    \param pos2 maximum allowed projection point
    \param dDistObjToLine will contain distance from opponent to line l
    \param dDistPos1ToPoint will contain distance from pos1 to projection point
```

```
   opponent on line l
   \return object type of closest object to line l */
ObjectT WorldModel::getClosestInSetTo( ObjectSetT set, Line l,
                        VecPosition pos1, VecPosition pos2,
                        double *dDistObjToLine, double *dDistPos1ToPoint)
{
 VecPosition posObj;
 VecPosition posOnLine;
 double      dConfThr  = PS->getPlayerConfThr();
 ObjectT     obj       = OBJECT_ILLEGAL;
 double      dDist     ;
 double      dMinDist  = 1000.0;
 double      dDistPos1 = 1000.0;
 int         iIndex;

 for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
      o != OBJECT_ILLEGAL;
      o = iterateObjectNext ( iIndex, set, dConfThr ) )
 {
  posObj    = getGlobalPosition( o );
  posOnLine = l.getPointOnLineClosestTo( posObj );
  dDist     = posObj.getDistanceTo( posOnLine );
  if( l.isInBetween( posOnLine, pos1, pos2 ) && dDist < dMinDist )
  {
   dMinDist  = dDist;
   obj       = o;
   dDistPos1 = pos1.getDistanceTo( posOnLine );
  }
 }
 iterateObjectDone( iIndex );
 if( dDistObjToLine != NULL )
  *dDistObjToLine = dMinDist;
 if( dDistPos1ToPoint != NULL )
  *dDistPos1ToPoint = dDistPos1;

 return obj;
}

/*! This method returns the object type of the closest object relative to
   the agent. Only objects are taken into account that are part of the set
   'set'.
   \param set ObjectSetT which denotes objects taken into consideration
   \param dDist will be filled with the closest relative distance
   \return ObjectType that is closest to the agent*/
ObjectT WorldModel::getClosestRelativeInSet( ObjectSetT set, double *dDist )
{
```

```
  ObjectT     closestObject = OBJECT_ILLEGAL;
  double      dMinMag       = 1000.0;
  int         iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, set, 1.0 );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, 1.0 ) )
  {
    if( getRelativeDistance( o ) < dMinMag )
    {
      dMinMag       = getRelativeDistance( o );
      closestObject = o;
    }
  }

  iterateObjectDone( iIndex );
  if( dDist != NULL )
    *dDist = dMinMag;
  return closestObject;
}

/*! This method returns the object type of the second closest object to the
    object type that is supplied as the second argument. Only objects are taken
    into account within set 'set' and with a confidence higher than the
    supplied threshold. If no threshold is supplied, the threshold defined in
    PlayerSettings is used.
    \param set ObjectSetT which denotes objects taken into consideration
    \param obj ObjectT that represent the type of the object to check
    \param dDist will be filled with the distance to this player.
    \param dConfThr minimum confidence threshold for the objects in 'set'
    \return ObjectType that is second closest to obj */
ObjectT WorldModel::getSecondClosestInSetTo ( ObjectSetT set, ObjectT obj,
                                              double *dDist,  double dConfThr )
{
  VecPosition v;
  ObjectT     closestObject       = OBJECT_ILLEGAL;
  ObjectT     secondClosestObject = OBJECT_ILLEGAL;
  double      dMinMag             = 1000.0;
  double      dSecondMinMag       = 1000.0;
  int         iIndex;

  if( dConfThr == -1.0 ) dConfThr = PS->getPlayerConfThr();

  for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, dConfThr ) )
```

```
  {
   if( o != obj )
   {
     v = getGlobalPosition( obj ) - getGlobalPosition( o );
     if( v.getMagnitude() < dMinMag )                // closer then first
     {
       dSecondMinMag         = dMinMag;              // put first to second
       secondClosestObject   = closestObject;
       dMinMag               = v.getMagnitude();     // and this to first
       closestObject         = o;
     }
     else if( v.getMagnitude() < dSecondMinMag )     // between 1st and 2nd
     {
       dSecondMinMag         = v.getMagnitude();     // put this to second
       secondClosestObject   = o;
     }
   }
  }
  iterateObjectDone( iIndex );
  if( dDist != NULL )
    *dDist = dSecondMinMag;
  return secondClosestObject;
}

/*! This method returns the object type of the second closest object relative
    to the agent. Only objects are taken into account within set 'set' and
    which where seen in the last see message.
    \param set ObjectSetT which denotes objects taken into consideration
    \param dDist will be filled with the distance to this this object
    \return ObjectType that is second closest to the agent */
ObjectT WorldModel::getSecondClosestRelativeInSet( ObjectSetT set,
                                        double *dDist )
{
  ObjectT    closestObject      = OBJECT_ILLEGAL;
  ObjectT    secondClosestObject = OBJECT_ILLEGAL;
  double     dMinMag            = 1000.0;
  double     dSecondMinMag      = 1000.0;
  double     d;
  int        iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, set, 1.0 );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, 1.0 ) )
  {
    d = getRelativeDistance( o );
    if( d < dMinMag )                          // closer then first
```

```
    {
      dSecondMinMag         = dMinMag;              // put first to second
      secondClosestObject   = closestObject;
      dMinMag               = d;                    // and this to first
      closestObject         = o;
    }
    else if( d < dSecondMinMag )                    // between first and 2nd
    {
      dSecondMinMag         = d;                    // put this to second
      secondClosestObject   = o;
    }
  }
  iterateObjectDone( iIndex );
  if( dDist != NULL )
    *dDist = dSecondMinMag;
  return secondClosestObject;
}


/*! This method returns the object type of the furthest object to the
    ObjectT that is supplied as the second argument. Only objects are
    taken into account that are part of the set 'set' and have a
    confidence higher than the supplied threshold. If no threshold is
    supplied, the threshold defined in PlayerSettings is used.

    \param set ObjectSetT which denotes objects taken into consideration
    \param o ObjectT that represent the type of the object to compare to
    \param dDist will be filled with the furthest distance
    \param dConfThr minimum confidence threshold for the objects in 'set'
    \return ObjectType that is furthest to o */
ObjectT WorldModel::getFurthestInSetTo( ObjectSetT set, ObjectT objTarget,
                                        double *dDist, double dConfThr )
{
  if( dConfThr == -1.0 ) dConfThr      = PS->getPlayerConfThr();

  ObjectT     furthestObject = OBJECT_ILLEGAL;
  double      dMaxMag        = -1000.0;
  VecPosition v;
  int         iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, dConfThr ) )
  {
    if( o != objTarget )
    {
```

```
      v = getGlobalPosition( objTarget ) - getGlobalPosition( o );
      if( v.getMagnitude() > dMaxMag )
      {
       dMaxMag        = v.getMagnitude();
       furthestObject = o;
      }
     }
    }
    iterateObjectDone( iIndex );
    if( dDist != NULL )
     *dDist = dMaxMag;
    return furthestObject;
}

/*! This method returns the type of the object that is located furthest
    relative to the agent. Only objects are taken into account
    that are part of the set 'set'.
    \param set ObjectSetT which denotes objects taken into consideration
    \param dDist will be filled with the furthest relative distance
    \return ObjectType that is furthest to the agent */
ObjectT WorldModel::getFurthestRelativeInSet( ObjectSetT set, double *dDist  )
{
    ObjectT     furthestObject = OBJECT_ILLEGAL;
    double      dMaxMag       = -1000.0;
    int         iIndex;

    for( ObjectT o = iterateObjectStart( iIndex, set, 1.0 );
         o != OBJECT_ILLEGAL;
         o = iterateObjectNext ( iIndex, set, 1.0 ) )
    {
     if( getRelativeDistance( o ) > dMaxMag )
     {
      dMaxMag        = getRelativeDistance( o );
      furthestObject = o;
     }
    }
    iterateObjectDone( iIndex );
    if( dDist != NULL )
     *dDist = dMaxMag;
    return furthestObject;
}


VecPosition WorldModel::getPosClosestOpponentTo( double *dDist, ObjectT o )
{
    if( o == OBJECT_ILLEGAL )
```

```
    o = getAgentObjectType();
  ObjectT objOpp = getClosestInSetTo( OBJECT_SET_OPPONENTS, o, dDist );
  if( objOpp == OBJECT_ILLEGAL )
    return VecPosition( UnknownDoubleValue, UnknownDoubleValue );

  return getGlobalPosition( objOpp );
}

double WorldModel::getMaxTraveledDistance( ObjectT o )
{
  return (getCurrentTime() - getTimeLastSeen( o ) )*SS->getPlayerSpeedMax();
}


void WorldModel::createInterceptFeatures( )
{
  static int count = 0;
  static Time timeLastCalled(0,0);

  if( timeLastCalled == getTimeLastSenseMessage() )
    count++;
  else
    count = 0;

  if( count > 4 )
    cerr << getPlayerNumber() << " called createIntercept too often: " <<
       count << endl;
  // we check all possible next positions of the ball and see
  // whether a player (opponent or teammate) can reach the ball at that point
  // if so, we log this as a feature. We finish when all features have been
  // found.
  ObjectSetT      set = OBJECT_SET_PLAYERS;
  int             iCycles       = -1;
  int             iMinCyclesTeam    = 100;
  int             iMinCyclesOpp     = 100;
  bool            bOnlyMe           = false;

  VecPosition     posObj;
  int             iIndex;
  int             iCyclesToObj  ;

  // no feature available, calculate information
  ObjectT         objFastestTeam            = OBJECT_ILLEGAL;
  ObjectT         objFastestTeamNoGoalie    = OBJECT_ILLEGAL;
  ObjectT         objFastestOpp             = OBJECT_ILLEGAL;
  ObjectT         objFastestPlayer          = OBJECT_ILLEGAL;
```

```
int          iCyclesFastestPlayer      = -1;
int          iCyclesFastestTeam        = -1;
int          iCyclesFastestTeamNoGoalie = -1;
int          iCyclesFastestOpp         = -1;
int          iCyclesFastestMe          = -1;

bool         bFinishedPlayer           = false;
bool         bFinishedTeammates        = false;
bool         bFinishedTeammatesNoGoalie = false;
bool         bFinishedOpponents        = false;
bool         bFinishedMe               = false;
bool         bFinished                 = false;

ObjectT      objLog                    = OBJECT_ILLEGAL;
int          iCyclesLog                = -1;
FeatureT     featLog                   = FEATURE_ILLEGAL;

// for each next position of the ball
while( bFinished == false && iCycles <= PS->getPlayerWhenToIntercept() )
{
 iCycles++;
 iMinCyclesTeam = 100;
 iMinCyclesOpp  = 100;
 Log.log( 460, "fastest loop: %d", iCycles );

 // determine its position and traverse all players to check the teammate
 // and opponent who can reach it first
 posObj     = predictPosAfterNrCycles( OBJECT_BALL, iCycles );
 for( ObjectT o = iterateObjectStart( iIndex, set );
   o != OBJECT_ILLEGAL;
   o = iterateObjectNext ( iIndex, set ) )
 {
  if( getGlobalPosition(o).getDistanceTo(posObj)/SS->getPlayerSpeedMax()
      < iCycles + 1 && (bOnlyMe == false || SoccerTypes::isOpponent( o )
      || o == getAgentObjectType() ) )
  {
   Log.log( 460, "call predictNrCyclesToPoint %d %d %d",
              iCycles, iMinCyclesTeam, iMinCyclesOpp );
   iCyclesToObj = predictNrCyclesToPoint( o, posObj );

   if( iCyclesToObj < iMinCyclesOpp && SoccerTypes::isOpponent( o ) )
   {
    iMinCyclesOpp = iCyclesToObj;
    objFastestOpp = o;
   }
```

```
    if( iCyclesToObj < iMinCyclesTeam && SoccerTypes::isTeammate( o ) )
    {
      iMinCyclesTeam = iCyclesToObj;
      objFastestTeam = o;
    }
  }
}
iterateObjectDone( iIndex );

bool bContinue = true;
bool bLastCall = ( iCycles == PS->getPlayerWhenToIntercept() );
// log all features that have been solved
while( bContinue )
{
  featLog = FEATURE_ILLEGAL;
  if( bLastCall )
    iCycles = 100;

  // if player not set yet and either team or opp is smaller than iCycles
  // set fastest player
  if( bFinishedPlayer == false &&
      ( min( iMinCyclesTeam, iMinCyclesOpp ) <= iCycles
        ||
        bLastCall == true ) )
  {
    featLog             = FEATURE_FASTEST_PLAYER_TO_BALL;
    iCyclesLog          = iCycles;
    iCyclesFastestPlayer = iCycles;
    objLog              = (iMinCyclesTeam<=iMinCyclesOpp) ?
                          objFastestTeam : objFastestOpp;
    objFastestPlayer    = objLog;
    bFinishedPlayer     = true;
  }
  // if teammate not set yet and min cycles team smaller set it
  else if( bFinishedTeammates == false &&
          (iMinCyclesTeam <= iCycles || bFinishedOpponents == true
           || bLastCall))
  {
    if( bFinishedOpponents == true )
      objFastestTeam = getFastestInSetTo( OBJECT_SET_TEAMMATES, posObj,
                       VecPosition(0,0), 0, &iCycles );
    featLog           = FEATURE_FASTEST_TEAMMATE_TO_BALL;
    iCyclesLog        = iCycles;
    iCyclesFastestTeam = iCycles;
    objLog            = objFastestTeam;
    bFinishedTeammates = true;
```

```
}
else if( bFinishedTeammatesNoGoalie == false &&
   ( ( iMinCyclesTeam <= iCycles && objFastestTeam != getOwnGoalieType())
     || bFinishedOpponents == true || bLastCall ) )
{
  if( bFinishedOpponents == true && objFastestTeam == getOwnGoalieType())
    objFastestTeam=getFastestInSetTo( OBJECT_SET_TEAMMATES_NO_GOALIE,
                posObj, VecPosition(0,0), 0, &iCycles );
  featLog                 = FEATURE_FASTEST_TEAMMATE_TO_BALL_NO_GOALIE;
  iCyclesLog              = iCycles;
  iCyclesFastestTeamNoGoalie = iCycles;
  objLog                  = objFastestTeam;
  objFastestTeamNoGoalie  = objFastestTeam;
  bFinishedTeammatesNoGoalie = true;
}
else if( bFinishedMe == false &&
  ((iMinCyclesTeam <= iCycles && objFastestTeam == getAgentObjectType())
  || bFinishedOpponents == true || bLastCall ) )
{
  if( bFinishedOpponents == true &&
                                    objFastestTeam != getAgentObjectType())
    iCycles = predictNrCyclesToPoint( getAgentObjectType(), posObj );
  featLog         = FEATURE_INTERCEPT_CYCLES_ME;
  iCyclesLog      = iCycles;
  iCyclesFastestMe = iCycles;
  objLog          = getAgentObjectType();
  bFinishedMe     = true;
}
else if( bFinishedOpponents == false &&
        ( iMinCyclesOpp <= iCycles  || bLastCall ) )
{
  featLog          = FEATURE_FASTEST_OPPONENT_TO_BALL;
  iCyclesLog       = iCycles;
  iCyclesFastestOpp  = iCycles;
  objLog           = objFastestOpp;
  bFinishedOpponents = true;

}
else
  bContinue = false;

if( featLog != FEATURE_ILLEGAL )
{
  Log.log( 460, "log feature %d object %d in %d cycles sense %d see %d",
    featLog, objLog, iCyclesLog,getTimeLastSenseMessage().getTime(),
    getTimeLastSeeMessage().getTime()  );
```

```
        setFeature( featLog,
                Feature( getTimeLastSeeMessage(),
                        getTimeLastSenseMessage(),
                        getTimeLastHearMessage(), objLog,
                        getTimeLastSeeMessage().getTime() + iCyclesLog));
      }
     }
     bFinished = bFinishedTeammates && bFinishedTeammatesNoGoalie;
     if( bFinished == true )
       bOnlyMe = true;
     bFinished &= bFinishedMe ;
     if( bFinished == true )
       set = OBJECT_SET_OPPONENTS;
     bFinished &= bFinishedOpponents;
   }
   Log.log( 460, "creatIntercept: team %d me %d opp %d",
      iCyclesFastestTeamNoGoalie, iCyclesFastestMe, iCyclesFastestOpp );
}


/*! This method returns the fastest object to a specified object and fills the
    last argument with the predicted amount of cycles needed to intercept
    this object. Only objects within the set 'set' are taken into
    consideration and the objects have to have a confidence higher than the
    player confidence threshold defined in PlayerSettings.
    \param set ObjectSetT which denotes objects taken into consideration
    \param obj object type of object that should be intercepted
    \param iCyclesToIntercept will be filled with the amount of cycles needed
    \returns object that can intercept object obj fastest */
ObjectT WorldModel::getFastestInSetTo( ObjectSetT set, ObjectT obj,
                        int *iCyclesToIntercept )
{
  ObjectT  objFastestOpp  = OBJECT_ILLEGAL, objFastestTeam  = OBJECT_ILLEGAL;
  int      iCyclesFastestOpp = 30; // how much do we try
  int     iCyclesFastestTeam;
  bool  bSkip = false;

  FeatureT       feature_type = FEATURE_ILLEGAL;
  ObjectT        fastestObject = OBJECT_ILLEGAL;
  int            iCycles       = -1;

  if( obj == OBJECT_BALL )
  {
    switch( set )
    {
      case OBJECT_SET_OPPONENTS:
```

```
      feature_type = FEATURE_FASTEST_OPPONENT_TO_BALL;
      break;
    case OBJECT_SET_TEAMMATES:
      feature_type = FEATURE_FASTEST_TEAMMATE_TO_BALL;
      break;
    case OBJECT_SET_TEAMMATES_NO_GOALIE:
      feature_type = FEATURE_FASTEST_TEAMMATE_TO_BALL_NO_GOALIE;
      break;
    case OBJECT_SET_PLAYERS:
      objFastestOpp =
       getFastestInSetTo( OBJECT_SET_OPPONENTS, obj, &iCyclesFastestOpp);
      objFastestTeam =
       getFastestInSetTo( OBJECT_SET_TEAMMATES, obj, &iCyclesFastestTeam);
      if( iCyclesFastestOpp < iCyclesFastestTeam )
      {
        fastestObject = objFastestOpp;
        iCycles = iCyclesFastestOpp;
      }
      else
      {
        fastestObject = objFastestTeam;
        iCycles = iCyclesFastestTeam;
      }
      bSkip = true;
      feature_type = FEATURE_FASTEST_PLAYER_TO_BALL;
      break;
    default:
      cerr << "WorldModel::getFastestInSetTo unknown set: " << set << endl;
      return OBJECT_ILLEGAL;
  }
  if( isFeatureRelevant( feature_type ) )
  {
    int i = max(0,
          ((int)getFeature( feature_type ).getInfo() - getCurrentCycle() ));
    if( iCyclesToIntercept != NULL )
      *iCyclesToIntercept  = i;
    return getFeature( feature_type ).getObject();
  }

  Log.log( 460, "create intercept features" );
  createInterceptFeatures( );
  Log.log( 460, "call fastest again" );
  return getFastestInSetTo( set, obj, iCyclesToIntercept );
  if( set == OBJECT_SET_TEAMMATES || set ==
OBJECT_SET_TEAMMATES_NO_GOALIE )
    objFastestOpp =
```

```
      getFastestInSetTo( OBJECT_SET_OPPONENTS, obj, &iCyclesFastestOpp);
}

// no feature available, calculate information
double         dConfThr      = PS->getPlayerConfThr();
int            iCyclesToObj  ;
int            iMinCycles    = 100;
int            iIndex;
VecPosition    posObj;


while( bSkip == false &&
       iCycles < iMinCycles &&
       iCycles <= iCyclesFastestOpp )
{
 iCycles++;
 iMinCycles = 100;
 posObj     = predictPosAfterNrCycles( obj, iCycles );
 Log.log( 460, "fastest loop: %d fastest_opp %d",
          iCycles, iCyclesFastestOpp );
 for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
    o != OBJECT_ILLEGAL;
    o = iterateObjectNext ( iIndex, set, dConfThr ) )
 {
   if( getGlobalPosition(o).getDistanceTo(posObj)/SS->getPlayerSpeedMax()
           < iMinCycles &&
       getGlobalPosition(o).getDistanceTo(posObj)/SS->getPlayerSpeedMax()
           < iCycles + 1 )
   {
     Log.log( 460, "call predictNrCyclesToPoint %d %d",
              iCycles,iMinCycles );
     iCyclesToObj = predictNrCyclesToPoint( o, posObj );
     if( iCyclesToObj < iMinCycles )
     {
       iMinCycles = iCyclesToObj;
       fastestObject = o;
     }
   }
 }
 iterateObjectDone( iIndex );
}

// opponent is faster and we haven't calculated who can go to the
// interception point the fastest
if( fastestObject == OBJECT_ILLEGAL )
  fastestObject = getFastestInSetTo(set,posObj,VecPosition(0,0),0, &iCycles);
```

```
  if( iCyclesToIntercept != NULL )
    *iCyclesToIntercept  = iCycles;

  if( feature_type != FEATURE_ILLEGAL )
  {
    Log.log( 460, "log feature %d object %d in %d cycles sense %d see %d",
           feature_type, fastestObject, iCycles,getTimeLastSenseMessage().
           getTime(), getTimeLastSeeMessage().getTime()  );
    setFeature( feature_type,
              Feature( getTimeLastSeeMessage(),
                     getTimeLastSenseMessage(),
                     getTimeLastHearMessage(), fastestObject,
                     getTimeLastSeeMessage().getTime() + iCycles ) );
  }

  return fastestObject;
}

/*! This method returns the fastest object to another object that is currently
    located at position 'pos' and has velocity 'vel' that decays with a value
    'dDecay'. The last argument will be filled with the predicted amount of
    cycles needed to reach this object.
    \param set ObjectSetT which denotes objects taken into consideration
    \param pos current position of the object
    \param vel current velocity of the object
    \param dDecay decay value of the velocity of the object
    \param iCyclesToIntercept will be filled with the amount of cycles needed
    \returns object that can reach it fastest */
ObjectT WorldModel::getFastestInSetTo( ObjectSetT set, VecPosition pos,
                VecPosition vel, double dDecay, int *iCyclesToIntercept)
{
  double  dConfThr      = PS->getPlayerConfThr();
  ObjectT fastestObject = OBJECT_ILLEGAL;
  int     iCycles       = 0;
  int     iCyclesToObj  ;
  int     iMinCycles    = 100;
  int     iIndex;

  while( iCycles <= iMinCycles && iCycles < 100)
  {
    iCycles    = iCycles + 1  ;
    iMinCycles = 100;
    Log.log( 460, "fastest to point: %d", iCycles );
    for( ObjectT o = iterateObjectStart( iIndex, set, dConfThr );
       o != OBJECT_ILLEGAL;
```

```
      o = iterateObjectNext ( iIndex, set, dConfThr ) )
    {
      if( getGlobalPosition(o).getDistanceTo(pos)/SS->getPlayerSpeedMax()
          < iMinCycles )
      {
       iCyclesToObj = predictNrCyclesToPoint( o, pos );
       if( iCyclesToObj < iMinCycles )
       {
        iMinCycles    = iCyclesToObj;
        fastestObject = o;
       }
      }
    }
    iterateObjectDone( iIndex );
    pos += vel;
    vel *= dDecay;
    if( vel.getMagnitude( ) < EPSILON ) // we can quit
    {
     iCycles = iMinCycles;
     iMinCycles--;
    }
  }

  if( iCyclesToIntercept != NULL )
    *iCyclesToIntercept = iCycles;
  return fastestObject;
}

/*! This method returns the first empty spot in the set 'set'. The
    first empty spot is returned as the object which has a lower
    confidence than the threshold player_conf_thr defined in the
    PlayerSettings. This can be used when information of an unknown
    object is perceived. It is set on the first position where there
    is currently no information stored. If 'iUnknownPlayer' is
    specified, the range that corresponds to this unknown player is
    used to dermine the position.

    \param set ObjectSetT consisting of the objects to check
    \param iUnknownPlayer indicates the unknownplayer that has to be mapped

    \return object type of which currently no up to date information
    is stored */
ObjectT WorldModel::getFirstEmptySpotInSet( ObjectSetT set, int iUnknownPlayer)
{
  int     iIndex;
```

```
  for( ObjectT o = iterateObjectStart( iIndex, set,  0.0, true );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, set, 0.0, true ) )
  {
    if( getConfidence( o ) <= PS->getPlayerConfThr() &&
        o != getAgentObjectType() )
      return o;
  }
  return OBJECT_ILLEGAL;
}


/*! This method returns a truth value that represents whether the object
    supplied as the first argument was seen in the last see message. If
    "touch" information was received, i.e. the object was very close to the
    agent but not in its view cone, false is also returned.
    \param ObjectT that represent the type of the object to check
    \return bool indicating whether o was seen in the last see message */
bool WorldModel::isVisible( ObjectT o )
{
  Object *object = getObjectPtrFromType( o );

  if( object != NULL &&
      object->getTimeLastSeen() == getTimeLastSeeMessage() )
    return true;

  return false;
}

/*! This method determines whether the ball is kicakble, i.e. the ball
    is in the kickable range of the agent (see ServerSettings). This
    value can be different for the different heterogeneous player
    types.

    \return bool indicating whether ball can be kicked. */
bool WorldModel::isBallKickable()
{
  return getRelativeDistance( OBJECT_BALL ) < SS->getMaximalKickDist();
}

/*! This method determines whether the ball is catchable. This only applies
    to a goalie. Three things are tested:
    - the ball hasn't been catched in catch_ban_cycles (see ServerSettings)
      before the current cycle
    - the relative distance to the ball is smaller than the length of the
      catchable area length of the goalie (see also ServerSettings)
```

```
    - the ball is in the (own) penalty area.

    \return true when above three constraints are met, false otherwise. */
bool WorldModel::isBallCatchable()
{
  return getTimeSinceLastCatch()          > SS->getCatchBanCycle()  &&
         getRelativeDistance( OBJECT_BALL ) <= SS->getCatchableAreaL() &&
         isInOwnPenaltyArea( getBallPos() );
}

/*! This method checks whether the ball is currently heading towards our own
    goal. For the ball to be heading to our goal a few constraints must be met:
    - ball must be located in our penalty area
    - line of ball heading must intersect with goal line within goal width +
    small constant.
    - ball must pass goal line within 20 cycles.

    If all these constraints are met true is returned, false otherwise
    \return bool indicating whether the ball is heading towards our own goal */
bool WorldModel::isBallHeadingToGoal(  )
{
  int iSide = 1;

  if( isPenaltyUs() || isPenaltyThem() )
    iSide = ( getSide() == getSidePenalty() ) ? 1 : -1;

  if( !isConfidenceGood( OBJECT_BALL ) ||
      fabs( getBallPos().getX() ) <  PENALTY_X - 5.0 )
  {
    Log.log( 553, "ball not towards goal: confidence too low" );
    return false;
  }

  // make line from ball heading and goal line
  Line l = Line::makeLineFromPositionAndAngle(getBallPos(),getBallDirection());
  Line l2= Line::makeLineFromTwoPoints( getPosOwnGoal(), getPosOwnGoal() +
                                        VecPosition( 0, 10 ));

  // if intersection is outside goalwidth, not heading to goal
  VecPosition posIntersect = l.getIntersection( l2 );
  if( fabs(posIntersect.getY()) > SS->getGoalWidth()/2.0 + 3.0)
  {
    Log.log( 553, "ball not towards goal: outside goal %f",
       posIntersect.getY());
    return false;
  }
```

```
  // check whether ball will be behind goal line within 20 cycles.
  VecPosition pos    = getBallPos();
  int         iCycle = 1;
  while( fabs( pos.getX() ) < PITCH_LENGTH/2.0 && iCycle < 20)
  {
    pos = predictPosAfterNrCycles( OBJECT_BALL, iCycle );
    Log.log( 553, "predicted pos %d cycles: (%f,%f)" ,
      iCycle, pos.getX(), pos.getY() );
    iCycle ++;
  }

  return ( iCycle == 20 ) ? false : true;
}

/*! This method returns whether the ball is in our possesion. This is defined
     by the fact if the fastest player to the ball is a teammate or not.
     \return bool indicating whether a teammate is the fastest player to the
             ball. */
bool WorldModel::isBallInOurPossesion( )
{
  int     iCyc;
  ObjectT o = getFastestInSetTo( OBJECT_SET_PLAYERS, OBJECT_BALL, &iCyc );

  if( o == OBJECT_ILLEGAL )
    return false;
  if( SoccerTypes::isTeammate( o ) )
    return true;
  else
    return false;
}

/*! This method returns whether the ball lies in the own penalty area.
    \return bool indicating whether ball lies in own penalty area.     */
bool WorldModel::isBallInOwnPenaltyArea( )
{
  return isInOwnPenaltyArea( getBallPos() );
}

/*! This method returns whether the specified position lies in the own penalty
    area.
    \param pos position which should be checked
    \return bool indicating whether 'pos' lies in own penalty area. */
bool WorldModel::isInOwnPenaltyArea( VecPosition pos )
{
  ObjectT     objFlag = ( getSide() == SIDE_LEFT  )
```

```
                        ?  OBJECT_FLAG_P_L_C
                        :  OBJECT_FLAG_P_R_C ;

  if( isPenaltyUs() || isPenaltyThem() )
    objFlag = ( getSidePenalty() == SIDE_LEFT ) ? OBJECT_FLAG_P_L_C
                                        : OBJECT_FLAG_P_R_C ;
  VecPosition posFlag =SoccerTypes::getGlobalPositionFlag( objFlag, getSide());
  if( fabs(pos.getX())   > fabs(posFlag.getX()) &&
      fabs( pos.getY() ) < PENALTY_AREA_WIDTH/2.0 )
    return true;

  return false;
}

/*! This method returns whether the specified position lies in the opponent
    penalty area.
    \param pos position which should be checked
    \return boolean indicating whether 'pos' lies in opponent penalty area. */
bool WorldModel::isInTheirPenaltyArea( VecPosition pos )
{
  ObjectT     objFlag = ( getSide() == SIDE_LEFT )
                        ?  OBJECT_FLAG_P_R_C
                        :  OBJECT_FLAG_P_L_C ;
  VecPosition posFlag = SoccerTypes::getGlobalPositionFlag( objFlag,getSide());

  if ( pos.getX() > posFlag.getX() &&
       fabs(pos.getY()) < PENALTY_AREA_WIDTH/2.0 )
    return true;

  return false;
}

/*! This method determines whether the confidence for 'o' is good. The
    confidence of the object is compared to the player_conf_thr
    defined in PlayerSettings. When the confidence is higher than this
    value and the object does not equal the agent object type true is
    returned, otherwise false.

    \param o object of which confidence value should be returned
    \return bool indicating whether object information has good confidence. */
bool WorldModel::isConfidenceGood( ObjectT o )
{
  return getConfidence( o ) > PS->getPlayerConfThr() &&
         o != getAgentObjectType();
}
```

```
/*! This method determines whether the confidence for 'o' is very
    good. The confidence of the object is compared to the
    player_high_conf_thr defined in PlayerSettings. When the
    confidence is higher than this value and the object does not equal
    the agent object type true is returned, otherwise false.

    \param o object of which confidence value should be returned
    \return bool indicating whether object information has good confidence. */
bool WorldModel::isConfidenceVeryGood( ObjectT o )
{
  return getConfidence( o ) > PS->getPlayerHighConfThr() &&
         o != getAgentObjectType();
}

/*! This method checks whether the specified object stands onside. This is done
    by comparing the x coordinate of the object to the offside line.
    \return boolean indicating whether 'obj' stands onside. */
bool WorldModel::isOnside( ObjectT obj )
{
  return getGlobalPosition( obj ).getX() < getOffsideX() - 0.5 ;
}

/*! This method determines whether there stands an opponent in the global
    direction of the specified angle and in distance 'dDist'. An opponent is
    considered to stand in the global direction when the angle difference with
    the specified angle is smaller than 60 degrees.
    \param ang angle of the global direction in which to check opponents
    \param dDist distance in which opponents should be checked
    \return bool indicating wheter an opponent was found. */
bool WorldModel::isOpponentAtAngle( AngDeg ang , double dDist )
{
  VecPosition posAgent   = getAgentGlobalPosition();
  VecPosition posOpp;
  AngDeg      angOpp;
  int         iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_OPPONENTS );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_OPPONENTS ) )
  {
    posOpp    = getGlobalPosition( o );
    angOpp    = ( posOpp - posAgent ).getDirection() ;
    if( fabs( angOpp - ang ) < 60 &&
        posAgent.getDistanceTo( posOpp ) < dDist )
      return true;
    else if( fabs( angOpp - ang ) < 120 &&
```

```
            posAgent.getDistanceTo( posOpp ) < dDist/2.0 )
      return true;
   }
  iterateObjectDone( iIndex );
  return false;
}

/*! This method returns the inverse confidence, i.e. the time that belongs
    to the specified confidence. This can be used to determine
    the time the object was last seen when the confidence is given. Herefore
    the current time is used.
    \param dConf confidence
    \return server cycle the object was last seen. */
Time WorldModel::getTimeFromConfidence( double dConf )
{
  return getCurrentTime()-(int)((1.00-dConf)*100);
}

/*! This method returns the object type of the last opponent defender. This
    opponent resembles the offside line.
    \param if non-null dX will be filled with the x position of this object
    \return object type of the last opponent defender */
ObjectT WorldModel::getLastOpponentDefender( double *dX )
{
  double  dHighestX = 0.0;
  double  dSecondX  = 0.0, x;

  ObjectT o, oLast = OBJECT_ILLEGAL, oSecondLast = OBJECT_ILLEGAL;
  for( int i = 0; i < MAX_OPPONENTS ; i ++ )
  {
    o = Opponents[i].getType();
    if( isConfidenceGood( o ) )
    {
      x = Opponents[i].getGlobalPosition().getX();
      if( x > dHighestX )         // if larger x than highest
      {
        dSecondX    = dHighestX;  // make second the previous highest
        dHighestX   = x;          // and this the new one
        oSecondLast = oLast;
        oLast       = o;
      }
      else if( x > dSecondX )     // if smaller than 1st  and larger than 2nd
      {
        dSecondX    = x;          // make it the second
        oSecondLast = o;
      }
```

```
    }
  }

  // if highest x is outside pen_area, it cannot be the goalie (unless playing
  // Portugal ;-) ), so assume goalie is just not seen
  if( dHighestX < PENALTY_X && getOppGoalieType() == OBJECT_ILLEGAL )
  {
    dSecondX    = dHighestX;
    oSecondLast = oLast;
  }
  if( dX != NULL )
    *dX = dSecondX ;
  return oSecondLast;
}

/*! This method returns the x coordinate of the offside line using the known
    information in the WorldModel. If a player moves beyond this line, he
    stands offside. First the opponent with the second highest x coordinate is
    located, then the maximum of this x coordinate and the ball x coordinate
    is returned.
    \param bIncludeComm boolean indicating whether communicated offside line
         should also be included.
    \return x coordinate of the offside line. */
double WorldModel::getOffsideX( bool bIncludeComm )
{
  double  x, dAgentX;

  getLastOpponentDefender( &dAgentX );
  x = getBallPos().getX();
  x = max( x, dAgentX );
  if( bIncludeComm == true && getCurrentTime() - m_timeCommOffsideX < 3 )
    x = max( x, m_dCommOffsideX );
  return x ;
}

/*! This method returns the outer position on the field given a position 'pos'
    and a global angle 'ang'. The outer position is defined as the point on
    the field where the line created from this position and angle crosses
    either a side line, goal line or penalty line. To be on the safe side a
    small value is specified, which denotes the distance from the side line
    that should be returned.
    \param pos position on the field from which outer position should be
    calculated
    \param ang global angle which denotes the global direction in pos
    \param dDist distance from line
    \param bWithPenalty boolean denoting whether penalty area should be taken
```

```
                    into account (if false only goal line and side line
                    are used.
    \return position denoting the outer position on the field */
VecPosition WorldModel::getOuterPositionInField( VecPosition pos, AngDeg ang,
                         double dDist, bool bWithPenalty )
{
  VecPosition posShoot;

  // make shooting line using position and desired direction
  Line lineObj     = Line::makeLineFromPositionAndAngle( pos, ang );

  // get intersection point between the created line and goal line
  Line lineLength  = Line::makeLineFromPositionAndAngle(
                      VecPosition( PITCH_LENGTH/2.0 - dDist, 0.0 ), 90 );
  posShoot         = lineObj.getIntersection( lineLength );

  // check whether it first crosses the penalty line
  Line linePenalty = Line::makeLineFromPositionAndAngle(
                      VecPosition( PENALTY_X - dDist, 0.0 ), 90.0 );
  double dPenaltyY = lineObj.getIntersection(linePenalty).getY();

  if( bWithPenalty && fabs(dPenaltyY) < PENALTY_AREA_WIDTH/2.0 )
  {
    if( fabs(dPenaltyY) < PENALTY_AREA_WIDTH/2.0 - 5.0 ||   // crosses inside
        fabs(posShoot.getY()) <  PENALTY_AREA_WIDTH/2.0 )   // or ends inside
      posShoot = lineObj.getIntersection( linePenalty );
  }

  // check where it crosses the side line
  Line lineSide = ( ang < 0 )
     ? Line::makeLineFromPositionAndAngle(
                      VecPosition( 0.0, - PITCH_WIDTH/2.0 + dDist ),0.0 )
     : Line::makeLineFromPositionAndAngle(
                      VecPosition( 0.0, + PITCH_WIDTH/2.0 - dDist ),0.0 );

  if( fabs(posShoot.getY()) > PITCH_WIDTH/2.0 - dDist )
    posShoot = lineObj.getIntersection( lineSide );

  return posShoot;
}

/*! This method determines the (global) direction which has the largest
    angle between the opponents and is located in the interval angMin..
    angMax.
    \param origin of which the angles angMin and angMax are based on.
    \param angMin minimal global direction that should be returned
```

```
    \param angMax maximal global direction that should be returned
    \param angLargest will contain the size of the largest angle of the
           direction that is returned
    \param dDist only opponents with relative distance smaller than this value
           will be taken into account.
    \return global direction with the largest angle between opponents */
AngDeg WorldModel::getDirectionOfWidestAngle(VecPosition posOrg, AngDeg angMin,
                        AngDeg angMax, AngDeg *angLargest, double dDist)
{
  list<double> v;
  list<double> v2;
  double       temp;
  int          iIndex;
  double       dConf  = PS->getPlayerConfThr();

  // add all angles of all the opponents to the list v
  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_OPPONENTS, dConf );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_OPPONENTS, dConf ) )
  {
    if( getRelativeDistance( o ) < dDist )
      v.push_back( (getGlobalPosition(o)-posOrg).getDirection());
  }
  iterateObjectDone( iIndex );
  v.sort();

  // if goalkeeper is spotted and he is located within the range that we want
  // to shoot at, make sure the angle with the goalkeeper is large enough, since
  // he has better intercepting capabilities than the normal players

  ObjectT     objGoalie = getOppGoalieType();
  VecPosition posGoalie = getGlobalPosition( objGoalie );
  AngDeg      angGoalie;

  if( objGoalie != OBJECT_ILLEGAL && posOrg.getX() > PITCH_LENGTH/4.0 &&
      posOrg.getDistanceTo( posGoalie ) < dDist )
  {
    angGoalie = ( posGoalie - posOrg ).getDirection();
    Log.log( 560, "direction_widest_angle: min %f max %f angGoalie %f",
                                      angMin, angMax, angGoalie );

    if( posOrg.getY() > 0 ) // right side of the field
    {
      angGoalie = VecPosition::normalizeAngle( angGoalie - 33 );
      angMax    = max( angMin, min( angGoalie, angMax ) );
    }
```

```
  else
  {
    angGoalie = VecPosition::normalizeAngle( angGoalie + 33 );
    angMin    = min( angMax, max( angMin, angGoalie ) );
  }
  Log.log( 560, "direction_widest_angle after: %f %f", angMin, angMax );
}


// Create new list with only opponents from interval [angMin..angMax].
// Note that opponents outside angMin and angMax can have an influence
// on the largest angle between the opponents, so they should be accounted
// for. To this end, a projection is defined in both angMin and angMax.
// The opponent with the smallest global angle difference a to angMin
// (either inside or outside the interval [angMin..angMax]) is determined
// and an extra angle angMin - a is added to the list. The situation for
// angMax is analogous.

double absMin     = 1000;
double absMax     = 1000;
double angProjMin = angMin;
double angProjMax = angMax;
double array[MAX_OPPONENTS+2];

while( v.size() > 0 )
{
  if( fabs( v.front() - angMin ) < absMin )      // opp near angMin
  {
    absMin     = fabs( v.front() - angMin ) ;    // make angMin wider
    angProjMin = angMin - absMin;                // to take him into account
  }
  if( fabs( v.front() - angMax ) < absMax )      // opp near angMax
  {
    absMax     = fabs( v.front() - angMax ) ;    // make angMax wider
    angProjMax = angMax + absMax;                // to take him into account
  }
  if( v.front() > angMin && v.front() < angMax ) // opp in range
    v2.push_back( v.front() );                   // add him
  v.pop_front();
}

// make all angles relative to angProjMin which has angle 0 and set them in
// the range 0..360, where the range -180..0 is moved to 180..360. Do this by
// adding 360 and then subtracting 360 if value is larger than 360.
v.push_back( 0 );
while( v2.size() > 0 )  // for all the opponents
```

```
{
  temp = VecPosition::normalizeAngle(v2.front()-angProjMin)+360.0;
  if( temp > 360 )
    temp -= 360;
  v.push_back( temp );
  v2.pop_front();
}
// add max projection.
temp = VecPosition::normalizeAngle(angProjMax-angProjMin)+360.0;
if( temp > 360 )
  temp -= 360;

v.push_back( temp );

// sort the list
v.sort();

// put all the values in an array
int i = 0;
while( v.size() > 0 )
{
  array[i++] = v.front();
  v.pop_front();
}

// find the largest angle and determine the associated midpoint direction
double dLargest = -1000;
double d;
double ang      = UnknownAngleValue;
for( int j = 0; j < i - 1 ; j ++ )
{
  d = VecPosition::normalizeAngle(( array[j+1] - array[j] )/2.0);
  if( d > dLargest )
  {
    ang = angProjMin + array[j] + d;
    ang = VecPosition::normalizeAngle( ang );
    dLargest = d;
  }
}

if( ang == UnknownAngleValue ) // no angle found -> get angle in between
{
  ang = getBisectorTwoAngles( angMin, angMax );
  if( angLargest != NULL )
    *angLargest = 360;
}
```

```
  else if( angLargest != NULL )
    *angLargest = dLargest;

  return ang;
}

/*! This method returns whether the position 'pos' is inside the playfield. */
bool WorldModel::isInField( VecPosition pos, double dMargin )
{
  return Rect(
          VecPosition( + PITCH_LENGTH/2.0 - dMargin,
                  - PITCH_WIDTH/2.0  + dMargin ),
          VecPosition( - PITCH_LENGTH/2.0 + dMargin,
                  + PITCH_WIDTH/2.0  - dMargin )
          ).isInside( pos );
}

/*! This method returns whether the position 'pos' is before the opp goal. */
bool WorldModel::isBeforeGoal( VecPosition pos )
{
  return Rect(
          VecPosition( + PENALTY_X - 2,    - ( SS->getGoalWidth()/2.0 + 1)),
          VecPosition( + PITCH_LENGTH/2.0, + ( SS->getGoalWidth()/2.0 + 1))
          ).isInside( pos );
}

/*! This method determine the strategic position for the specified object. This
    is done using the Formations class. In this class all information
    about the current formation, player number in formation and otheic
    values are stored. The strategic position is based on the position of
    the ball. If the confidence in the position of the ball is lower than the
    threshold defined in PlayerSettings, it is assumed that the ball is at
    position (0,0).
    \param obj for which the strategic position should be calculated.
    \param ft formation for which to calculate the strategic position
    \return VecPosition strategic position for player 'iPlayer' */
VecPosition WorldModel::getStrategicPosition( ObjectT obj, FormationT ft )
{
  return getStrategicPosition( SoccerTypes::getIndex( obj ), ft );
}

/*! This method determine the strategic position for the specified player. This
    is done using the Formations class. In this class all information
    about the current formation, player number in formation and otheic
    values are stored. The strategic position is based on the position of
    the ball. If the confidence in the position of the ball is lower than the
```

```
    threshold defined in PlayerSettings, it is assumed that the ball is at
    position (0,0).
    \param iPlayer role in formation for which strategic position should be
      returnd. With default value is -1 it is assumed that the strategic
      position of the agent itself should be returned.
    \param ft formation for which to calculate the strategic position
    \return VecPosition strategic position for player 'iPlayer' */
VecPosition WorldModel::getStrategicPosition( int iPlayer, FormationT ft )
{
  if( iPlayer > MAX_TEAMMATES )
    cerr << "WM:getStrategicPosition with player nr " << iPlayer << endl;

  VecPosition pos, posBall = getBallPos();
  bool bOwnBall = isBallInOurPossesion();

  // -1 is default -> get player number in formation
  if( iPlayer == -1 )
    iPlayer = formations->getPlayerInFormation();

  // get maximal allowed x coordinate, this is offside x coordinate
  double dMaxX = max( -0.5, getOffsideX() - 1.5 );

  if( bOwnBall &&
      getGlobalPosition(
        SoccerTypes::getTeammateObjectFromIndex(iPlayer)).getX()
      < posBall.getX() )
    dMaxX = max( dMaxX, posBall.getX()  );

  // after standing offside we are not allowed to move for ball
  // with a goal kick of them we are not allowed to move into penalty area

  if( isGoalKickThem() )
    dMaxX = min( dMaxX, PENALTY_X - 1.0 );
  else if( isBeforeKickOff() )
    dMaxX = min( dMaxX, -2.0 );
  else if ( isOffsideUs() )
    dMaxX = posBall.getX() - 0.5;

  // change the ball position on which strategic position is based
  // depending on the different deadball situation and thus the
  // expected movement of the ball
  if( isBeforeKickOff() )
    posBall.setVecPosition( 0, 0 );
  else if( isGoalKickUs() ||
      getTimeSinceLastCatch(  ) < PS->getCyclesCatchWait() + 5  ||
      ( isFreeKickUs() && posBall.getX() < - PENALTY_X ) )
```

```
    posBall.setX( -PITCH_LENGTH/4 + 5.0 );
  else if( getConfidence( OBJECT_BALL ) < PS->getBallConfThr() )
    posBall.setVecPosition( 0.0, 0.0 );
  else if( isGoalKickThem() ||
          ( isFreeKickThem() && posBall.getX() > PENALTY_X ) )
    posBall.setX( PENALTY_X - 10.0 );
  else if( isFreeKickThem() )
    posBall.setX( posBall.getX() - 5.0 );
  else if( isBallInOurPossesion() &&
          !( isDeadBallUs() || isDeadBallThem() ) )
    posBall.setX( posBall.getX() + 5.0 );
  else if( posBall.getX() < - PENALTY_X + 5.0 )
    posBall = predictPosAfterNrCycles( OBJECT_BALL, 3 );

  // get the strategic position
  pos = formations->getStrategicPosition( iPlayer, posBall, dMaxX,
                      bOwnBall, PS->getMaxYPercentage(),
                                                                  ft );
  return pos;
}

/*! This method returns a global position on the field which denotes
    the position to mark position 'pos'. It receives three arguments:
    a position pos (usually an opponent) that the agent wants to mark,
    a distance 'dDist' representing the desired distance between o and
    the marking position and a type indicator that denotes the type of
    marking that is required. We distinguish three types of marking: -
    MARK BALL: marking the opponent by standing at a distance 'dDist'
    away from him on the line between him and the ball. This type of
    marking will make it di±cult for the opponent to receive a pass.
    - MARK GOAL: marking the opponent by standing at a distance
    'dDist' away from him on the line between him and the center point
    of the goal he attacks. This type of marking will make it
    difficult for the opponent to score a goal.  - MARK BISECTOR:
    marking the opponent by standing at a distance 'dDist' away from
    him on the bisector of the ball-opponent-goal angle. This type of
    marking enables the agent to intercept both a direct and a leading
    pass to the opponent.  \param pos position that has to be marked
    \param dDist distance marking position is located from object
    position \param mark marking technique that should be used \return
    position that is the marking position. */
VecPosition WorldModel::getMarkingPosition( VecPosition pos, double dDist,
                                                              MarkT mark)
{
  VecPosition posBall  = getBallPos();
    //edictPosAfterNrCycles( OBJECT_BALL, 3 );
```

```
  VecPosition posGoal  = getPosOwnGoal( );
  if( posBall.getX() < - PITCH_LENGTH/2.0 + 10.0 )
    posGoal.setX( posBall.getX() + 1  );
  else if( posBall.getX() > -PITCH_LENGTH/3.0 )
  {
    posGoal.setX( -PITCH_LENGTH/2.0 );
    double dY  = posBall.getY();
    if( fabs( dY ) > 12 )
      dY += ( sign( dY ) > 0   ) -5 ? : 5 ;
    posGoal.setY( dY );
  }

  VecPosition posAgent = getAgentGlobalPosition();
  VecPosition posMark;
  AngDeg      ang, angToGoal, angToBall;

  if( mark == MARK_GOAL )                    // position in direction goal
  {
    angToGoal = (posGoal-pos).getDirection( );
    Line line = Line::makeLineFromTwoPoints( pos, posGoal );

    // we want to know when distance from ball to point p equals distance
    // from opp to point p :
    // d1 + d3 = sqrt(d2^2 + d3^2) > (d1+d3)^2 = d2^2 + d3^2 =>
    // d1^2 + 2*d1*d3 = d2^2 -> d3 = (d2^2 - d1^2 ) / 2*d1
    double dCalcDist;
    VecPosition posIntersect = line.getPointOnLineClosestTo( posAgent );
    double dDistAgent = posIntersect.getDistanceTo( posAgent );
    double dDistOpp = posIntersect.getDistanceTo( pos );
    dCalcDist = (dDistAgent*dDistAgent-dDistOpp*dDistOpp)/(2*dDistOpp);
    double dExtra = 2.0;
    //    if( posBall.getX() <  PENALTY_X + 5 )
    if( pos.getDistanceTo(posAgent) < 5 )
      dExtra = 0.0;
    dCalcDist += dDistOpp + dExtra;
    Log.log( 513, "dDistOpp %f dDistAgent %f calc %f min %f",
                                        dDistOpp, dDistAgent, dCalcDist,
0.75*pos.getDistanceTo(posGoal));
    dCalcDist = min( dCalcDist, 0.75*pos.getDistanceTo( posGoal ) );
    double x = -PITCH_LENGTH/2 + 4;
    double y = line.getYGivenX( x);
    posMark = pos + VecPosition( dCalcDist, angToGoal, POLAR );
    if( posMark.getX() < x )
      {
                                    Log.log( 513, "change posmark to (%f,%f)", x, y );
                                    posMark.setVecPosition( x, y );
```

```
      }
    // if interception point iss outside range or very close to marking
    // point, but far away from opp (is this possible?) move closer.
    if( ! line.isInBetween( posMark, pos, posGoal ) ||
                                    ( posMark.getDistanceTo( posAgent ) < 1.5 &&
                                      posMark.getDistanceTo( pos ) > 2*dDist ) )
    {
      Log.log( 513, "set marking position at dDist %f", min(dDistAgent,7.0) );
      posMark   = pos + VecPosition( min( dDistAgent, 7.0 ), angToGoal, POLAR );
    }
    Log.log( 513, "marking position calc (%f,%f) pos(%f,%f) calcdist %f",
                                        posMark.getX(), posMark.getY(), pos.getX(), pos.getY(),
                                        dCalcDist );
  }
  else if( mark == MARK_BALL )                // position in direction ball
  {
    angToBall = (posBall-pos).getDirection( );
    posMark   = pos + VecPosition( dDist, angToBall, POLAR );
  }
  else if( mark == MARK_BISECTOR )            // pos between ball and goal
  {
    angToBall = (posBall - pos).getDirection( );
    angToGoal = (posGoal - pos).getDirection( );
    ang       = getBisectorTwoAngles( angToBall, angToGoal );
    posMark   = pos + VecPosition( dDist, ang ,POLAR );
  }
  if( fabs( posMark.getX() ) > PITCH_LENGTH/2.0 - 2.0 )
    posMark.setX( sign(posMark.getX())*(PITCH_LENGTH/2.0 - 2.0) );
  return posMark;

}


/*! The actual power with which the ball is kicked depends on the
    relative location of the ball to the player. The kick is more
    powerful when the ball is very close to and in front of the player.
    The actual kickpowerrate with which the power of the kick command
    is multiplied is equal to<BR>
    KickPowerRate*(1 - 0.25*DirDiff / 180 -
    0.25*(DistBall-PlayerSize-BallSize)/KickableMargin)<BR>
    with DirDiff = global angle of the ball rel to the body dir agent<BR>
        DistBall = the distance from the center of the player to the ball<BR>
    See soccermanual
    \return the actual kick power rate with which power is multiplied  */
double WorldModel::getActualKickPowerRate( )
{
```

```
 // true indicates that relative angle to body should be returned
 double dir_diff      = fabs( getRelativeAngle( OBJECT_BALL, true ) );
 double dist          = getRelativeDistance( OBJECT_BALL ) -
                    SS->getPlayerSize( ) - SS->getBallSize( );
 return SS->getKickPowerRate() *
        ( 1 - 0.25 * dir_diff/180.0 - 0.25 * dist / SS->getKickableMargin());
}

/*! The actual power with which the ball must be kicked depends on the
    relative location of the ball to the player. The kick is more
    powerful when the ball is very close to and in front of the player.
    The actual power with which the ball must be kicked is equal to<BR>
    Speed / KickPowerRate*(1 - 0.25*DirDiff / 180 -
    0.25*(DistBall-PlayerSize-BallSize)/KickableMargin)<BR>
    with DirDiff = global angle of the ball rel to the body dir agent<BR>
        DistBall = the distance from the center of the player to the ball<BR>
    See soccermanual for further information.
    This method receives a speed vector which the ball should have after the
    kick command and calculates the power for the kick command to reach this.
    This value can be higher than is possible to shoot!
    \param dDesiredSpeed the desired speed after the kick command
    \return the actual power for kick command to get dDesiredSpeed */
double WorldModel::getKickPowerForSpeed( double dDesiredSpeed )
{
  // acceleration after kick is calculated by power * eff_kick_power_rate
  // so actual kick power is acceleration / eff_kick_power_rate
  return dDesiredSpeed / getActualKickPowerRate( );
}


/*! This method determines the power with which the ball must be kicked in
    order to travel a given distance and still have a speed after that
    distance
    \param dDistance distance ball should travel
    \param dEndSpeed speed ball should have at target position
    \return power value for kick command */
double WorldModel::getKickSpeedToTravel( double dDistance, double dEndSpeed )
{
  // if endspeed is zero we have an infinite series and return the first term
  // that corresponds to the distance that has to be travelled.
  if( dEndSpeed < 0.0001  )
    return Geometry::getFirstInfGeomSeries(dDistance, SS->getBallDecay() );

  // use geometric series to calculate number of steps and with that the
  // velocity to give to the ball, we start at endspeed and therefore use
  // the inverse of the ball decay (r).
```

```
  // s = a + a*r + .. a*r^n since we calculated from endspeed (a) to
  // firstspeed, firstspeed equals a*r^n = endspeed*r^nr_steps
  double dNrSteps = Geometry::getLengthGeomSeries( dEndSpeed,
                              1.0/SS->getBallDecay( ), dDistance );
  return getFirstSpeedFromEndSpeed( dEndSpeed, (int)rint(dNrSteps) ) ;
}


/*! This method returns the speed that has to be given to the ball when
    it should have an endspeed of 'dEndSpeed' after 'dCycles' number of cycles.
    This can be calculated using a geometric series.
    \param dEndSpeed desired end speed for the ball
    \param dCycles nr of cycles after which ball should have speed 'dEndSpeed'
    \return initial speed given to the ball to have speed 'dEndSpeed' after
    'dCycles' cycles. */
double WorldModel::getFirstSpeedFromEndSpeed( double dEndSpeed, double dCycles,
                                      double dDecay  )
{
  if( dDecay < 0 )
    dDecay = SS->getBallDecay();

  // geometric serie: s = a + a*r^1 + .. + a*r^n, now given endspeed = a*r^n ->
  // endspeed = firstspeed * ratio ^ length ->
  // firstpeed = endspeed * ( 1 / ratio ) ^ length
  return dEndSpeed * pow( 1 / dDecay, dCycles );
}

/*! This method returns the speed that has to be given to an object when
    it should have travelled a distance 'dDist' after 'dCycles' number of
    cycles. This can be calculated using a geometric series where 'dDecay'
    is the used decay factor (default this value equals ball_decay).
    \param dDist distance the ball has to travel
    \param dCycles nr of cycles after which ball should have travelled 'dDist'
    \param dDecay decay of the geometric series.
    \return initial speed for the ball to travel 'dDist' in 'dCycles' cycles */
double WorldModel::getFirstSpeedFromDist( double dDist, double dCycles, double
                dDecay )
{
  if( dDecay < 0 )
    dDecay = SS->getBallDecay();

  return Geometry::getFirstGeomSeries( dDist, dDecay, dCycles);
}

/*! This method returns the speed the ball will have after 'dCycles' cycles
    when it is given an initial speed of 'dFirstSpeed'.
```

```
   This can be calculated using a geometric series.
   \param dFirstSpeed given speed to the ball
   \param dCycles nr of cycles after which ball speed should be determined
   \return speed of the ball after 'dCycles' server cycles */
double WorldModel::getEndSpeedFromFirstSpeed(double dFirstSpeed,double dCycles)
{
 // geometric series: s = a + a*r^1 + .. + a*r^n, with firstspeed = a ->
 // endspeed = firstspeed * ratio ^ length ;
 return dFirstSpeed * pow( SS->getBallDecay(), dCycles );
}

/*! This method determines the angle that should be sent to the soccerserver
when
   the player wants to turn angDesiredAngle. This value depends on the current
   velocity and the inertia moment of the player
   \param angDesiredAngle angle that player wants to turn
   \param dSpeed current speed of the player
   \return angle that can be sent with turn command */
AngDeg WorldModel::getAngleForTurn( AngDeg angDesiredAngle, double dSpeed,
                                    ObjectT obj )
{
 AngDeg a = angDesiredAngle * (1.0 + getInertiaMoment( obj ) * dSpeed );
 if( a > SS->getMaxMoment() )
   return SS->getMaxMoment() ;
 else if ( a < SS->getMinMoment() )
   return SS->getMinMoment() ;
 else
   return a;
}

/*! This method determines the actual angle that is used when 'angTurn' is
   sent to the SoccerServer. This value depends on the current velocity and
   the inertia moment of the player
   \param angAngleForSend angle send with turn command
   \param dSpeed current speed of the player
   \return actual angle that player is turned */
AngDeg WorldModel::getActualTurnAngle( AngDeg angTurn,double dSpeed,ObjectT o )
{
 return angTurn / (1.0 + getInertiaMoment( o ) * dSpeed );
}

/*! This method determines the optimal dash power to mantain an optimal speed
   When the current speed is too high and the distance is very small, a
   negative dash is performed. Otherwise the difference with the maximal speed
   is determined and the dash power rate is set to compensate for this
   difference.
```

```
    \param posRelTo relative point to which we want to dash
    \param angBody body angle of the agent
    \param vel current velocity of the agent
    \param dEffort current effort of the player
    \param iCycles desired number of cycles to reach this point
    \return dash power that should be sent with dash command */
double WorldModel::getPowerForDash( VecPosition posRelTo, AngDeg angBody,
            VecPosition vel, double dEffort, int iCycles )
{
  // the distance desired is the x-direction to the relative position we
  // we want to move to. If point lies far away, we dash maximal. Furthermore
  // we subtract the x contribution of the velocity because it is not necessary
  // to dash maximal.
  double dDist = posRelTo.rotate(-angBody).getX(); // get distance in direction
  if( iCycles <= 0 ) iCycles = 1;
  double dAcc  = getFirstSpeedFromDist(dDist,iCycles,SS->getPlayerDecay());
                                      // get speed to travel now
  if( dAcc > SS->getPlayerSpeedMax() )           // if too far away
    dAcc = SS->getPlayerSpeedMax();              // set maximum speed
  dAcc -= vel.rotate(-angBody).getX();           // subtract current velocity

  // acceleration = dash_power * dash_power_rate * effort ->
  // dash_power = acceleration / (dash_power_rate * effort )
  double dDashPower = dAcc/(SS->getDashPowerRate() * dEffort );
  if( dDashPower > SS->getMaxPower() )
    return SS->getMaxPower();
  else if( dDashPower < SS->getMinPower() )
    return SS->getMinPower();
  else
    return dDashPower;
}


/*! This method returns the closest player in the current formation to the
    position 'pos'. In case of a dead ball situation this method can be used
    to determine whether you should move to the ball.
    \param pos position which is used in the comparision
    \param bIncludeGoalie boolean to determine whether goalkeeper should be
          taken into account
    \param ps player set to which the returned player must belong
    \return role number of the agent in the current formation who is closest
          to this position */
int WorldModel::getClosestPlayerInFormationTo( VecPosition pos,
                                    bool bIncludeGoalie,
                                    ObjectT objWithout,
                                    PlayerSetT ps,
```

```
                                    FormationT ft )
{
 double      dDist = 1000.0;
 VecPosition posStrat;
 int         iPlayer = -1;

 for( int i = 0; i < MAX_TEAMMATES; i++ )
 {
  if( bIncludeGoalie == false && i == 0 )
    continue;
  else if( objWithout == SoccerTypes::getTeammateObjectFromIndex( i ) )
    continue;
  else if( !SoccerTypes::isPlayerTypeInSet(
                                          formations->getPlayerType(i,ft), ps ))
    continue;

  posStrat = getStrategicPosition( i, ft );

  if( isDeadBallUs( )&&
      getBallPos().getX() < PITCH_LENGTH/3.0 &&
      i >= 9 )
    ;  // don't use attackers when in dead ball situation and not upfront
  else if( posStrat.getDistanceTo( pos ) < dDist )
  {
    dDist   = posStrat.getDistanceTo( pos );
    iPlayer = i;
  }
 }
 return iPlayer;
}
```

```
#include <stdio.h>
#include "WorldModel.h"

/*************************************************************************
/
/************************** PREDICTIONS
*************************************/
/*************************************************************************
/

/*! This method predicts the state of an player after it performs a specific
   SoccerCommand. The current state of the agent is passed by giving the
   position, velocity and body and neck direction. These arguments are
   updated and after return of this method will contain the new values
   as if the command was performed.
   \param com SoccerCommand that will be performed
   \param pos current position of the object.
   \param vel current velocity of the object
   \param angGlobalBody global body angle
   \param angGlobalBody global neck angle
   \param current stamina information.
   \return boolean which indicates whether values were updated */
bool WorldModel::predictStateAfterCommand( SoccerCommand com,
   VecPosition *pos, VecPosition *vel, AngDeg *angGlobalBody,
   AngDeg *angGlobalNeck, ObjectT obj, Stamina *sta )
{
 switch( com.commandType ) // based on kind of command, choose action
 {
   case CMD_DASH:
     predictStateAfterDash( com.dPower, pos, vel, sta, *angGlobalBody, obj );
     break;
   case CMD_TURN:
     predictStateAfterTurn( com.dAngle, pos, vel,
                        angGlobalBody, angGlobalNeck, obj, sta);
     break;
   case CMD_TURNNECK: // note that position and velocity are not updated
     *angGlobalNeck = VecPosition::normalizeAngle(*angGlobalNeck+com.dAngle);
     break;
   case CMD_KICK:
   case CMD_CATCH:
   case CMD_TACKLE:
     predictStateAfterDash( 0.0, pos, vel, sta, *angGlobalBody, obj );
     break;
   case CMD_MOVE:
     pos->setVecPosition( com.dX, com.dY );
     vel->setMagnitude( 0.0 );
```

```
      break;
    case CMD_ILLEGAL:
      predictStateAfterDash( 0.01, pos, vel, sta, *angGlobalBody, obj );
      break;
    default:
      return false;
  }
  return true;
}

/*! This method predicts the state of the agent after it performs a specific
    SoccerCommand. This method makes use of the method predictInfoAfterCommand.
    All arguments are initialized with the current values of the agent.
    \param com SoccerCommand that will be performed
    \param pos will be filled with updated position of agent
    \param vel will be filled with updated velocity of agent
    \param angGlobalBody will be filled with global body angle of agent
    \param angGlobalBody will be filled with global neck angle of agent
    \param current will be filled with stamina information of agent
    \return boolean which indicates whether values were updated */
bool WorldModel::predictAgentStateAfterCommand( SoccerCommand com,
    VecPosition *pos, VecPosition *vel, AngDeg *angGlobalBody,
    AngDeg *angGlobalNeck, Stamina *sta )
{
  *pos          = getAgentGlobalPosition();
  *vel          = getAgentGlobalVelocity();
  *angGlobalBody = getAgentGlobalBodyAngle();
  *angGlobalNeck = getAgentGlobalNeckAngle();
  *sta          = getAgentStamina();
  predictStateAfterCommand( com, pos, vel, angGlobalBody, angGlobalNeck,
                            getAgentObjectType(), sta );

  return true;
}

/*! This method predicts the state of the agent after it performs a specific
    SoccerCommand. This method makes use of the method predictInfoAfterCommand.
    All arguments are initialized with the current values of the agent.
    \param com SoccerCommand that will be performed
    \param pos will be filled with updated position of agent
    \param vel will be filled with updated velocity of agent
    \param angGlobalBody will be filled with global body angle of agent
    \param angGlobalBody will be filled with global neck angle of agent
    \param current will be filled with stamina information of agent
    \return boolean which indicates whether values were updated */
bool WorldModel::predictObjectStateAfterCommand( ObjectT obj,SoccerCommand com,
```

```
    VecPosition *pos, VecPosition *vel, AngDeg *angGlobalBody,
    AngDeg *angGlobalNeck, Stamina *sta )
{
  if( obj == getAgentObjectType() )
    return predictAgentStateAfterCommand(
              com, pos, vel, angGlobalBody, angGlobalNeck, sta );
  *pos          = getGlobalPosition( obj );
  *vel          = getGlobalVelocity( obj );
  *angGlobalBody = getGlobalBodyAngle( obj  );
  *angGlobalNeck = getGlobalNeckAngle(obj );
  predictStateAfterCommand( com, pos, vel, angGlobalBody, angGlobalNeck, obj );

  return true;
}

/*! This method returns the global position of the agent after the specified
    command is performed. This method makes use of the method
    'predictAgentInfoAfterCommand'
    \param com SoccerCommand that will be performed.
    \return VecPosition new global position of the agent. */
VecPosition WorldModel::predictAgentPosAfterCommand( SoccerCommand com )
{
  VecPosition p1, p2;
  AngDeg      a1, a2;
  Stamina     sta;
  predictAgentStateAfterCommand( com, &p1, &p2, &a1, &a2, &sta );
  return p1;
}

/*! This method determines the state of a player after a dash command is
    performed. The current state of the player is specified by the passed
    arguments. After this method returns, all arguments are updated.
    \param pos initial position, will be changed to the predicted position
    \param vel intital velocity, will be changed to the predicted velocity
    \param dActualPower actual power that is send with dash command
    \param sta pointer to stamina, when not NULL, effort is used and updated
    \param dDirection direction of dash */
void WorldModel::predictStateAfterDash( double dActualPower, VecPosition *pos,
          VecPosition *vel, Stamina *sta, double dDirection, ObjectT obj )
{
  // get acceleration associated with actualpower
  double dEffort = ( sta != NULL ) ? sta->getEffort() : getEffortMax( obj );
  double dAcc    = dActualPower * getDashPowerRate( obj ) * dEffort;

  // add it to the velocity; negative acceleration in backward direction
  if( dAcc > 0 )
```

```
    *vel += VecPosition::getVecPositionFromPolar( dAcc, dDirection );
  else
    *vel += VecPosition::getVecPositionFromPolar( fabs(dAcc),
        VecPosition::normalizeAngle(dDirection+180));

  // check if velocity doesn't exceed maximum speed
  if( vel->getMagnitude() > SS->getPlayerSpeedMax() )
    vel->setMagnitude( SS->getPlayerSpeedMax() );

  // add velocity to current global position and decrease velocity
  *pos += *vel;
  *vel *= getPlayerDecay(obj);
  if( sta != NULL )
    predictStaminaAfterDash( dActualPower, sta );
}

/*! This method determines the state of a player after a turn command is
    performed. The global position is updated with the velocity and the
    velocity is updated. Then the actual turn angle is calculated taken the
    inertia into account. This actual turn angle is used to update both
    the global body and global neck direction.
    \param dSendAngle actual angle given in command
    \param pos initial position, will be changed to the predicted position
    \param vel intital velocity, will be changed to the predicted velocity
    \param angBody global body direction
    \param angNeck global neck direction
    \param sta Stamina of player can be NULL */
void WorldModel::predictStateAfterTurn( AngDeg dSendAngle, VecPosition *pos,
         VecPosition *vel, AngDeg *angBody, AngDeg *angNeck, ObjectT obj,
         Stamina *sta )
{
  // calculate effective turning angle and neck accordingly
  double dEffectiveAngle;
  dEffectiveAngle = getActualTurnAngle( dSendAngle, vel->getMagnitude(), obj );
  *angBody = VecPosition::normalizeAngle( *angBody + dEffectiveAngle );
  *angNeck = VecPosition::normalizeAngle( *angNeck + dEffectiveAngle );

  // update as if dashed with no power
  predictStateAfterDash( 0.0, pos, vel, sta, *angBody, obj );
  return;
}

void WorldModel::predictBallInfoAfterCommand( SoccerCommand soc,
                VecPosition *pos, VecPosition *vel )
{
  VecPosition posBall = getGlobalPosition( OBJECT_BALL ) ;
```

```
  VecPosition velBall = getGlobalVelocity( OBJECT_BALL );

  if( soc.commandType == CMD_KICK )
  {
    int iAng   = (int)soc.dAngle;
    int iPower = (int)soc.dPower;

    // make angle relative to body
    // calculate added acceleration and add it to current velocity
    AngDeg ang = VecPosition::normalizeAngle(iAng+getAgentGlobalBodyAngle());
    velBall += VecPosition( getActualKickPowerRate()*iPower, ang, POLAR ) ;
    if( velBall.getMagnitude() > SS->getBallSpeedMax() )
      velBall.setMagnitude( SS->getBallSpeedMax() );
    Log.log( 600, "ang: %f kick_rate %f", ang, getActualKickPowerRate() );
    Log.log( 600, "update for kick: %f %f", soc.dPower, soc.dAngle );
  }

  posBall += velBall;
  velBall *= SS->getBallDecay();

  if( pos != NULL )
    *pos = posBall;
  if( vel != NULL )
    *vel = velBall;
}

/*! This method determines the global position of the object o after iCycles
    When the object is the ball, only the decay of the ball is taken into
    account. When the object is a player it
    is assumed that the player dashes with 'iDashPower' every cycle.
    \param o objectT of which global position will be predicted
    \param iCycles pos is predicted after this number of cycles
    \param iDashPower dash power that is used every cycle in dash (default 100)
    \param vel will be filled with predicted global velocity after iCycles
    \return predicted global position after iCycles. */
VecPosition WorldModel::predictPosAfterNrCycles( ObjectT o, double dCycles,
        int iDashPower, VecPosition *posIn, VecPosition *velIn, bool bUpdate )
{
  VecPosition vel = ( velIn == NULL ) ? getGlobalVelocity( o ) : *velIn ;
  VecPosition pos = ( posIn == NULL ) ? getGlobalPosition( o ) : *posIn ;

  if( o == OBJECT_BALL )
  {
    // get the speed and the distance it travels in iCycle's.
    // use this distance and direction it travels in, to calculate new pos
    // geom series is serie s=a+ar+..+ar^n...decay=r,iCycles=n,dSpeed=a
```

```
    double dDist = Geometry::getSumGeomSeries( vel.getMagnitude(),
                                       SS->getBallDecay(),
                                       dCycles);
    pos          += VecPosition( dDist, vel.getDirection(), POLAR );
    vel          *= pow( SS->getBallDecay(), dCycles );
   }
   else if( SoccerTypes::isKnownPlayer( o )  )
   {
    double      dDirection = 0.0; // used when no info about global body
    Stamina     stamina;          // used when object is agent

    if( getAgentObjectType() == o )
    {
     dDirection = getAgentGlobalBodyAngle();
     stamina    = getAgentStamina();
    }
    else if( getTimeGlobalAngles(o) > getCurrentTime() - 2 )
     dDirection = getGlobalBodyAngle(o);

    for( int i = 0; i < (int)dCycles ; i ++ )
     predictStateAfterDash( iDashPower, &pos, &vel, &stamina, dDirection, o );
   }

   if( posIn != NULL && bUpdate )
    *posIn = pos;
   if( velIn != NULL && bUpdate )
    *velIn = vel;

   return pos;
}

/*! This method predicts the position of the agent after 'iCycles' when every
    cycle is dashed with 'iDashPower'. The method
    'predictGlobalPosAfterNrCycles' is used to calculate this position.
    \param iCycles number of cycles
    \param iDashPower dash power that is passed
    \return VecPosition indicating predicted global position of agent. */
VecPosition WorldModel::predictAgentPos( int iCycles, int iDashPower )
{
 return predictPosAfterNrCycles( getAgentObjectType(), iCycles, iDashPower);
}

/*! This method predicts the final position of the agent when no commands
    are issued, but the agent just rolls out.
    \return VecPosition indicating predicted final global position of agent. */
VecPosition WorldModel::predictFinalAgentPos(VecPosition *pos,VecPosition *vel)
```

```
{
 VecPosition velAgent   = (vel==NULL) ? getAgentGlobalVelocity (): *vel;
 VecPosition posAgent   = (pos==NULL) ? getAgentGlobalPosition (): *pos;
 double      dDistExtra =
  Geometry::getSumInfGeomSeries(velAgent.getMagnitude(),SS->getPlayerDecay());
 return posAgent + VecPosition(dDistExtra,velAgent.getDirection(), POLAR );

}

/*! This method check how many cycles are needed for object 'o' to travel
    a distance 'dDist' when it currently has a speed 'dSpeed'. */
int WorldModel::predictNrCyclesForDistance ( ObjectT o, double dDist,
                                             double dSpeed )
{
 double dSpeedPrev = -1.0;
 int    iCycles    = 0;
 double dDecay     = getPlayerDecay( o );
 double dDashRate  = getDashPowerRate( o );
 double dMinDist   = getMaximalKickDist( o );

 // stop this loop when max speed is reached or the distance is traveled.
 while( dDist > dMinDist &&
        (fabs(dSpeed - dSpeedPrev) > EPSILON || dSpeed < 0.3 ) &&
        iCycles < 40 ) // ignore standing still and turning
 {
  dSpeedPrev = dSpeed;
  dSpeed    += SS->getMaxPower()*dDashRate;
  if( dSpeed > SS->getPlayerSpeedMax() )
    dSpeed = SS->getPlayerSpeedMax();
  dDist = max( 0, dDist - dSpeed );
  dSpeed *= dDecay;
  iCycles++;
 }
 dSpeed /= dDecay;

 // if distance not completely traveled yet, count the number of cycles to
 // travel the remaining distance with this speed.
 if( dDist > dMinDist  )
  iCycles += (int)ceil(( dDist - dMinDist )/dSpeed);
 return max(0, iCycles ) ;
}


/*! This method gives an estimate for the number of cycles a player needs to
    reach a specific position. A position is reached when the player is located
    in the maximal kick distance of this position. When this is not the case
```

```
    a dash (or turn) is performed until the player is in the kickable distance.
    \param o objectT which wants to reach posTo
    \param posTo global position which has to be reached
    \param angToTurn angle to 'posTo' when is turned (instead of dashed)
    \return predicted nr of cycles for o to reach posTo  */
int WorldModel::predictNrCyclesToPoint( ObjectT o, VecPosition posTo )
{
  char         strBuf[128];
  VecPosition   posGlobal = getGlobalPositionLastSee( o ), posPred;
  VecPosition   vel;
  int          iCycles;
  AngDeg       angBody, angNeck = 0, ang;
  AngDeg       angDes = (posTo-posGlobal).getDirection();
  SoccerCommand soc;

  Log.log( 460, "predict steps for %s with dist %f (time %d) and body %f (%d)",
           SoccerTypes::getObjectStr( strBuf, o ),
           posTo.getDistanceTo( posGlobal ),
           getTimeGlobalPositionLastSee( o ).getTime(),
           getGlobalBodyAngle(o), getTimeChangeInformation(o).getTime() );

  // if already in kickable distance, return 0
  if( posTo.getDistanceTo( posGlobal ) < getMaximalKickDist( o) )
  {
   Log.log( 460, "already close: 0" );
   return 0;
  }

  // first check how old the change info (and thus body and vel.) info is
  // if too old, assume information is perfect and set time to position info
  // otherwise update all information with stored information
  iCycles = getTimeChangeInformation(o).getTime() - getCurrentCycle();
  if( o == getAgentObjectType() )
  {
   angBody = getAgentGlobalBodyAngle();
   vel     = getAgentGlobalVelocity( );
   posPred = getAgentGlobalPosition( );
   iCycles = 0;
  }
  else if( iCycles < -3 )
  {
   angBody = angDes;
   vel.setVecPosition( 0.3, angDes, POLAR );
   if( SoccerTypes::isOpponent( o ) )
     iCycles = -2; // otherwise too optimistic
   else
```

```
    iCycles = 0;
  posPred = getGlobalPositionLastSee( o );
}
else
{
  angBody = getGlobalBodyAngleLastSee( o );
  vel     = getGlobalVelocityLastSee( o );
  posPred = getGlobalPositionLastSee( o );
}

Log.log( 460, "rel. time angle info (des %f,now %f,speed %f): %d (%d-%d)",
     angDes, angBody, vel.getMagnitude(), iCycles,
     getTimeChangeInformation(o).getTime(), getCurrentCycle() );

if( o != getAgentObjectType() &&
    getTimeGlobalPositionLastSee( o ) > getTimeChangeInformation(o) )
{
  Log.log( 460, "update cycles to global pos. time: %d",
             getTimeGlobalPositionLastSee( o ).getTime() );
  iCycles = max(iCycles,
             getTimeGlobalPositionLastSee(o).getTime()-getCurrentCycle());
}

soc = predictCommandToMoveToPos(o,posTo,1,2.5,false,&posPred,&vel,&angBody );
ang = VecPosition::normalizeAngle( angBody - angDes );

// sometimes we dash to stand still and turn then
while( soc.commandType == CMD_TURN ||
       ( fabs( ang ) > 20 && soc.commandType == CMD_DASH && soc.dPower < 0 ))
{
  iCycles++;
  predictStateAfterCommand( soc, &posPred, &vel, &angBody, &angNeck, o );
  if( posTo.getDistanceTo( posPred ) < getMaximalKickDist( o ) )
  {
    Log.log( 460, "reached point during turning, vel %f: %d",
                  vel.getMagnitude(), iCycles );
    return iCycles;
  }
  soc=predictCommandToMoveToPos(o,posTo,1,2.5,false,&posPred,&vel,&angBody );
  ang = VecPosition::normalizeAngle( angBody - angDes );
}
Log.log( 460, "cycles after turning: %d (ang %f, %f) vel %f",
         iCycles, ang, angDes, vel.getMagnitude() );

if( o != getAgentObjectType() )
{
```

```
    // iCycles++; // do not count last dash -> predictState not called
    double dVel = vel.rotate(-angBody).getX(); // get distance in direction
    iCycles += predictNrCyclesForDistance(o,posPred.getDistanceTo(posTo),dVel);
   }
   else
   {
    while( posPred.getDistanceTo( posTo ) > getMaximalKickDist( o ) )
    {
     soc=predictCommandToMoveToPos(o,posTo,1,2.5,0,&posPred,&vel,&angBody);
     predictStateAfterCommand( soc, &posPred, &vel, &angBody, &angNeck, o );
     iCycles++;
    }
   }

   Log.log( 460, "total cycles: %d", iCycles );
   return iCycles;
}

/*! This method returns the number of cycles it will take the object 'objFrom'
    to reach the object 'objTo' (usually respectively the player and the ball).
    \param objFrom ObjectT that is the object that wants to move
    \param objTo ObjectT to which is moved
    \return number of cycles it will take objFrom to move to objTo */
int WorldModel::predictNrCyclesToObject( ObjectT objFrom, ObjectT objTo )
{
   VecPosition posPrev(UnknownDoubleValue,UnknownDoubleValue);

   if( objFrom == OBJECT_ILLEGAL || objTo == OBJECT_ILLEGAL ||
       getGlobalPosition( objFrom ).getDistanceTo( getGlobalPosition( objTo )
       ) > 40 )
     return 101;

   // this is part of the intercept
   if( objFrom == getAgentObjectType() && objTo == OBJECT_BALL )
   {
     FeatureT  feature_type = FEATURE_INTERCEPT_CYCLES_ME;
     if( isFeatureRelevant( feature_type ) )
     {
       return max(0,
            ((int)getFeature( feature_type ).getInfo() - getCurrentCycle() ));
     }
     else
     {
       Log.log( 460, "create intercept features" );
       createInterceptFeatures( );
       Log.log( 460, "call predict again" );
```

```
      return predictNrCyclesToObject( objFrom, objTo );
    }
  }

  // in case of ball with no velocity, calculate cycles to point
  if( objTo == OBJECT_BALL && getBallSpeed() < 0.01 )
    return predictNrCyclesToPoint( objFrom, getBallPos() );

  int        iCycles      = 0;
  int        iCyclesToObj = 100;
  VecPosition posObj(0,0);

  // continue calculating number of cycles to position until or we can get
  // earlier at object position, are past maximum allowed number of cycles or
  // the object does not move anymore.
  while( iCycles <= iCyclesToObj && iCycles < PS->getPlayerWhenToIntercept() &&
         posObj.getDistanceTo( posPrev ) > EPSILON )
  {
    iCycles      = iCycles + 1  ;
    posPrev      = posObj;
    posObj       = predictPosAfterNrCycles( objTo,   iCycles );

    if(getGlobalPosition(objFrom).getDistanceTo(posObj)/SS->getPlayerSpeedMax()
            < iCycles + 1 )
    {
      Log.log( 460, "predictNrCyclesToPoint after %d cycles", iCycles );
      iCyclesToObj = predictNrCyclesToPoint ( objFrom, posObj );
    }
  }

  return iCyclesToObj;
}

/*! This method updates all the stamina variables using the calculations from
    the soccer manual. It is not really important since stamina is read from
    sense_body every cycle. That information is more up to date.
    \param power of last dash command
    \param stamina pointer to all stamina values, will change to new value
    \return stamina class will be updated to new stamina values */
void WorldModel::predictStaminaAfterDash( double dPower, Stamina *stamina )
{
  double sta = stamina->getStamina();
  double eff = stamina->getEffort();
  double rec = stamina->getRecovery();

  // double negative value when dashed backwards
```

```
  sta -= ( dPower > 0.0 ) ? dPower : -2*dPower ;
  if( sta < 0 ) sta = 0;

  // stamina below recovery threshold, lower recovery
  if( sta <= SS->getRecoverDecThr()*SS->getStaminaMax() &&
                    rec > SS->getRecoverMin() )
     rec -= SS->getRecoverDec();

  // stamina below effort decrease threshold, lower effort
  if( sta <= SS->getEffortDecThr()*SS->getStaminaMax() &&
                 eff > SS->getEffortMin() )
     eff -= SS->getEffortDec();

  // stamina higher than effort incr threshold, raise effort and check maximum
  if( sta >= SS->getEffortIncThr() * SS->getStaminaMax() &&
     eff < 1.0)
  {
    eff += SS->getEffortInc();
    if ( eff > 1.0 )
       eff = 1.0;
  }

  // increase stamina with (new) recovery value and check for maximum
  sta += rec*SS->getStaminaIncMax();
  if ( sta > SS->getStaminaMax() )
    sta = SS->getStaminaMax();

  stamina->setStamina ( sta );
  stamina->setEffort  ( eff );
  stamina->setRecovery( rec );
}

/*! This method returns the command for object 'obj' to turn towards a point
    'posTo' on the field when it has 'iCycles' to reach that point. If
    the point is within 'dDistBack' behind the object it will try to dash
    backwards. In the case that 'bMoveBack' is true, it will always try to
    move backwards. When posIn, velIn and angBodyIn are equal to NULL, the
    current agent information is used. */
SoccerCommand WorldModel::predictCommandTurnTowards( ObjectT obj, VecPosition
  posTo, int iCycles, double dDistBack, bool bMoveBack,
  VecPosition *posIn, VecPosition *velIn, AngDeg *angBodyIn )
{
  SoccerCommand soc, socFirst;
  VecPosition   pos, vel;
  AngDeg        angBody, ang, angNeck, angTo;
  Stamina       sta;
```

```
bool         bFirst = true;

// fill in all values
angBody = ( angBodyIn == NULL ) ? getGlobalBodyAngle( obj ) : *angBodyIn;
pos     = ( posIn     == NULL ) ? getGlobalPosition ( obj ) : *posIn;
vel     = ( velIn     == NULL ) ? getGlobalVelocity ( obj ) : *velIn;
angNeck = getGlobalNeckAngle( obj );

// predict where we will finally stand when our current vel is propogated
// and then check the orthogonal distance w.r.t. our body direction
VecPosition posPred=predictPosAfterNrCycles( obj, min(iCycles,4),
                                  0, &pos, &vel, false );
Line       line   =Line::makeLineFromPositionAndAngle( posPred, angBody );
double     dDist  =line.getDistanceWithPoint( posTo );

// get the angle to this point
angTo = (posTo - posPred).getDirection();
angTo = VecPosition::normalizeAngle( angTo - angBody );

// determine whether we want to turn based on orthogonal distance
double dRatioTurn;
if( pos.getDistanceTo(posTo) > 30.0 )
  dRatioTurn = 4.0;
if( pos.getDistanceTo(posTo) > 20.0 )
  dRatioTurn = 3.0;
else if( pos.getDistanceTo(posTo) > 10 )
  dRatioTurn = 2.0;
else
  dRatioTurn = 0.90 ;

AngDeg angTmp = angTo + (bMoveBack) ? 180 : 0;
angTmp = VecPosition::normalizeAngle( angTmp );

// turn when:
//  1. point lies outside our body range (forward and backwards)
//  2. point lies outside distBack and behind us (forward move)
//  3. point lies outside distBack and in front of us backwards move)
int    turn = 0;
while( ( dDist > dRatioTurn*getMaximalKickDist( obj ) ||
       ( posPred.getDistanceTo( posTo ) > dDistBack &&
          ( ( fabs( angTo ) > 90 && bMoveBack == false  ) ||
            ( fabs( angTo ) < 90 && bMoveBack == true ) ) ) )
      && turn < 5 && fabs( angTmp ) > PS->getPlayerWhenToTurnAngle() )
{

   ang = (posTo - posPred).getDirection() + (( bMoveBack == true )?180:0);
```

```
    ang = VecPosition::normalizeAngle( ang - angBody );
    soc = SoccerCommand(CMD_TURN,getAngleForTurn(ang,vel.getMagnitude(),obj));
    Log.log( 468, "angTo %f, dDist %f, ang %f %d angBody %f soc %f vel %f %f",
            angTo, dDist, ang, obj, angBody, soc.dAngle, vel.getMagnitude(),
            getInertiaMoment( obj ));
    if( bFirst == true )
      socFirst = soc;
    bFirst = false;
    predictStateAfterTurn(soc.dAngle, &pos, &vel, &angBody,&angNeck,obj,&sta);
    line = Line::makeLineFromPositionAndAngle( posPred, angBody );
    dDist = line.getDistanceWithPoint( posTo );
    angTo = (posTo - posPred).getDirection();
    angTo = VecPosition::normalizeAngle( angTo - angBody );
    turn++;
  }

  // if very close and have to turn a lot, it may be better to move with our
  // back to that point
  if( turn > 1 && iCycles < 4 && posPred.getDistanceTo( posTo ) < dDistBack &&
      bMoveBack == false)
  {
    angBody = ( angBodyIn == NULL ) ? getGlobalBodyAngle( obj ) : *angBodyIn;
    pos     = ( posIn     == NULL ) ? getGlobalPosition ( obj ) : *posIn;
    vel     = ( velIn     == NULL ) ? getGlobalVelocity ( obj ) : *velIn;
    ang = (posTo - posPred).getDirection() + 180;
    ang = VecPosition::normalizeAngle( ang - angBody );
    soc = SoccerCommand(CMD_TURN,getAngleForTurn(ang,vel.getMagnitude(),obj));
    predictStateAfterTurn( soc.dAngle,&pos,&vel,&angBody,&angNeck,obj,&sta);
    line = Line::makeLineFromPositionAndAngle( posPred, angBody );
    dDist = line.getDistanceWithPoint( posTo );
    if( dDist < 0.9*getMaximalKickDist( obj )  )
    {
      Log.log( 463, "turn around and intercept with back" );
      return soc;
    }
  }

  return socFirst;
}

/*! This method returns the command to move to a position, first it checks
    whether a turn is necessary. When this is the case, it performs the turn.
    Otherwise a dash command is generated to move in 'iCycles' cycles to
    the point 'posTo'. */
SoccerCommand WorldModel::predictCommandToMoveToPos( ObjectT obj,
  VecPosition posTo, int iCycles, double dDistBack,
```

```
  bool bMoveBack,VecPosition *posIn, VecPosition *velIn, AngDeg *angBodyIn)
{
  VecPosition   pos, vel;
  AngDeg        angBody;
  SoccerCommand soc;
  double        dPower;

  // fill in all values
  angBody = ( angBodyIn == NULL ) ? getGlobalBodyAngle( obj ) : *angBodyIn;
  pos     = ( posIn     == NULL ) ? getGlobalPosition ( obj ) : *posIn;
  vel     = ( velIn     == NULL ) ? getGlobalVelocity ( obj ) : *velIn;

  soc = predictCommandTurnTowards(obj, posTo, iCycles, dDistBack, bMoveBack,
                                  posIn, velIn,   angBodyIn);
  if( ! soc.isIllegal() )
    return soc;

  dPower = getPowerForDash( posTo-pos, angBody, vel,getAgentEffort(),iCycles );
  return SoccerCommand( CMD_DASH, dPower );
}

/*! This command returns the command for object 'obj' to intercept the ball.
    It needs the command 'socClose' as the command to intercept a close ball
    (may be CMD_ILLEGAL). 'iCycles' will be filled with the number of cycles
    to get to this ball position and 'posIntercept' will be filled with the
    final interception point. When posIn, velInn, angBody are equal to NULL,
    the agent information is used in the calculations. */
SoccerCommand WorldModel::predictCommandToInterceptBall( ObjectT obj,
    SoccerCommand socClose, int *iCycles, VecPosition *posIntercept,
    VecPosition *posIn, VecPosition *velIn, AngDeg *angBodyIn )
{
  FeatureT feature_type = FEATURE_INTERCEPTION_POINT_BALL;

  // check whether we already have calculated this value
  if( isFeatureRelevant( feature_type ) )
  {
    int i = max(0,((int)getFeature(feature_type).getInfo()-getCurrentCycle()));
    if( iCycles != NULL )
      *iCycles = i;
    if( posIntercept != NULL )
      *posIntercept = predictPosAfterNrCycles( OBJECT_BALL, i );

    Log.log( 463, "intercept, use old info, feature %d: %d", feature_type,
        max(0,((int)getFeature( feature_type ).getInfo()-getCurrentCycle())));
    return getFeature( feature_type ).getCommand();
  }
```

```
// declare all needed variables
SoccerCommand soc;
VecPosition   pos, vel, posPred, posBall(0,0), posBallTmp, velBall, posAgent;
AngDeg        angBody, angNeck;
int           iMinCyclesBall=100, iFirstBall=100;
double        dMaxDist = getMaximalKickDist( obj );
double        dBestX = UnknownDoubleValue;
Stamina       sta;
double        dMinOldIntercept = 100, dDistanceOfIntercept = 10.0;
int           iOldIntercept = UnknownIntValue;
static Time   timeLastIntercepted(-1,0);
static VecPosition posOldIntercept;

// didn't intercept ball in last two cycles -> reset old interception point
if( (getCurrentTime() - timeLastIntercepted) > 2 )
  posOldIntercept.setVecPosition( UnknownDoubleValue, UnknownDoubleValue);
timeLastIntercepted = getCurrentTime();

int iCyclesBall = 0;

Log.log( 468, "old interception point: (%f,%f)", posOldIntercept.getX(),
 posOldIntercept.getY() );

// for each new pos of the ball, check whether agent can reach ball
// and update the best interception point
while( iCyclesBall <= PS->getPlayerWhenToIntercept() &&
       iCyclesBall <= iFirstBall + 20 &&
       isInField( posBall ) == true )
{
  // re-initialize all variables
  angBody = ( angBodyIn == NULL ) ? getGlobalBodyAngle( obj ) : *angBodyIn;
  angNeck = getGlobalNeckAngle( obj );
  pos     = ( posIn     == NULL ) ? getGlobalPosition ( obj ) : *posIn;
  vel     = ( velIn     == NULL ) ? getGlobalVelocity ( obj ) : *velIn;
  sta     = getAgentStamina();
  soc.commandType = CMD_ILLEGAL;

  // predict the ball position after iCycles and from that its velocity
  posBallTmp = predictPosAfterNrCycles( OBJECT_BALL, iCyclesBall );
  if( iCyclesBall == 0 )
    velBall = getGlobalVelocity( OBJECT_BALL );
  else
    velBall = posBallTmp - posBall;
  posBall   = posBallTmp;
```

```
// predict the agent position
posPred   = predictPosAfterNrCycles( obj, min(iCyclesBall,4), 0 );
posAgent  = getGlobalPosition( obj );

// if too far away, we can never reach it and try next cycle
if( posPred.getDistanceTo(posBall)/getPlayerSpeedMax( obj )
     > iCyclesBall + dMaxDist || isInField( posBall ) == false )
{
  iCyclesBall++;
  continue;
}

// predict our position after the same nr of cycles when intercepting
for( int i = 0; i < iCyclesBall; i++ )
{
  soc = predictCommandToMoveToPos( obj, posBall, iCyclesBall - i ,
        2.5, false, &pos, &vel, &angBody );
  predictStateAfterCommand( soc, &pos, &vel, &angBody, &angNeck, obj );
}

// if in kickable distance, we can reach the ball!
if (pos.getDistanceTo( posBall ) < dMaxDist  )
{
  Log.log( 468, "can intercept ball in %d cycles, dist %f, old %f obj %d",
          iCyclesBall, pos.getDistanceTo( posBall ),
          posBall.getDistanceTo( posOldIntercept ), obj  );

  if( iMinCyclesBall == 100 ) // log first possible interception point
    iFirstBall = iMinCyclesBall = iCyclesBall;

  // too get some consistency in the interception point and avoid
  // too many turns, also keep track of the current possible
  // interception point.  This is the point close to the old
  // interception point. Two constraints are that the ball has to
  // have some speed (else it does not really matter where to
  // intercept) and the ball must be intercepted safely, that is
  // the ball is close to the body when intercepting.
  if( posBall.getDistanceTo( posOldIntercept ) <
                    min( 1.0, dMinOldIntercept ) &&
      pos.getDistanceTo( posBall ) < 0.70*getMaximalKickDist( obj ) &&
      velBall.getMagnitude() > 0.6 )
  {
    Log.log( 468, "update old interception point %d", iCyclesBall );
    dBestX           = posBall.getX();
    iOldIntercept    = iCyclesBall;
    dDistanceOfIntercept = pos.getDistanceTo( posBall );
```

```
      dMinOldIntercept = posBall.getDistanceTo( posOldIntercept );
    }
    // determine the safest interception point. This point must be
    // better than the current intercept, the distance to ball must
    // be very small after interception and close to the previous
    // calculated interception point
    else if( pos.getDistanceTo( posBall ) < dDistanceOfIntercept &&
             dDistanceOfIntercept > 0.50*getMaximalKickDist( obj ) &&
             ( iCyclesBall <= iMinCyclesBall + 3  ||
               iCyclesBall <= iOldIntercept + 3 ) &&
             fabs( posBall.getY() ) < 32.0 &&
             fabs( posBall.getX() ) < 50.0 )
    {
      iMinCyclesBall = iCyclesBall;
      dDistanceOfIntercept = pos.getDistanceTo( posBall );
      Log.log( 468, "safer interception at %d", iMinCyclesBall );
      if( iOldIntercept == iMinCyclesBall - 1 )
      {
        Log.log( 468, "old interception point -> safer" );
        iOldIntercept = iMinCyclesBall;
      }
    }
  }
  else
    Log.log( 468, "cannot intercept ball in %d cycles, dist %f, %f and %f",
      iCyclesBall, pos.getDistanceTo(posBall), pos.getDistanceTo( posAgent ),
       posBall.getDistanceTo( posAgent ) - dMaxDist);;

  iCyclesBall++;
}

Log.log( 463, "first interception point:        %d cycles", iFirstBall );
Log.log( 463, "best interception point:         %d cycles", iMinCyclesBall );
Log.log( 463, "old interception point           %d cycles", iOldIntercept );

// check special situations where we move to special position.
if( !( iMinCyclesBall > iOldIntercept + 2 ) &&
    iOldIntercept != UnknownIntValue  )
{
  Log.log( 463, "move to old interception point." );
  iMinCyclesBall = iOldIntercept;
}
else
{
  Log.log( 463, "move to first intercept" );
  iMinCyclesBall = iFirstBall;
```

```
  }

  posBall = predictPosAfterNrCycles( OBJECT_BALL, iMinCyclesBall );
  Log.log( 463, "choose %d cycles", iMinCyclesBall );
  logCircle( 463, posBall, 1.0 );
  if( iCycles != NULL )
    *iCycles = iMinCyclesBall;

  posOldIntercept = posBall;
  posPred = predictPosAfterNrCycles( obj, min(iMinCyclesBall,4), 0 );
  if( posIntercept != NULL )
    *posIntercept = posBall;

  if( iMinCyclesBall < 3 && ! socClose.isIllegal() )
  {
    Log.log( 463, "do close intercept" );
    iMinCyclesBall = 1;
    soc = socClose;
  }
  else if( posPred.getDistanceTo( posBall ) < 0.5 )
  {
    Log.log( 463, "intercept: do not move already close" );
    soc = SoccerCommand( CMD_ILLEGAL );
  }
  else
  {
    Log.log( 463, "intercept: move to (%f,%f)", posBall.getX(),posBall.getY());
    Log.log( 560, "intercept: move to (%f,%f) in %d cycles",
          posBall.getX(),posBall.getY(), iMinCyclesBall);
    if( isDeadBallUs() && !isGoalKickUs())     // do not dash backwards
      soc = predictCommandToMoveToPos( obj, posBall, iMinCyclesBall, 0 );
    else
      soc = predictCommandToMoveToPos( obj, posBall, iMinCyclesBall );
  }

  // store the calculated action as a feature
  if( obj == getAgentObjectType() )
    setFeature( feature_type,
            Feature( getTimeLastSeeMessage(),
                  getTimeLastSenseMessage(),
                  getTimeLastHearMessage(),
                  OBJECT_ILLEGAL,
                  getTimeLastSeeMessage().getTime() + iMinCyclesBall,
                  soc ) );
  return soc;
}
```

```
/*! This method determines whether a dash command (supplied as the first
    argument) will result in collision with another player.
    This is checked by determing the global position after the command
    and then check whether the positions of one of the other players lies
    with the player size. Since it cannot be known what kind of action the
    other player takes in this cycle, it is also difficult to predict what the
    global position of the player will be in the next cycle. This method
    therefore assumes the other players have issued a dash with maximum power
    in the last cycle.
    \return bool indicating whether dash will result in a collision. */
bool WorldModel::isCollisionAfterCommand( SoccerCommand soc )
{
  VecPosition posPred, velPred;
  AngDeg ang1, ang2;
  Stamina sta;

  predictAgentStateAfterCommand( soc, &posPred, &velPred, &ang1,&ang2,&sta );
  velPred /= SS->getPlayerDecay();
  VecPosition posBall = predictPosAfterNrCycles( OBJECT_BALL, 1 );
  if( soc.commandType == CMD_KICK )
    predictBallInfoAfterCommand( soc, &posBall );
  double  dDist   = posPred.getDistanceTo( posBall ) -
                    SS->getPlayerSize() - SS->getBallSize();
  Log.log( 510, "check collision dist %f, noise_ball %f noise_me %f",
          dDist, getBallSpeed()*SS->getBallRand(),
          velPred.getMagnitude()*SS->getPlayerRand() );

  // we could also take into account the error in player movement, but this
  // is very large, so we would in many cases get a dash
  if( dDist < getBallSpeed()*SS->getBallRand() )
    return true;

  return false;
}
```

```
#include "WorldModel.h"
#include "Parse.h"
#include <stdio.h>      // needed for sprintf
#include <list>         // needed for list

#include <sys/times.h>  // needed for times
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>


/*****************************************************************************
/
/********************* CLASS WORLDMODEL
*************************************/
/*****************************************************************************
/


/*! This method is called when new visual information about objects on
    the field is received from the see_global message (only the coach
    receives these messages). This method updates the information of the
    Object that corresponds to the ObjectT that is supplied as the first
    argument.
    \param o objectType that should be updated
    \param time time of the corresponding values
    \param pos global position of this object
    \param vel global velocity of this object
    \param angBody body angle of this object
    \param angNeck neck angle of this object */
void WorldModel::processSeeGlobalInfo( ObjectT o, Time time,
     VecPosition pos, VecPosition vel, AngDeg angBody, AngDeg angNeck)
{
  DynamicObject * dobj;
  PlayerObject * pobj;

  if( o == OBJECT_ILLEGAL )
    return;
  if( SoccerTypes::isPlayer( o ) )
  {
    pobj = (PlayerObject*)getObjectPtrFromType( o );
    pobj->setTimeLastSeen( time );
    pobj->setGlobalPositionLastSee( pos, time );
    pobj->setTimeChangeInformation( time );
    pobj->setGlobalPosition( pos, time );
    pobj->setGlobalVelocity( vel, time );
```

```
    pobj->setGlobalBodyAngle( angBody, time );
    pobj->setGlobalNeckAngle( VecPosition::normalizeAngle(angBody+angNeck),
                                                time );
    pobj->setIsKnownPlayer( true );

  }
  else if( SoccerTypes::isBall( o ) )
  {
    dobj = (DynamicObject*)getObjectPtrFromType( o );
    dobj->setTimeLastSeen( time );
    dobj->setGlobalPosition( pos, time );
    dobj->setGlobalVelocity( vel, time );
  }
}

/*! This method is called when new visual information about the agent
    is received. It updates the information of the AgentObject stored in
    the WorldModel.
    \param vq new ViewQuality for the agent
    \param va new ViewAngle for the agent
    \param dStamina new stamina for the agent
    \param dEffort new effort for the agent
    \param dSpeed magnitude of the velocity
    \param angSpeed angle of the speed velocity (relative to neck)
    \param angHeadAngle new relative angle between body and neck
    \param iTackleExpire number of cycles before tackle expires
    \param iArmMovable indicates whether arm is movable
    \param iArmExpires number of cycles before arm expires
    \param posArm relative position to which arm is pointed
    \return bool indicating whether update was succesful */
bool WorldModel::processNewAgentInfo( ViewQualityT vq, ViewAngleT va,
     double dStamina, double dEffort, double dSpeed, AngDeg angSpeed,
     AngDeg angHeadAngle, int iTackleExpires, int iArmMovable,
     int iArmExpires, VecPosition posArm )
{
  Stamina sta = agentObject.getStamina();

  sta.setStamina                ( dStamina                        );
  sta.setEffort                 ( dEffort                         );
  agentObject.setStamina        ( sta                             );
  // this is already done when change_view is sent, and thus updated to the
  // predicted view angle, quality in the next cycle...
  if( vq == VQ_ILLEGAL )
    agentObject.setViewQuality    ( vq                            );
  if( va == VA_ILLEGAL )
    agentObject.setViewAngle      ( va                            );
```

```
  agentObject.setSpeedRelToNeck    ( VecPosition( dSpeed, angSpeed, POLAR) );
  agentObject.setBodyAngleRelToNeck( angHeadAngle                          );
  agentObject.setTackleExpires     ( iTackleExpires                        );
  agentObject.setArmMovable        ( iArmMovable == 0                      );
  agentObject.setArmExpires        ( iArmExpires                           );
  agentObject.setGlobalArmPosition ( getAgentGlobalPosition() + posArm     );

  return true;
}

/*! This method is called when new visual information about other
    objects on the field is received. It updates the information of
    the Object that corresponds to the ObjectT that is supplied as the
    first argument. If some of the arguments are not received with the
    visual information, they are passed with the value
    'UnknownDoubleValue'. Note that the objects are only updated with
    the (relative) information passed as arguments. To make sure all
    global information in the objects is synchronized with the
    relative information, the method updateAll() has to be called
    after all (relative) object information is updated.

    \param o objectType that should be updated
    \param time time of the corresponding values
    \param dDist distance from agent to this object o
    \param iDir angle from agent with this object o (relative to neck)
    \param dDistChange distance change between agent and this object o
    \param dDirChange angle change between agent and this object o
    \param angRelBodyAng angle between neck of agent and body of object o
    \param angRelNeckAng angle between neck of agent and neck of object o
    \param isGoalie bool indicating whether object is a goalie
    \param dPointDir pointing direction of arm of observed player
    \param isTackling bool indicating whether object is tackling
    \return bool indicating whether update was succesful */
void WorldModel::processNewObjectInfo( ObjectT o, Time time,
      double dDist, int iDir, double dDistChange, double dDirChange,
      AngDeg angRelBodyAng,   AngDeg angRelNeckAng, bool isGoalie,
      ObjectT objMin, ObjectT objMax, double dPointDir,  bool isTackling )
{
  if( dDist == UnknownDoubleValue || o == OBJECT_ILLEGAL )
    return; // no sense to update when only angle is known.

  if( SoccerTypes::isFlag( o ) )
  {

    Flags[SoccerTypes::getIndex(o)].setRelativePosition(
                                        dDist,(double)iDir,time);
```

```
 Flags[SoccerTypes::getIndex(o)].setTimeLastSeen    ( time              );
 Flags[SoccerTypes::getIndex(o)].setType            ( o                 );
}
else if( SoccerTypes::isPlayer( o ) || SoccerTypes::isBall( o ) )
{
 DynamicObject *d ;

 // if we do not have all information, update UnknownPlayer array
 if( !( SoccerTypes::isKnownPlayer( o ) || SoccerTypes::isBall( o ) ) )
 {
  UnknownPlayers[iNrUnknownPlayers].setIsKnownPlayer( false );
  UnknownPlayers[iNrUnknownPlayers].setPossibleRange( objMin, objMax );
  d = &UnknownPlayers[iNrUnknownPlayers];
  iNrUnknownPlayers++;
 }
 else // else update the known object (teammate, opponent, ball)
 {
  d = (DynamicObject*)getObjectPtrFromType( o );
  if( SoccerTypes::isPlayer( o ) )
    ((PlayerObject*)d)->setIsKnownPlayer( true );
 }

 if( d != NULL )  // if object was known
 {
  // set all values for this dynamicobject
  d->setRelativePosition( dDist, (double)iDir, time );
  if( dDistChange != UnknownDoubleValue )
    d->setRelativeDistanceChange( dDistChange, time );
  if( dDirChange != UnknownDoubleValue )
    d->setRelativeAngleChange( dDirChange, time );
  if( angRelBodyAng != UnknownAngleValue )
    ((PlayerObject*)d)->setRelativeBodyAngle( angRelBodyAng, time );
  if( angRelNeckAng != UnknownAngleValue )
    ((PlayerObject*)d)->setRelativeNeckAngle( angRelNeckAng, time );
  if( isGoalie == true && SoccerTypes::isPlayer( o ))
    ((PlayerObject*)d)->setIsGoalie( true );
  else if( SoccerTypes::isPlayer( o ))
    ((PlayerObject*)d)->setIsGoalie( false );
  d->setType( o );
  d->setTimeLastSeen( time );

  if( isTackling )
  {
    // if last observed tackle time has been passed.
    if( ((PlayerObject*)d)->getTimeTackle() + SS->getTackleCycles()
                                < getCurrentTime() )
```

```
    ((PlayerObject*)d)->setTimeTackle( getCurrentTime() - 1  );
  }
  else
    ((PlayerObject*)d)->setTimeTackle( Time(-1,0)  );

  if( dPointDir != UnknownDoubleValue )
   ((PlayerObject*)d)->setGlobalArm( dPointDir, getCurrentTime() );

  // check if there wasn't an unknown player located in the worldmodel
  // that corresponded to the same player
  if( SoccerTypes::isPlayer( o ) && SoccerTypes::isKnownPlayer( o )  )
  {
   int        iIndex;
   ObjectT    objMin = OBJECT_ILLEGAL;
   double     dMinDist = 1000.0, dTmp;
   ObjectSetT set = SoccerTypes::isOpponent( o )
                    ? OBJECT_SET_OPPONENTS
                    : OBJECT_SET_TEAMMATES;

   for( ObjectT obj = iterateObjectStart( iIndex, set );
        obj != OBJECT_ILLEGAL;
        obj = iterateObjectNext ( iIndex, set ) )
   {
    if( obj != getAgentObjectType() && isKnownPlayer( obj ) == false )
    {
     dTmp=(getRelativePosition(obj)-d->getRelativePosition()
                                          ).getMagnitude();
     if( dTmp < dMinDist )
     {
      objMin   = obj;
      dMinDist = dTmp;
     }
    }
   }
   iterateObjectDone( iIndex );
   if( objMin != OBJECT_ILLEGAL &&
      dMinDist < PS->getPlayerDistTolerance() &&
      dMinDist < getMaxTraveledDistance( objMin )  )
   {
    PlayerObject *pob = (PlayerObject*) getObjectPtrFromType( objMin );
    pob->setTimeLastSeen( -1 ); // delete the unknown player
    Log.log( 464, "set time objMin %d to -1 mapped to  player %d %f %f",
             objMin, o, dMinDist, getMaxTraveledDistance( objMin )  );
   }
   else
    Log.log( 464, "don't set time objMin %d to -1  player %d dist %f",
```

```
                objMin, o, dMinDist  );

      }
    }
  }
  else if( SoccerTypes::isLine( o ) )
  {
    // angle returned is angle of neck angle with line, convert to angle
    // of neck with orthogonal to line
    iDir = ( iDir < 0 ) ? (90 + iDir ) : - (90 - iDir );

    Lines[SoccerTypes::getIndex(o)].setRelativePosition(
                                        dDist,(double)iDir,time);
    Lines[SoccerTypes::getIndex(o)].setTimeLastSeen( time );
    Lines[SoccerTypes::getIndex(o)].setType( o );
  }
}

/*! This method stores a message communicated by an other player in the world
    model. It is not processed yet. This is done by the updateAll method.
    \param iPlayer player who communicated
    \param strMsg string that was communicated
    \param iDiff difference between cycles message was communicated and current
    time
    \return bool indicating whether all values were stored successfully. */
bool WorldModel::storePlayerMessage( int iPlayer, char *strMsg, int iCycle )
{
  strcpy( m_strPlayerMsg, strMsg );
  m_iCycleInMsg = iCycle;
  m_timePlayerMsg  = getCurrentTime();
  m_iMessageSender = iPlayer;
  return true;
}

/*! This method processes a communication message from a teammate. */
bool WorldModel::processPlayerMessage( )
{
  static char strMessage[MAX_MSG];                    // location for message
  int         iDiff = getCurrentCycle() - m_iCycleInMsg;// time difference
  double      dDiff = (double)iDiff/100.0;              // conf difference
  char        *strMsg;
  strcpy( strMessage, m_strPlayerMsg );
  strMsg = strMessage;                              // pointer to work with

  char cOffside = strMsg[1];                       // read offside information
  if( cOffside >= 'a' && cOffside <= 'z' )         // a-z corresponds to 0-25
```

```
{
  m_dCommOffsideX    = (double)(cOffside - 'a');
  m_timeCommOffsideX = getCurrentTime() - 1;
}
else if( cOffside >= 'A' && cOffside <= 'Z' )     // A-Z corresponds to 26-52
{
  m_dCommOffsideX = 26 + (double)(cOffside - 'A');
  m_timeCommOffsideX = getCurrentTime() - 1;
}
else                                    // wrong message
  return false;

if( strMsg[2] >= '0' && strMsg[2] <= '9' &&       // received ball info
    strlen( strMessage ) == 12)
{
  // translate ball position nd velocity back to initial range.
  double dBallX =    (int)(strMsg[2]-'0')*10 +(int)(strMsg[3] - '0' ) - 48.0;
  double dBallY =    (int)(strMsg[4]-'0')*10 +(int)(strMsg[5] - '0' ) - 34.0;
  double dBallVelX = (int)(strMsg[6]-'0') + (int)(strMsg[7] - '0' )/10.0-2.7;
  double dBallVelY = (int)(strMsg[8]-'0') + (int)(strMsg[9] - '0' )/10.0-2.7;

  VecPosition pos( dBallX, dBallY );
  VecPosition vel( dBallVelX, dBallVelY );
  for( int i = 0; i < iDiff ; i ++ )
  {
    pos += vel;
    vel *= SS->getBallDecay();
  }

  // if ball not seen or felt for three cycles, update ball information
  if( getTimeLastSeen( OBJECT_BALL ) == -1 ||
      (
        getTimeChangeInformation( OBJECT_BALL ) < getCurrentTime() - 3 &&
        getRelativeDistance( OBJECT_BALL ) > SS->getVisibleDistance()
       ) ||
      (
        getTimeChangeInformation( OBJECT_BALL ) < getCurrentTime() - iDiff &&
        vel.getDistanceTo( getGlobalVelocity(OBJECT_BALL) ) > 0.3 &&
        getRelativeDistance( OBJECT_BALL ) > SS->getVisibleDistance()
      )
    )
  {
    Log.log( 600,
      "update ball comm. (%1.2f,%1.2f)(%1.2f,%1.2f) diff %d, last %d",
          pos.getX(), pos.getY(), vel.getX(), vel.getY(), iDiff,
          getTimeLastSeen( OBJECT_BALL ).getTime() );
```

```
      Log.log( 601, "update ball from comm (%1.2f,%1.2f)(%1.2f,%1.2f) diff %d",
            pos.getX(), pos.getY(), vel.getX(), vel.getY(), iDiff );
      processPerfectHearInfoBall( pos, vel, 1.00 - dDiff - 0.01 )    ;
     }
     else
      Log.log( 600, "do not update ball time_change %d, now %d, diff %d, d %f",
        getTimeChangeInformation( OBJECT_BALL ).getTime(),
        getCurrentCycle(),
        iDiff,
        vel.getDistanceTo( getGlobalVelocity(OBJECT_BALL) ) );
    }
   else if( strlen( strMessage ) == 12 &&            // received attacker info
          strMsg[2] >= 'a' && strMsg[2] <= 'a' + 10 &&
          strMsg[6] == ' ' )
   {
    ObjectT objOpp
           = SoccerTypes::getOpponentObjectFromIndex((int)(strMsg[2]-'a'));
    char   *str  = &strMsg[3];                    // get pointer to string
    double dOppX = -1*Parse::parseFirstInt( &str );
    dOppX /= 10.0;
    double dOppY = Parse::parseFirstInt( &str );
    dOppY /= 10.0;
    VecPosition posOpp( dOppX, dOppY );
    processPerfectHearInfo( objOpp, posOpp, 0.99, false )  ;
   }

   return true;
}

bool WorldModel::processRecvThink( bool b )
{
  m_bRecvThink = b;
  if( b == true && SS->getSynchMode() == true )
  {
#ifdef WIN32
    //EnterCriticalSection( &mutex_newInfo );
    bNewInfo           = true;
    SetEvent           (  event_newInfo );
    //LeaveCriticalSection( &mutex_newInfo );
#else
    pthread_mutex_lock  ( &mutex_newInfo );
    bNewInfo           = true;
    pthread_cond_signal ( &cond_newInfo );
    pthread_mutex_unlock( &mutex_newInfo );
#endif
  }
```

```
  return true;
}

/*! This method is called when new information about the ball is heard.
    It updates the information of the ball only when the confidence is higher
    than the information already available in the WorldModel.
    \param posGlobal global position of the ball
    \param vel global velocity of the ball
    \param dConf confidence in the ball information
    \return true when information is updated, false otherwise */
bool WorldModel::processPerfectHearInfoBall( VecPosition posGlobal,
                        VecPosition vel, double dConf )
{
  Log.log( 501, "ball conf: %f %d", getConfidence( OBJECT_BALL ),
    getTimeLastSeen( OBJECT_BALL ).getTime() );
  if( Ball.getConfidence( getCurrentTime() ) < dConf ||
      vel.getDistanceTo( getGlobalVelocity(OBJECT_BALL) ) > 0.3  )
  {
    Time time = getTimeFromConfidence( dConf );
    Ball.setGlobalPosition( posGlobal, time );
    Ball.setGlobalVelocity( vel,       time );
    Ball.setTimeLastSeen  (            time );
    updateObjectRelativeFromGlobal( OBJECT_BALL );
    setTimeLastHearMessage( getCurrentTime() );
    return true;
  }
  return false;
}

/*! This method is called when new information about an object is heard. But
    only when the player who said the message was completely sure about the
    object type (it would not give information about a player it does not know
    the player number of, then it would call processUnsureHearInfo).
    It updates the information of the specified object only when the confidence
    is higher than the information already available in the WorldModel.
    \param o object type to which this information relates to
    \param posGlobal global position of this object type.
    \param dConf confidence in the object information.
    \param bIsGoalie boolean indicating whether 'o' is a goalie
    \return true when information is updated, false otherwise */
bool WorldModel::processPerfectHearInfo( ObjectT o, VecPosition posGlobal,
                                double dConf, bool bIsGoalie )
{
  if( SoccerTypes::isBall( o ) || o == getAgentObjectType() )
    return false; // ball should be called with processPerfectHearInfoBall
  else if( !SoccerTypes::isKnownPlayer( o ) )
```

```
    return processUnsureHearInfo( o, posGlobal, dConf );

  PlayerObject *object = (PlayerObject *)getObjectPtrFromType( o );
  if( object == NULL )
    return false;

  Time time = getTimeFromConfidence( dConf ) ;

  // if we are not sure about the exact player number of this player in
  // the world model (getIsKnownPlayer() == false) we overwrite the
  // information of this player since the player who said this information
  // is sure about it (otherwise processUnsureHearInfo would be called instead
  // of processPERFECTHearInfo)
  if( object->getConfidence( getCurrentTime() ) < dConf ||
      object->getIsKnownPlayer() == false  )
  {
    object->setGlobalPosition     ( posGlobal         , time );
    object->setTimeLastSeen       ( time                     );
    object->setGlobalVelocity     ( VecPosition( 0, 0), time );
    object->setIsKnownPlayer      ( true                     );
    object->setIsGoalie           ( bIsGoalie                );
    updateObjectRelativeFromGlobal( o                        );
    setTimeLastHearMessage( getCurrentTime() );
    return true;
  }
  return false;
}

/*! This method is called when new information about an object is heard. But
    only when the player who said the message was not sure about the
    object type. It is tried to map the given information to an object already
    in the WorldModel. If we cannot find such a player, we add the player info
    at the position of the first player who we don't have information about.
    The information of the specified object is only updated when the confidence
    is higher than the information already available in the WorldModel.
    \param o object type to which this information relates to
    \param pos global position of this object type.
    \param dConf confidence in the object information.
    \return true when information is updated, false otherwise */
bool WorldModel::processUnsureHearInfo( ObjectT o, VecPosition pos,
                                        double dConf )
{
  double     dMinDist;       // used to find closest player to pos
  ObjectT    objInitial = o;

  if( o != OBJECT_TEAMMATE_UNKNOWN && o != OBJECT_OPPONENT_UNKNOWN )
```

```
    return false;

  // if o is a teammate find closest teammate to pos and store distance
  if( SoccerTypes::isTeammate( o ) )
    o = getClosestInSetTo( OBJECT_SET_TEAMMATES, pos, &dMinDist);
  else if( SoccerTypes::isOpponent( o ) )  // if o is an opponent, do the same
    o = getClosestInSetTo( OBJECT_SET_OPPONENTS, pos, &dMinDist);

  if( o == getAgentObjectType() &&
     pos.getDistanceTo(getAgentGlobalPosition())<PS->getPlayerDistTolerance())
    return false;  // do not update my own position, localization is better

  // if opponent or teammate was found and distance lies in tolerance distance
  //  update this opponent or teammate with the specified information.
  // else put the information in the first player position of which we have
  //  no information.
  else if( SoccerTypes::isKnownPlayer(o) &&
          dMinDist < PS->getPlayerDistTolerance())
  {
    processPerfectHearInfo( o, pos, dConf );
    return true;
  }

  if( objInitial == OBJECT_TEAMMATE_UNKNOWN )
    o = getFirstEmptySpotInSet( OBJECT_SET_TEAMMATES );
  else if( objInitial == OBJECT_OPPONENT_UNKNOWN )
    o = getFirstEmptySpotInSet( OBJECT_SET_OPPONENTS );
  else
    return false ;  // in case of OBJECT_PLAYER_UNKNOWN

  if( o != OBJECT_ILLEGAL )   // can be the case that there is no empty spot
  {
    processPerfectHearInfo( o, pos, dConf );
    setIsKnownPlayer( o, false );
  }
  return true;
}

/*! This methods fills the heterogeneous player array (stored as 'pt') with the
    specified information. This information is directly parsed from the
    player_type message received from the server. This method is therefore
    only called from the SenseHandler. The information in this array can later
    be used (by the coach) to determine the best heterogeneous player for a
    specific position on the field and to update the parameters in the
    ServerSettings when the player type of the agent is changed.
    \param iIndex index of the heterogeneous player as indicated by the server
```

```
    \param dPlayerSpeedMax maximum speed of the player
    \param dStaminaIncMax maximum increase of stamina for the player
    \param dPlayerDecay amount of velocity decay of the player
    \param dIntertiaMoment indication how fast is turned when moving
    \param dDashPowerRate how is power in dash command converted into speed
    \param dPlayerSize what is the size of the player
    \param dKickableMarng in which area can player still kick the ball
    \param dKickRand how much noise is added to kick command
    \param dExtraStamina how much extra stamina does player receive
    \param dEffortMax what is the maximum effort (effective dash)
    \param dEffortMin what is the minimum effort
    \return bool indicating whether update was succesfull. */
bool WorldModel::processNewHeteroPlayer( int iIndex,    double dPlayerSpeedMax,
          double dStaminaIncMax, double dPlayerDecay, double dInertiaMoment,
          double dDashPowerRate, double dPlayerSize,  double dKickableMargin,
          double dKickRand,      double dExtraStamina,double dEffortMax,
          double dEffortMin )
{
  pt[iIndex].dPlayerSpeedMax  = dPlayerSpeedMax;
  pt[iIndex].dStaminaIncMax   = dStaminaIncMax;
  pt[iIndex].dPlayerDecay     = dPlayerDecay;
  pt[iIndex].dInertiaMoment   = dInertiaMoment;
  pt[iIndex].dDashPowerRate   = dDashPowerRate;
  pt[iIndex].dPlayerSize      = dPlayerSize;
  pt[iIndex].dKickableMargin  = dKickableMargin;
  pt[iIndex].dKickRand        = dKickRand;
  pt[iIndex].dExtraStamina    = dExtraStamina;
  pt[iIndex].dEffortMax       = dEffortMax;
  pt[iIndex].dEffortMin       = dEffortMin;
  pt[iIndex].dMaximalKickDist = dKickableMargin +
                        dPlayerSize +
                        SS->getBallSize();

  return true;
}

/*! This method sets the catch cycle time for the goalie. This is the
    cycle time that the ball is catched by the goalie. This
    information is said by the referee by indicating the side that
    catched the ball. This method checks whether our side catched the
    ball. If this is the case, the time is set and the goalie cannot
    catch the ball for a certain cycles (see ServerSettings).

    \param rm referee message to indicated side of goalie that catched ball
    \param iTime time the ball was catched.  */
void WorldModel::processCatchedBall( RefereeMessageT rm, Time time )
```

```
{
  if( rm == REFC_GOALIE_CATCH_BALL_LEFT && sideSide == SIDE_LEFT )
    timeLastCatch = time;
  else if( rm == REFC_GOALIE_CATCH_BALL_RIGHT && sideSide == SIDE_RIGHT )
    timeLastCatch = time;
  Ball.setGlobalVelocity( VecPosition(0,0), getCurrentTime() );
}

/*! This method sets the performed commands by the agent object. Using this
    information, the future world states can be calculated when an update is
    performed. In this method a timestamp of the current time cycle is added
    to the commands structure for later usage.
    This method is called by the ActHandler when it has sent these commands to
    the server.
    \param commands all the commands that were sent in this cycle
    \param iCommands number of commands that were sent in this cycle. */
void WorldModel::processQueuedCommands( SoccerCommand commands[],
                                        int iCommands )
{
  if( iCommands > CMD_MAX_COMMANDS )
  {
    cerr << "(WorldModel::setQueuedCommands) queuedCommands array cannot "
         << "contain so many commands!\n";
    return;
  }

  // put all sent commands to the array which stores queued commands.
  for( int i = 0 ; i < iCommands ; i ++ )
  {
    commands[i].time                          = getCurrentTime();
    queuedCommands[(int)commands[i].commandType] = commands[i];
  }
}

/*! This is called to update the WorldModel. It determines the the last
    message (see or sense) and updates the worlmodel accordingly. When see
    information is received ("perfect" information) all objects are updated
    with this received information. After sense message the information of
    the object is calculated for the new cycle using the last visual
    information.
    \return bool to indicate whether update succeeded. */
bool WorldModel::updateAll( )
{
  static Timing timer;
  double dTimeSense = 0.0, dTimeSee = 0.0, dTimeComm=0.0, dTimeFastest = 0.0;
  static struct tms times1, times2;
```

```
bool       bReturn           = false, bUpdateAfterSee = false;
bool       bUpdateAfterSense  = false, bDebug = false;
static Time timeLastHoleRecorded;
static Time timeBeginInterval;
static Time timePlayersCounted;
static int  iNrHolesLastTime   = 0;
static Time timeLastSenseUpdate;
static Time timeLastSeeUpdate;
static Time timeLastSayUpdate;
if( bDebug )
{
 timer.restartTime();
 times( &times1 );
}

// check if last update of agent was not more than one cycle ago
if( agentObject.getTimeGlobalPosition() < getCurrentTime() - 1  )
  Log.log( 3, "(WorldModel::updateAll) missed a sense??" );

// call update method depending on last received message
if( isFullStateOn( ) == true )
{
 Log.log( 4, "full state is on" );
 updateRelativeFromGlobal();
 timeLastSenseMessage = timeLastRecvSeeMessage;
 timeLastSeeMessage   = timeLastRecvSeeMessage;
 bUpdateAfterSee = bUpdateAfterSense = false;
 bReturn = true;
}
else
{
 Log.log( 4, "full state is off" );
 if( getTimeLastRecvSeeMessage() > timeLastSeeUpdate )
   bUpdateAfterSee = true;
 if( getTimeLastRecvSenseMessage() > timeLastSenseUpdate )
   bUpdateAfterSense = true;
}

// rare situation: can occur that see arrives between sense and calling
// updateAll or sense arrives between see and calling updateAll.
if( bUpdateAfterSee && bUpdateAfterSense )
{
 Log.log( 3, "!!! Two updates !!! " );
 Log.log( 3, "see: %d sense: %d", getTimeLastRecvSeeMessage().getTime(),
    getTimeLastRecvSenseMessage().getTime() );
```

```
  if( getTimeLastRecvSeeMessage( ) == getTimeLastRecvSenseMessage() )
  {
    Log.log( 3, "update sense" );
    timeLastSenseMessage = timeLastRecvSenseMessage;
    bReturn  = updateAfterSenseMessage( );
    if( bDebug ) dTimeSense = timer.getElapsedTime(1000);
    updateRelativeFromGlobal();
    Log.log( 3, "update see" );
    timeLastSeeMessage = timeLastRecvSeeMessage;
    bReturn &= updateAfterSeeMessage( );
    if( bDebug ) dTimeSee = timer.getElapsedTime(1000) - dTimeSense;
  }
  else if( getTimeLastRecvSeeMessage( ) < getTimeLastRecvSenseMessage() )
  {
    Log.log( 3, "update see" );
    timeLastSeeMessage = timeLastRecvSeeMessage;
    bReturn  = updateAfterSeeMessage( );
    if( bDebug ) dTimeSee = timer.getElapsedTime(1000);
    Log.log( 3, "update sense" );
    timeLastSenseMessage = timeLastRecvSenseMessage;
    bReturn &= updateAfterSenseMessage( );
    updateRelativeFromGlobal();
    if( bDebug ) dTimeSense = timer.getElapsedTime(1000) - dTimeSee;
  }
  timeLastSenseUpdate = getTimeLastSenseMessage();
  timeLastSeeUpdate   = getTimeLastSeeMessage();
}
else if( bUpdateAfterSee )                    // process see message
{
  Log.log( 3, "update see" );
  timeLastSeeMessage = timeLastRecvSeeMessage;
  bReturn          = updateAfterSeeMessage( );
  timeLastSeeUpdate = getTimeLastSeeMessage();
  if( bDebug ) dTimeSee = timer.getElapsedTime(1000);
}
else if( bUpdateAfterSense )                  // process sense message
{
  Log.log( 3, "update sense" );
  timeLastSenseMessage = timeLastRecvSenseMessage;
  bReturn            = updateAfterSenseMessage( );
  timeLastSenseUpdate = getTimeLastSenseMessage();
  updateRelativeFromGlobal();
  if( bDebug ) dTimeSense = timer.getElapsedTime(1000);
}

if( timeLastSayUpdate != m_timePlayerMsg &&
```

```
    isFullStateOn() == false        )           // process communication msg
{
 Log.log( 3, "update hear" );
 if( bDebug ) dTimeComm = timer.getElapsedTime(1000);
 timeLastSayUpdate = m_timePlayerMsg;
 processPlayerMessage();
 if( bDebug ) dTimeComm = timer.getElapsedTime(1000) - dTimeComm;
}

SoccerCommand soc = getChangeViewCommand( );
if( ! soc.isIllegal() )
{
 setAgentViewAngle( soc.va );
 setAgentViewQuality( soc.vq );
}

// check for holes
if( isQueuedActionPerformed() == false &&
    timeLastHoleRecorded != getCurrentTime() &&
    isFullStateOn() == false )
{
 Log.log( 2, "HOLE recorded" );
 timeLastHoleRecorded = getCurrentTime();
 iNrHoles++;
}

// determine number of holes in last time interval and act accordingly
int    iTimeDiff = getCurrentTime() - timeBeginInterval;
double dHolePerc = (double)(iNrHoles - iNrHolesLastTime)/iTimeDiff*100;
if( ! isLastMessageSee( ) && iTimeDiff % 400 == 0 && dHolePerc > 1.0 &&
    PS->getFractionWaitSeeEnd() > 0.70 )
{
 PS->setFractionWaitSeeEnd( PS->getFractionWaitSeeEnd() - 0.05 );
 timeBeginInterval = getCurrentTime();
 cerr << getCurrentCycle() << ": lowered send time to " <<
        PS->getFractionWaitSeeEnd() << " for player number "   <<
        getPlayerNumber()          <<
        "; nr of holes is "<< dHolePerc << "%" << "( " << iNrHoles << ", "
        << iNrHolesLastTime << ")" << endl;
 iNrHolesLastTime   = iNrHoles;
}

// store some statistics about number of players seen each cycle
if( bUpdateAfterSense == true  && ! isTimeStopped() &&
    getCurrentTime() != timePlayersCounted )
{
```

```
  iNrTeammatesSeen += getNrInSetInRectangle( OBJECT_SET_TEAMMATES );
  iNrOpponentsSeen += getNrInSetInRectangle( OBJECT_SET_OPPONENTS );
  timePlayersCounted = getCurrentTime();
 }

 // log specific information when log level is set
 if( Log.isInLogLevel( 456 ) )
  logObjectInformation( 456, getAgentObjectType() );

 if( bUpdateAfterSee == true )
  Log.logWithTime(3, "  finished update_all see; start determining action" );
 if( bUpdateAfterSense == true )
  Log.logWithTime(3, "  finished update_all sense;start determining action");

 if( Log.isInLogLevel( 459 ) )
 {
  Log.log( 459, "%s%s", strLastSeeMessage, strLastSenseMessage );
  show( OBJECT_BALL, Log.getOutputStream() );
  show( OBJECT_SET_PLAYERS, Log.getOutputStream() );
 }
 if( ( Log.isInLogLevel( 101 ) && getRelativeDistance( OBJECT_BALL ) < 2.0 ) )
  show( OBJECT_BALL, Log.getOutputStream() );
 if( Log.isInLogLevel( 556 ) &&
     getRelativeDistance( OBJECT_BALL ) < SS->getVisibleDistance() )
 {
  Log.log( 556, "%s", strLastSeeMessage );
  show( OBJECT_SET_PLAYERS, Log.getOutputStream() );
  show( OBJECT_BALL, Log.getOutputStream() );
 }
 if( LogDraw.isInLogLevel( 460 ) )
 {
  int iCycles;
  dTimeFastest = timer.getElapsedTime( 1000 );
  ObjectT obj = getFastestInSetTo(OBJECT_SET_OPPONENTS,OBJECT_BALL,&iCycles);
  logCircle( 460, getGlobalPosition( obj ), 1.5 );
  logCircle( 460, predictPosAfterNrCycles( OBJECT_BALL, iCycles ), 0.5 );

obj=getFastestInSetTo(OBJECT_SET_TEAMMATES_NO_GOALIE,OBJECT_BALL,&iCycle
s);
  logCircle( 460, getGlobalPosition( obj ), 1.5 );
  logCircle( 460, predictPosAfterNrCycles( OBJECT_BALL, iCycles ), 0.75 );
  dTimeFastest = timer.getElapsedTime( 1000 ) - dTimeFastest;
 }

 if( LogDraw.isInLogLevel( 701 ) )
  drawCoordinationGraph( );
```

```
  logLine( 602, VecPosition( getOffsideX(), -PITCH_WIDTH/2.0),
         VecPosition( getOffsideX(),  PITCH_WIDTH/2.0) );
  if( bDebug )
  {
    Log.logWithTime( 461, "time update all: %f\n time comm:        %1.5f\n\
    time see:       %1.5f\n time sense:    %1.5f\n time fastest:     %1.5f\n\
    time rest:      %1.5f\n utime:         %1.5f",
    timer.getElapsedTime()*1000, dTimeComm, dTimeSee, dTimeSense,dTimeFastest,
    timer.getElapsedTime()*1000-(dTimeComm+dTimeSee+dTimeSense+dTimeFastest ),
    times2.tms_utime  - times1.tms_utime );
  }
  return bReturn;
}

/*****************************************************************************
/
/*************** WORLDMODEL: SEE RELATED UPDATES
*****************************/
/*****************************************************************************
/

void  WorldModel::processLastSeeMessage( )
{
  ObjectT o;
  double  dDist, dDistChange,    dDirChange,    dPointDir,    dTemp;
  int     iDir;
  AngDeg  angBodyFacingDir, angHeadFacingDir;
  Time    time = getTimeLastSeeMessage();
  bool    isGoalie, isTackling;
  static char strTmp[MAX_MSG];
  char   *strMsg = strLastSeeMessage ;
  Parse::parseFirstInt( &strMsg );        // get the time

  ObjectT objMin     = OBJECT_ILLEGAL;
  ObjectT objMax     = OBJECT_ILLEGAL;

  while( *strMsg != ')' )                      // " ((objname.." or ")"
  {
    dDist = dDistChange = dDirChange = dTemp = dPointDir = UnknownDoubleValue;
    angBodyFacingDir = angHeadFacingDir =         UnknownAngleValue;
    strMsg += 2;          // get the object name
    isTackling = false;

    // get the object name from the first part of the string
//    cerr << "string: |" << strMsg << endl;
```

```
o = SoccerTypes::getObjectFromStr( &strMsg, &isGoalie, getTeamName() );

if( o == OBJECT_ILLEGAL )
{
  Log.log( 4, "Illegal object in: %s", strLastSeeMessage );
  Log.log( 4, "rest of message: %s", strMsg );
}
else if( SoccerTypes::isKnownPlayer( o ) )    // we know the player
  objMin = o;
else if( SoccerTypes::isPlayer( o ) )         // and thus an unknown player
{
  if( SoccerTypes::isTeammate( o ) &&
      ( objMin == OBJECT_ILLEGAL || SoccerTypes::isOpponent( objMin ) ) )
    objMin = OBJECT_TEAMMATE_1 ;
  else if( SoccerTypes::isOpponent( o )  &&
      ( objMin == OBJECT_ILLEGAL || SoccerTypes::isTeammate( objMin ) ) )
    objMin = OBJECT_OPPONENT_1 ;
  else if( objMin == OBJECT_ILLEGAL )
    objMin = (getSide() == SIDE_LEFT ) ? OBJECT_TEAMMATE_1
                                : OBJECT_OPPONENT_1 ;
  else if( objMin == OBJECT_TEAMMATE_11 )
    objMin = OBJECT_OPPONENT_1;
  else if( objMin == OBJECT_OPPONENT_11 )
    objMin = OBJECT_TEAMMATE_1;
  else
    objMin = (ObjectT)((int)objMin + 1);

  if( objMin == getAgentObjectType() )
  {
     if( objMin == OBJECT_TEAMMATE_11 )
       objMin = OBJECT_OPPONENT_1;
     else if( objMin == OBJECT_OPPONENT_11 )
       objMin = OBJECT_TEAMMATE_1;
     else
      objMin = (ObjectT)((int)objMin + 1);
  }
  strcpy( strTmp, strMsg );                 // get the maximum object by
  objMax = getMaxRangeUnknownPlayer( objMin, strTmp ); // look to next obj
}

dTemp = Parse::parseFirstDouble( &strMsg );   // parse first value
if ( *strMsg == ')' )                       // if it was the only value
  iDir = (int)dTemp;                        // it was the direction
else
{
  dDist = dTemp;                            // else it was distance
```

```
  iDir = Parse::parseFirstInt( &strMsg );     // and have to get direction

  double dValues[7];
  int    i = 0;
  if( *strMsg == ' ' )                        // stil not finished
  {                                    // get dist and dir change
    while( *strMsg != ')' && *(strMsg+1) != 't' && i < 7 )
    {
      dValues[i]  = Parse::parseFirstDouble( &strMsg );
      i++;
    }
  }

  switch( i )
  {
    case 0:
      break;
    case 1:
      dPointDir   = dValues[0];
      break;
    case 5:
      dPointDir   = dValues[4]; // fall through
    case 4:
      angBodyFacingDir = dValues[2];
      angHeadFacingDir = dValues[3]; // fall through
    case 2:                        // in case of ball
      dDistChange = dValues[0];
      dDirChange  = dValues[1];
      break;
    default:
      cerr << "(WorldModelUpdate::analyzeSee) wrong param nr " << i <<endl;

  }

  if( *(strMsg+1) == 't')
    isTackling = true;
}

// go to end object information
// skip ending bracket of object information.
Parse::gotoFirstOccurenceOf( ')', &strMsg );
strMsg++;

if( SoccerTypes::isPlayer( o ) && !SoccerTypes::isKnownPlayer( o ) )
{
  if( objMin != OBJECT_ILLEGAL )
```

```
      {
       if( objMin == objMax )
       {
         Log.log( 459, "range, only %d left", objMin );
         o = objMin;
       }
       else if( SoccerTypes::isTeammate( objMin ) &&
              SoccerTypes::isTeammate( objMax ) )
         o = OBJECT_TEAMMATE_UNKNOWN;
       else if( SoccerTypes::isOpponent( objMin ) &&
              SoccerTypes::isOpponent( objMax ) )
         o = OBJECT_OPPONENT_UNKNOWN;
      }
    }

    // process the parsed information (unread values are Unknown...)
    processNewObjectInfo( o, time, dDist, iDir, dDistChange,
          dDirChange, angBodyFacingDir, angHeadFacingDir,
          isGoalie, objMin, objMax, dPointDir, isTackling );
  }
}

/*! This method is called to update the WorldModel after a see message.
    The update methods for the agent, all teammates, all opponents and the
    ball are called. But only when the last see message of these objects equals
    the time of the last received see message.
    \return bool to indicate whether update succeeded. */
bool WorldModel::updateAfterSeeMessage( )
{
  processLastSeeMessage( );

   // update the agent (global position and angle using flags and lines)
  if( getCurrentTime().getTime() != -1 )
    updateAgentObjectAfterSee( );

  mapUnknownPlayers( getTimeLastSeeMessage() ); // map players with unknown nr

  // walk past all players on the field an when new information was perceived
  // (and put in the relative attributes) update this dynamic object. When it
  // was not seen, convert its global position (this is an estimate from the
  // sense message) to a relative position
  double dConfThr = PS->getPlayerConfThr();
  int    iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_PLAYERS, dConfThr );
       o != OBJECT_ILLEGAL;
```

```
      o = iterateObjectNext ( iIndex, OBJECT_SET_PLAYERS, dConfThr ) )
  {
   if( getTimeLastSeen( o ) == getTimeLastSeeMessage() &&
       o != getAgentObjectType()  )
     updateDynamicObjectAfterSee   ( o );
   else
     updateObjectRelativeFromGlobal( o );
  }
  iterateObjectDone( iIndex );

  // if ball was seen update him, otherwise make estimated global relative
  if( getTimeLastSeen( OBJECT_BALL ) == getTimeLastSeeMessage() )
    updateDynamicObjectAfterSee   ( OBJECT_BALL );
  else
    updateObjectRelativeFromGlobal( OBJECT_BALL );

  // delete objects from wordmodel that should have been seen, but aren't
  if( Log.isInLogLevel( 2 ) )
    show( OBJECT_BALL, Log.getOutputStream() );
  removeGhosts();
  if( Log.isInLogLevel( 2 ) )
    show( OBJECT_BALL, Log.getOutputStream() );


  return true;
}

/*! This method updates an agent after a see message.
    The global position and the global neck angle are calculated using the
    visual information using the method calculateStateAgent.
    \return bool to indicate whether update succeeded. */
bool WorldModel::updateAgentObjectAfterSee(  )
{
  VecPosition posGlobal, velGlobal;
  AngDeg      angGlobal;

  // calculate the state of the agent
  calculateStateAgent( &posGlobal, &velGlobal, &angGlobal );

  // and set the needed attributes
  agentObject.setTimeLastSeen         ( getTimeLastSeeMessage() );
  // store difference with predicted global position to compensate for error
  // in global position when global velocity is calculated for other objects.
  agentObject.setPositionDifference(posGlobal-agentObject.getGlobalPosition());
  agentObject.setGlobalPosition       ( posGlobal, getTimeLastSeeMessage() );
  agentObject.setGlobalPositionLastSee( posGlobal, getTimeLastSeeMessage() );
```

```
  agentObject.setGlobalNeckAngle      ( angGlobal );
  agentObject.setGlobalVelocity       ( velGlobal, getTimeLastSeeMessage() );

  return true;
}


/*! This method updates a dynamic object after a see message.
    The global position and the velocity are calculated using the
    visual information. For the ball the method 'calculateStateBall'
    is called, for players the method 'calculateStatePlayer' is called.
    \param o object that should be updated after a see message */
bool WorldModel::updateDynamicObjectAfterSee( ObjectT o )
{
  VecPosition posGlobal, velGlobal;

  if( o == OBJECT_BALL )
  {
    calculateStateBall          ( &posGlobal, &velGlobal             );
    Ball.setGlobalVelocity      ( velGlobal,  getTimeLastSeeMessage() );
    Ball.setGlobalPosition      ( posGlobal,  getTimeLastSeeMessage() );
    Ball.setGlobalPositionLastSee( posGlobal,  getTimeLastSeeMessage() );
    return true;
  }
  else if( SoccerTypes::isKnownPlayer( o )  )
  {
    calculateStatePlayer( o, &posGlobal, &velGlobal );

    PlayerObject *pob = (PlayerObject*) getObjectPtrFromType( o );

    // the time of the velocity does not have to be set, since it equals
    // the time the last change information has been set.
    pob->setGlobalVelocity      ( velGlobal, getTimeLastSeeMessage() );
    pob->setGlobalPosition      ( posGlobal, getTimeLastSeeMessage() );
    pob->setGlobalPositionLastSee( posGlobal, getTimeLastSeeMessage() );

    if( pob->getTimeRelativeAngles() == getTimeLastSeeMessage() )
    {
      AngDeg ang = getAgentGlobalNeckAngle() + pob->getRelativeBodyAngle();
      ang = VecPosition::normalizeAngle( ang );
      pob->setGlobalBodyAngle( ang, getTimeLastSeeMessage() );
      pob->setGlobalBodyAngleLastSee( ang );
      ang = getAgentGlobalNeckAngle() + pob->getRelativeNeckAngle();
      ang = VecPosition::normalizeAngle( ang );
      pob->setGlobalNeckAngle( ang, getTimeLastSeeMessage() );
      pob->setGlobalVelocityLastSee( velGlobal );
```

```
    }
    return true;
  }

  return false;
}


/*************************************************************************
/
/****************** SENSE RELATED UPDATES
************************************/
/*************************************************************************
/

void WorldModel::processLastSenseMessage( )
{
  char   *strMsg = strLastSenseMessage ;

  Parse::parseFirstInt( &strMsg );// get time
  strMsg++;                              // go to ( before view_mode

  Parse::gotoFirstOccurenceOf( ' ', &strMsg );   // skip view_mode
  strMsg++;                              // skip space

  ViewQualityT vq = SoccerTypes::getViewQualityFromStr( strMsg );// get quality
  Parse::gotoFirstOccurenceOf( ' ', &strMsg );
  strMsg++;                              // skip space; get view_angle
  ViewAngleT va = SoccerTypes::getViewAngleFromStr( strMsg );
  double dStamina = Parse::parseFirstDouble( &strMsg );  // get stamina
  double dEffort  = Parse::parseFirstDouble( &strMsg );  // get effort

  double dSpeed   = Parse::parseFirstDouble( &strMsg );  // get speed
  AngDeg angSpeed = Parse::parseFirstDouble( &strMsg );  // get speed ang

  // minus sign since we store angle between neck and body and not vice versa
  int iHeadAngle = - Parse::parseFirstInt( &strMsg );    // get head_angle

  // set all number of performed commands
  setNrOfCommands( CMD_KICK       , Parse::parseFirstInt( &strMsg ) );
  setNrOfCommands( CMD_DASH       , Parse::parseFirstInt( &strMsg ) );
  setNrOfCommands( CMD_TURN       , Parse::parseFirstInt( &strMsg ) );
  setNrOfCommands( CMD_SAY        , Parse::parseFirstInt( &strMsg ) );
  setNrOfCommands( CMD_TURNNECK   , Parse::parseFirstInt( &strMsg ) );
  setNrOfCommands( CMD_CATCH      , Parse::parseFirstInt( &strMsg ) );
  setNrOfCommands( CMD_MOVE       , Parse::parseFirstInt( &strMsg ) );
```

```
  setNrOfCommands( CMD_CHANGEVIEW , Parse::parseFirstInt( &strMsg ) );
  int    iArmMovable = Parse::parseFirstInt( &strMsg );         // arm movable
  int    iArmExpires = Parse::parseFirstInt( &strMsg );         // arm expires
  double dArmDist    = Parse::parseFirstDouble( &strMsg );      // arm dist
  AngDeg angArm      = Parse::parseFirstDouble( &strMsg );       // arm dir

  setNrOfCommands( CMD_POINTTO    , Parse::parseFirstInt( &strMsg ) );// count

  // focus target can be none (no integer) so check this
  int i = Parse::parseFirstInt( &strMsg );
  char c = (i >= 10 ) ? *(strMsg-4) : *(strMsg-3);
  Log.log( 602, "focus %d |%c|", i, c );
  if( c == 'l' || c == 'r' )    // target l or r
  {
   Log.log( 602, "set focus: %d",
                SoccerTypes::getTeammateObjectFromIndex( -1 + i ) );
   setObjectFocus( SoccerTypes::getTeammateObjectFromIndex( -1 + i ) );
   i = Parse::parseFirstInt( &strMsg );
  }
  setNrOfCommands( CMD_ATTENTIONTO , i );

  int iTackleExpires = Parse::parseFirstInt( &strMsg );   // tackle expires
  setNrOfCommands( CMD_TACKLE , Parse::parseFirstInt( &strMsg ) );

  angArm = VecPosition::normalizeAngle( angArm );
  processNewAgentInfo( vq, va, dStamina, dEffort, dSpeed,
       (AngDeg) angSpeed, (AngDeg)iHeadAngle, iTackleExpires,
        iArmMovable, iArmExpires, VecPosition( dArmDist, angArm, POLAR ));
}

/*! This method is called to update the WorldModel after a sense message.
    The current information for the agent, all teammates, all opponents and the
    ball are calculated using the associated methods. This is done by updating
    the objects till the time of the information matches the current time of
    the sense message. After all global
    positions are determined, the relative information is updated using the new
    global information using the method determineRelativeFromGlobal().
    \return bool to indicate whether update succeeded. */
bool WorldModel::updateAfterSenseMessage( )
{
  processLastSenseMessage( );

  // update agent information
  updateAgentAndBallAfterSense( );

  // update all global information of players that have a good confidence
```

```
  double dConfThr = PS->getPlayerConfThr();
  int    iIndex;

  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_PLAYERS, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_PLAYERS, dConfThr ) )
  {
    if( o != getAgentObjectType() && getTimeGlobalPosition(o) > 0 &&
        ! isBeforeKickOff() )
    {
      updateDynamicObjectForNextCycle( o,
                           getCurrentTime() - getTimeGlobalPosition(o) );
    }
  }
  iterateObjectDone( iIndex);

  // before the kick off all player information can be set to their strat. pos
  if( isBeforeKickOff() )
  {
    for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_TEAMMATES, -2000 );
         o != OBJECT_ILLEGAL;
         o = iterateObjectNext ( iIndex, OBJECT_SET_TEAMMATES, -2000  ) )
   {
      if( o == getAgentObjectType() )
        continue;
      PlayerObject *pob = (PlayerObject*) getObjectPtrFromType( o );

      VecPosition pos = getStrategicPosition( SoccerTypes::getIndex( o )  ) ;
      VecPosition vel( 0, 0 );

      pob->setGlobalVelocity       ( vel, getTimeLastSenseMessage() );
      pob->setGlobalPosition       ( pos, getTimeLastSenseMessage() );
      pob->setGlobalPositionLastSee( pos, getTimeLastSenseMessage() );
      pob->setTimeLastSeen         ( getTimeLastSenseMessage() - 2 );
      updateObjectRelativeFromGlobal( o );
      Log.log( 459, "set info %d (%f,%f)", SoccerTypes::getIndex( o ),
        pos.getX(), pos.getY() );
    }
    Ball.setGlobalPosition( VecPosition(0,0), getTimeLastSenseMessage() );
    Ball.setGlobalVelocity( VecPosition(0,0), getTimeLastSenseMessage() );
  }
//    show( OBJECT_SET_PLAYERS, Log.getOutputStream() );
  iterateObjectDone( iIndex);

  return true;
}
```

```
/*! This method updates the agent and ball information after a sense message.
    This is done using the current known information and the action that is
    performed in the last server cycle as indicated by the attribute
    'PerformedCommands'
    \return bool to indicate whether update succeeded. */
bool WorldModel::updateAgentAndBallAfterSense( )
{
  // get info from commands from previous cycle (if timestopped current cycle)
  bool       bBallUpdated   = false;
  VecPosition pos           = getAgentGlobalPosition();
  AngDeg      angGlobalNeck  = getAgentGlobalNeckAngle();
  AngDeg      angGlobalBody  = getAgentGlobalBodyAngle();
  Stamina     sta            = getAgentStamina();
  VecPosition velNeck        = agentObject.getSpeedRelToNeck(); // !!
  VecPosition vel            = getAgentGlobalVelocity();
  Time        time           = getCurrentTime() - 1 ;

  // first update information based on performed actions without using
  // the received velocity information.
  for( int i = 0; i < CMD_MAX_COMMANDS; i ++  )
  {
    if( performedCommands[i] == true && // sense msg indicates we executed this
        ( queuedCommands[i].time.getTime() == time.getTime() ||      // no hole
          queuedCommands[i].time.getTime() == time.getTime() - 1 ) )// hole
    {
      switch( queuedCommands[i].commandType )
      {
        case CMD_TURN:
        case CMD_TURNNECK:
        case CMD_DASH:
        case CMD_TACKLE: // in case of tackle do not update ball -> we do not
                         // know whether tackle succeeded
          predictStateAfterCommand( queuedCommands[i], &pos, &vel,
                                    &angGlobalBody, &angGlobalNeck,
                                    getAgentObjectType(), &sta );
          break;
        case CMD_KICK:
          updateBallAfterKick( queuedCommands[i] );
          bBallUpdated     = true;         // should not be updated later on.
          queuedCommands[i].time.updateTime( -1 );
          break;
        case CMD_MOVE:
          pos.setVecPosition( queuedCommands[i].dX, queuedCommands[i].dY );
          initParticlesAgent( pos );
          break;
```

```
      default:
        break;
    }
  }
}

// update the ball when not done yet
if( !bBallUpdated )
{
  m_bPerformedKick = false;
  updateDynamicObjectForNextCycle( OBJECT_BALL, 1 );
}
else
  m_bPerformedKick = true;

// now use velocity information to check whether a collision occured
velNeck.rotate( angGlobalNeck ); // make relative info global
double dDistBall = pos.getDistanceTo( getBallPos() );

if( velNeck.getMagnitude() < EPSILON &&
    vel.getMagnitude() > 0.01 &&
    dDistBall < 0.6 )
  Log.log( 102, "I assume collision after kick" );

if( (
      velNeck.getMagnitude() < EPSILON &&
      vel.getMagnitude() > 0.01 &&
      dDistBall < 0.8
      )
    ||
  ( dDistBall<SS->getPlayerSize()+SS->getBallSize() && // ball too close AND
      velNeck.getMagnitude( ) < EPSILON                // vel. gives no info
    )
   ||                                      // OR
   ( dDistBall < SS->getMaximalKickDist() + 0.5  &&     // ball close
    velNeck.getMagnitude( ) > EPSILON &&               // can compare vel
    ( sign( vel.getX() ) != sign( velNeck.getX() ) ||   // both signs have
      fabs( vel.getX() ) < 0.08 )               &&     // changed or are
    ( sign( vel.getY() ) != sign( velNeck.getY() ) ||   // so small it is in
      fabs( vel.getY() ) < 0.08 )               &&     // error range
     velNeck.getMagnitude() < 0.25 * vel.getMagnitude() ) ) // lessened a lot
{
  m_bWasCollision = true;
  m_timeLastCollision = getCurrentTime();
  double dDistPlayer;
  getClosestInSetTo(
```

```
      OBJECT_SET_PLAYERS, getAgentObjectType(), &dDistPlayer );
  if( dDistBall < dDistPlayer )
  {
    updateBallForCollision( pos );
    Log.log( 102, "COLLISION WITH BALL %f, vel(%f,%f) velNeck(%f,%f)",
       dDistBall, vel.getX(), vel.getY(), velNeck.getX(), velNeck.getY() );
  }
  else
    Log.log( 102, "COLLISION WITH PLAYER vel(%f,%f) velNeck(%f,%f)",
       vel.getX(), vel.getY(), velNeck.getX(), velNeck.getY() );

  // new vel. agent is the one received from sense message
  vel = velNeck;
}
else
{
  if( dDistBall < SS->getVisibleDistance() )
    Log.log( 102, "No collision: dist: %f, velNeck (%f,%f), vel (%f,%f)",
      dDistBall, velNeck.getX(), velNeck.getY(), vel.getX(), vel.getY() );
  m_bWasCollision = false;
  // use better vel. estimate in sense message to update information
  pos            = getAgentGlobalPosition();
  angGlobalNeck  = getAgentGlobalNeckAngle();
  angGlobalBody  = getAgentGlobalBodyAngle();
  sta            = getAgentStamina();
  vel            = agentObject.getSpeedRelToNeck();

  // calculate velocity at end of previous cycle using velocity from
  // current cycle. Although we do not know direction yet (this is
  // relative to neck which is not yet known), we can use the
  // magnitude to determine traveled  distance (speed) of the agent
  // After neck angle is estimated, we can rotate velocity vector
  // to get actual velocity.
  vel.setMagnitude( vel.getMagnitude()/SS->getPlayerDecay() );

  for( int i = 0; i < CMD_MAX_COMMANDS; i ++  )
  {
    if( performedCommands[i] == true &&
       ( queuedCommands[i].time.getTime() == time.getTime() ||
         queuedCommands[i].time.getTime() == time.getTime() - 1 ) )
    {
      switch( queuedCommands[i].commandType )
      {
        case CMD_TURN:
        case CMD_TURNNECK:
          predictStateAfterCommand( queuedCommands[i], &pos, &vel,
```

```
                    &angGlobalBody, &angGlobalNeck,
                    getAgentObjectType(), &sta );
        break;
      case CMD_MOVE:
        pos.setVecPosition( queuedCommands[i].dX, queuedCommands[i].dY );
        initParticlesAgent( pos );
        break;
      case CMD_DASH:    // no action needed, since resulting
                        // vel. is already available from sense
                        // this vel can be used to update pos
        break;
      case CMD_TACKLE:  // no action needed, we are never sure that tackle
                        // is executed
        break;
      default:
        break;
      }
    }
    queuedCommands[i].time.updateTime( -1 );          // processed
  }
  // reset pos and vel information to previous cycle (since can be
  // changed in predictStateAfterCommand)
  vel = agentObject.getSpeedRelToNeck();
  pos = getAgentGlobalPosition();
  vel.setMagnitude( vel.getMagnitude()/SS->getPlayerDecay() );
  vel.rotate( angGlobalNeck ); // rotate vel using information about neck

  // predict global position using the calculated vel at the end of
  // the previous cycle (power and direction can thus both be set
  // to zero). There is just little noise in this perception, so
  // gives a good estimate
  predictStateAfterDash( 0.0, &pos, &vel, &sta, 0, getAgentObjectType() );
 }

 // update particles that keep track of position of agent using vel
 updateParticlesAgent( vel, true );

 // body angle is not set since relative angle to neck is already
 // set after parsing sense_body message, same holds for stamina
 agentObject.setGlobalPosition ( pos,       getCurrentTime() );
 agentObject.setGlobalVelocity ( vel,       getCurrentTime() );
 agentObject.setGlobalNeckAngle( angGlobalNeck );

 return true;
}
```

```
/*! This methods updates the ball after a kick command. First it is checked
    whether the ball is indeed in the kickable range. If this is the case
    the accelaration of the ball is calculated and added to the current
    velocity. With this information the new global position of the ball
    is set.
    \param soc performed kick command
    \return bool indicating whether update was succesful */
bool WorldModel::updateBallAfterKick( SoccerCommand soc )
{
  if( getRelativeDistance( OBJECT_BALL ) < SS->getMaximalKickDist() )
  {
    VecPosition posBall, velBall;
    predictBallInfoAfterCommand( soc, &posBall, &velBall );
    Ball.setGlobalPosition( posBall, getCurrentTime()  );
    Ball.setGlobalVelocity( velBall, getCurrentTime()  );
// updateParticlesBall( particlesPosBall, particlesVelBall,
//                      iNrParticlesBall, dPower, ang );
  }
  else
  {
    updateDynamicObjectForNextCycle( OBJECT_BALL, 1 );
//  updateParticlesBall( particlesPosBall, particlesVelBall,
//  iNrParticlesBall, 0, 0 );
#ifdef WIN32
    Log.log( 21, "(WorldModel::%s) KICK command, but ball not kickable (%f)",
        "updateBallAfterKick", getRelativeDistance( OBJECT_BALL ) );
#else
    Log.log( 21, "(WorldModel::%s) KICK command, but ball not kickable (%f)",
        __FUNCTION__, getRelativeDistance( OBJECT_BALL ) );
#endif
  }
  return true;
}

/*! This methods updates a dynamic object for 'iCycles' cycle. This is done by
    updating the global position with the velocity vector. After this
    the velocity vector is decreased with the associated decay.
    \param obj object that should be updated
    \param iCycles denotes for how many cycles dynamic object should be updated
    \return bool indicating whether update was succesful */
bool WorldModel::updateDynamicObjectForNextCycle( ObjectT obj, int iCycles)
{
  DynamicObject *o = (DynamicObject*) getObjectPtrFromType( obj );
  if( o == NULL )
    return false;
```

```
// get velocity and add it to current global position
VecPosition vel = getGlobalVelocity( obj );
VecPosition pos = getGlobalPosition( obj );
VecPosition posBall = getBallPos();
//  ObjectT objFastest =  getFastestInSetTo( OBJECT_SET_TEAMMATES,
//                                  OBJECT_BALL );
if( SoccerTypes::isBall( obj ) )
{
  for( int i = 0; i < iCycles ; i++ )
  {
    pos += vel;
    vel *= SS->getBallDecay();
  }
  double  dDist;
  ObjectT objOpp =
      getClosestInSetTo( OBJECT_SET_OPPONENTS, OBJECT_BALL, &dDist );
  if( objOpp != OBJECT_ILLEGAL &&
      pos.getDistanceTo( getGlobalPosition( objOpp ) )
                    < getMaximalKickDist( objOpp ) )
  {
    Log.log( 556, "update dyn. obj; set ball velocity to 0, opp has it" );
    vel.setVecPosition(0,0);
  }
}
else if( SoccerTypes::isTeammate( obj ) &&
        obj != OBJECT_TEAMMATE_1 && getPlayMode() == PM_PLAY_ON )
{
  for( int i = 0; i < iCycles ; i++ )
  {
    // no idea of intention opponent, just let him roll out
    pos += vel;
    vel *= SS->getPlayerDecay();
  }
}
else if( SoccerTypes::isOpponent( obj ) )
{
  VecPosition velBall = getGlobalVelocity( OBJECT_BALL );
  for( int i = 0; i < iCycles ; i++ )
  {
    // no idea of intention opponent, just let him roll out
    if( obj == getOppGoalieType() )
          ;
    // if it is the fastest opponent, move him towards ball
    else if( obj == getFastestInSetTo( OBJECT_SET_OPPONENTS, OBJECT_BALL ) )
    {
         AngDeg ang = (getBallPos()-getGlobalPosition( obj )).getDirection();
```

```
            pos += VecPosition( 0.3, ang, POLAR );
            vel *= SS->getPlayerDecay();
            Log.log( 556, "update fastest opp to (%f,%f) cyc %d",
                     pos.getX(), pos.getY(), iCycles );
        }
        else if( velBall.getX() > 0 && ! ( isDeadBallUs() || isDeadBallThem() )
                 && ! ( fabs( pos.getX() ) > PENALTY_X + 5 ) )
        {
            pos += VecPosition( (velBall.getX() > 1.0) ? 0.5 : 0.25, 0 );
            vel *= SS->getPlayerDecay();
        }
        else if( ! ( isDeadBallUs() || isDeadBallThem() )
                 && ! ( fabs( pos.getX() ) > PENALTY_X + 5 ) )
        {
            //     pos += vel;
            pos -= VecPosition( (velBall.getX() < -1.0 ) ? 0.5 : 0.25, 0 );
            vel *= SS->getPlayerDecay();
        }
      }
    }

    o->setGlobalVelocity ( vel, getCurrentTime() );
    o->setGlobalPosition ( pos, getCurrentTime() );

    return true;
}


/*! This method checks updates the ball information when it overlaps with a
    player. If this is the case the player and the ball are moved to a position
    where they do not overlap any more. Both velocities are multiplied with 0.1
    \return true when collision occured, false otherwise */
bool WorldModel::updateBallForCollision( VecPosition posAgent )
{
    VecPosition posBall  = getGlobalPosition( OBJECT_BALL );
    VecPosition velBall  = getGlobalVelocity( OBJECT_BALL );

    // get the direction the ball was coming from and put it at small distance
    // and multiply the velocity with -0.1
    AngDeg ang = (posBall - posAgent).getDirection();
    Ball.setGlobalPosition( posAgent + VecPosition( 0.2, ang, POLAR ),
                            getCurrentTime() );
    Ball.setGlobalVelocity( velBall*-0.1, getCurrentTime() );
    Log.log( 102, "updateBallForCollision; vel from (%f,%f) to (%f,%f)",
          velBall.getX(), velBall.getY(), getGlobalVelocity(OBJECT_BALL).getX(),
          getGlobalVelocity(OBJECT_BALL).getY() );
```

```
  return true;
}

/*! This method uses the global position of the agent and the global
    position of all objects to update their relative information. When
    the new global position of the objects is updated after a sense message,
    their relative information is not up to date.
   \return bool indicating whether update was succesful */
bool WorldModel::updateRelativeFromGlobal()
{
  double dConfThr = PS->getPlayerConfThr();
  int    iIndex;

  // update all player objects
  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_PLAYERS, dConfThr );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_PLAYERS, dConfThr ) )
  {
    if( o != getAgentObjectType() )
      updateObjectRelativeFromGlobal( o );
  }
  iterateObjectDone( iIndex);

  // and also the ball
  if( isConfidenceGood( OBJECT_BALL ) )
    updateObjectRelativeFromGlobal( OBJECT_BALL );

  // flags and lines are not updated, since they are not used except
  // immediately after see message when they're up to date.
  return true;
}

/*! This method updates the relative information for one object using the
    global information of this object and the global information of the
    agent.
    \param o object information that should be updated.
    \return bool indicating whether the update was succesfull. */
bool WorldModel::updateObjectRelativeFromGlobal( ObjectT o )
{
  Object *obj = (Object*) getObjectPtrFromType( o );
  if( obj == NULL )
    return false;

  // get global position and translate and rotate it to agent neck
  VecPosition rel = obj->getGlobalPosition();
  rel.globalToRelative( getAgentGlobalPosition(), getAgentGlobalNeckAngle() );
```

```
  obj->setRelativePosition( rel, obj->getTimeGlobalPosition() );
  return true;
}

/*! This method calculates the different state information of the agent, that
    is the global position, global velocity and global neck angle using the
    available information in the world model. This method uses a particle
    filter to determine this information.
    \param posGlobal will be filled with global position of the agent
    \param velGlobal will be filled with global velocity of the agent
    \param angGlobal will be filled with global neck angle of the agent
    \return global position of the agent */
bool WorldModel::calculateStateAgent( VecPosition *posGlobal,
                VecPosition *velGlobal, AngDeg *angGlobal )
{
  int    iNrLeft;

  // first determine global neck angle
  ObjectT objLine = getFurthestRelativeInSet( OBJECT_SET_LINES );
  if( objLine != OBJECT_ILLEGAL )
  {
    double angGlobalLine = getGlobalAngle  ( objLine );
    AngDeg angRelLine    = getRelativeAngle( objLine );
    *angGlobal           = angGlobalLine - angRelLine;
    *angGlobal           = VecPosition::normalizeAngle( *angGlobal );
  }
  else
  {
    Log.log( 21,
           "(WorldModel::calculateStateAgent) no line in last see message" );
    *angGlobal           = getAgentGlobalNeckAngle();
  }

  // use global neck angle to determine estimate of current global velocity
  // neck angle is better estimate than after sense -> better estimate velocity
  // update all position particles of the agent with this velocity
  //  'false' denotes update after see message. Since global neck angle can
  //  be determined more precise after 'see' it is better to predict position
  //  again (although it was already done after sense)
  // and then check which particles are possible given current perceptions
  *velGlobal = agentObject.getSpeedRelToNeck().rotate( *angGlobal );
  velGlobal->setMagnitude( velGlobal->getMagnitude()/SS->getPlayerDecay() );

  updateParticlesAgent         ( *velGlobal, false );
  iNrLeft = checkParticlesAgent ( *angGlobal        );
```

```
  if( iNrLeft == 0 ) // if no particles left, initialize all particles
  {
    // initialize particles (from random samples using closest flag)
    // check particles are then checked with other flags
    initParticlesAgent ( *angGlobal );
    iNrLeft = checkParticlesAgent( *angGlobal );
    if( iNrLeft == 0 )
    {
      // not succeeded, use second method (weighted average flags)
      // and initialize all particles to this position
      calculateStateAgent2( posGlobal, velGlobal, angGlobal );
      initParticlesAgent( *posGlobal );
      return false;
    }
  }

  // determine global position (= average of all particles)
  // and resample all particles
  *posGlobal = averageParticles( particlesPosAgent, iNrLeft );
  resampleParticlesAgent( iNrLeft );

  // use the position to calculate better global neck angle of the agent
  // and recalculate global velocity with improved neck angle
  AngDeg ang = calculateAngleAgentWithPos( *posGlobal );
  if( ang != UnknownAngleValue )
    *angGlobal = ang;

  *velGlobal = agentObject.getSpeedRelToNeck().rotate(*angGlobal);
  return true;
}

/*! This method initializes all particles that represent the global position
    of the agent. This is done using the closest perceived flag. Points are
    generated from the area that could generate the perceived information.
    The global neck angle of the agent 'angGlobal' is used to determine the
    global position of the agent based on the perceived relative information
    to the closest flag.
    \param angGlobal global neck angle of the agent */
void WorldModel::initParticlesAgent( AngDeg angGlobal )
{
  double  dDist, dMaxRadius, dMinRadius, dInput;
  AngDeg  ang;

  // get closest flag from which samples will be generated
  ObjectT objFlag = getClosestRelativeInSet( OBJECT_SET_FLAGS );
```

```
 if( objFlag == OBJECT_ILLEGAL )
 {
   Log.log( 21, "(WorldModel::%s) no flag in last see message",
      "initParticlesAgent" );
   return;
 }

 // get the distance to this flag and the possible range it was derived from.
 dInput = getRelativeDistance( objFlag );
 getMinMaxDistQuantizeValue( dInput, &dMinRadius, &dMaxRadius,
                                    SS->getQuantizeStepL(), 0.1 ) ;

 // get the perceived angle to this flag (add 180 to get angle relative from
 // flag to agent ) and make it global by adding global neck angle agent.
 AngDeg angFlag   = getRelativeAngle( objFlag ) + 180 + angGlobal ;
 // for all particles
 for( int i = 0 ; i < iNrParticlesAgent ; i++ )
 {
   // determine random point from distance range and
   // determine random point from the range it could be generated from
   // angles are rounded and since noise is in global neck angle and relative
   // angle flag, maximum error is in range [-0.5,0.5] + [-0.5,0.5] = [-1,1].
   dDist = dMinRadius + drand48()*(dMaxRadius-dMinRadius);
   ang   = VecPosition::normalizeAngle( angFlag - 1.0 + 2*drand48() );

   // create random point from possible interval
   particlesPosAgent[i] = getGlobalPosition( objFlag ) +
                      VecPosition( dDist, ang, POLAR );
 }
}

/*! This method initializes all particles that represent the global position
   of the agent. All points are initialized with the position 'posInitial'.
   This method can be used when you are completely sure of the position of
   the agent (for example after a 'move' command).
   \param posInitial global position of the agent */
void WorldModel::initParticlesAgent( VecPosition posInitial )
{
 for( int i = 0 ; i < iNrParticlesAgent ; i++ )
   particlesPosAgent[i] = posInitial;
}

/*! This method checks all the particles that represent the global
   position of the agent using the flags in the last perceived see
   message. The global neck angle of the agent is used to make the
   perceived information global. Particles that are not possible are
```

```
    deleted and all legal particles will be shifted to the beginning
    of the array. The number of legal particles is returned.

    \param angGlobalNeck global neck of the angle
    \return number of legal particles */
int WorldModel::checkParticlesAgent( AngDeg angGlobalNeck  )
{
  double dMaxRadius, dMinRadius, dInput, dDist;
  AngDeg ang;
  int    iIndex, iNrLeft = iNrParticlesAgent, iLength = iNrParticlesAgent;

  // for all current perceived flags
  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_FLAGS, 1.0 );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_FLAGS, 1.0 ) )
  {
    iNrLeft = 0;                      // reset number of correct particles
    dInput = getRelativeDistance( o );  // get possible distance range
    getMinMaxDistQuantizeValue( dInput, &dMinRadius, &dMaxRadius,
                                SS->getQuantizeStepL(), 0.1 )  ;

    // find all "correct points"
    for( int i = 0; i < iLength; i ++ )
    {
      // determine distance particle to flag
      // determine difference in direction between direction between global
      // flag and agent position and global perceived direction
      dDist = particlesPosAgent[i].getDistanceTo( getGlobalPosition( o ) );
      ang = (getGlobalPosition(o) - particlesPosAgent[i]).getDirection();
      ang = ang - ( getRelativeAngle( o ) + angGlobalNeck );

      // if in possible range, save it (maximum angle range is 0.5 for
      // neck angle and 0.5 for relative flag angle gives 1.0)
      if( dDist > dMinRadius && dDist < dMaxRadius &&
          fabs(VecPosition::normalizeAngle( ang )) <= 1.0 )
        particlesPosAgent[iNrLeft++] = particlesPosAgent[i];
    }
    // change maximum of correct particles
    iLength = iNrLeft;
  }
  return iNrLeft;
}

/*! This method updates all the particles that represent the global position
    of the agent.The velocity 'vel' is added to each particle to move to
    the next location. When the last boolean argument is true, the velocity
```

```
    is simple added. If this value equals true the previously added velocity
    is subtracted before the specified velocity 'vel' is added. This for
    instance occurs when a better velocity estimate (after a see message) is
    determined with which the particles should be updated.
    \param velocity that should be added to each particle
    \param bAfterSense bool denoting whether this update is done after sense
          or see. When true last added velocity is first subtracted. */
void WorldModel::updateParticlesAgent( VecPosition vel, bool bAfterSense )
{
  // used to denote last added velocity
  static VecPosition prev_vel;

  for( int i = 0; i < iNrParticlesAgent  ; i ++ )
  {
    if( bAfterSense == false ) // if after see, subtract last added 'vel'
    {
      particlesPosAgent[i].setX( particlesPosAgent[i].getX()-prev_vel.getX());
      particlesPosAgent[i].setY( particlesPosAgent[i].getY()-prev_vel.getY());
    }

    particlesPosAgent[i].setX( particlesPosAgent[i].getX( ) + vel.getX() );
    particlesPosAgent[i].setY( particlesPosAgent[i].getY( ) + vel.getY() );
  }
  prev_vel = vel;
}


/*! This method returns the average of the particles contained in 'posArray'.
    \param posArray containing all possible global positions of the agent
    \param iLength number of particles in posArray
    \return average position of all the particles */
VecPosition WorldModel::averageParticles( VecPosition posArray[], int iLength )
{
  if( iLength == 0 )
    return VecPosition( UnknownDoubleValue, UnknownDoubleValue );

  // take average of particles
  double x = 0, y = 0;
  for( int i = 0; i < iLength  ; i ++ )
  {
     x += posArray[ i ].getX( );
     y += posArray[ i ].getY( );
  }

  return VecPosition( x/iLength, y/iLength );
}
```

```
/*! This method resamples the particles that represent the global position of
    the agent. 'iLength' denotes the number of particles contained in
    'posArray' and 'iLeft' the points that do do not have to be resampled.
    A particle is resampled by taking a random particle from the first part
    of the array.
    \param iLeft number of particles that should be contained */
void WorldModel::resampleParticlesAgent( int iLeft )
{
  for ( int i = iLeft; i < iNrParticlesAgent; i ++ )
    particlesPosAgent[ i ] = particlesPosAgent[ (int)(drand48()*iLeft) ];
}

/*! This method calculates the different state information of the agent, that
    is the global position, global velocity and global neck angle using the
    available information in the world model. This method uses a weighted
    average of all currently perceived flags.
    \param posGlobal will be filled with global position of the agent
    \param velGlobal will be filled with global velocity of the agent
    \param angGlobal will be filled with global neck angle of the agent
    \return global position of the agent */
bool WorldModel::calculateStateAgent2( VecPosition *posGlobal,
                            VecPosition *velGlobal, AngDeg *angGlobal)
{
  double      x=0.0, y=0.0, dMinRadius1, dMaxRadius1, dMinRadius2, dMaxRadius2;
  double      dTotalVar = UnknownDoubleValue, dVar, K;
  int         iIndex1, iIndex2;
  ObjectT     obj2;
  VecPosition pos;

  // for all flags that were perceived in last see message
  for( ObjectT obj1 = iterateObjectStart( iIndex1, OBJECT_SET_FLAGS, 1.0 );
       obj1 != OBJECT_ILLEGAL;
       obj1 = iterateObjectNext ( iIndex1, OBJECT_SET_FLAGS, 1.0 ) )
  {
    // calculate global position with all other flags using two flags
    iIndex2 = iIndex1;
    for( obj2 = iterateObjectNext ( iIndex2, OBJECT_SET_FLAGS, 1.0 ) ;
         obj2 != OBJECT_ILLEGAL;
         obj2 = iterateObjectNext ( iIndex2, OBJECT_SET_FLAGS, 1.0 ) )
    {
      // calculate the position using the two flags
      pos = calculatePosAgentWith2Flags( obj1, obj2 );

      // get distance range from which perceived value can originate from
      // calculate variance (=weighted factor) based on the distance to the
```

```
    // two flags, use variance corresponding to uniform distribution
    // this is not completely correct, better would be to use the
    // intersection area size of the two circle, but is too computational
    // intensive
    getMinMaxDistQuantizeValue(getRelativeDistance(obj1),
            &dMinRadius1, &dMaxRadius1, SS->getQuantizeStepL(), 0.1 )  ;
    getMinMaxDistQuantizeValue(getRelativeDistance(obj2),
            &dMinRadius2, &dMaxRadius2, SS->getQuantizeStepL(), 0.1 )  ;
    dVar =  (dMaxRadius1-dMinRadius1)*(dMaxRadius1-dMinRadius1)/12;
    dVar += (dMaxRadius2-dMinRadius2)*(dMaxRadius2-dMinRadius2)/12;

    if( pos.getX() != UnknownDoubleValue &&
       dTotalVar  == UnknownDoubleValue )
    {
      dTotalVar = dVar;                  // initialize the position
      x         = pos.getX();
      y         = pos.getY();
    }
    else if( pos.getX() != UnknownDoubleValue )
    {
      K = dTotalVar / ( dVar + dTotalVar ); // otherwise use new position
      x += K*( pos.getX() - x );            // based on weighted variance
      y += K*( pos.getY() - y );
      dTotalVar -= K*dTotalVar;
    }
   }
   iterateObjectDone( iIndex2 );
  }
  iterateObjectDone( iIndex1 );
  posGlobal->setVecPosition( x, y );

  // now calculate global position (experiments show best results are obtained
  // when average with all perceived flags is taken).
  *angGlobal = calculateAngleAgentWithPos( *posGlobal );

  // update velocity since after 'see' we have a better estimate of neck angle
  *velGlobal = agentObject.getSpeedRelToNeck().rotate(*angGlobal);

  return true;
}

/*! This method calculates the different state information of the
    agent, that is the global position, global velocity and global
    neck angle using the available information in the world
    model. Only the closest two global flags are taken into account.
```

```
    \param posGlobal will be filled with global position of the agent
    \param velGlobal will be filled with global velocity of the agent
    \param angGlobal will be filled with global neck angle of the agent
    \return global position of the agent */
bool WorldModel::calculateStateAgent3( VecPosition *posGlobal,
                           VecPosition *velGlobal, AngDeg *angGlobal )
{
  // first determine the two closest flags
  ObjectT objFlag1 = getClosestRelativeInSet( OBJECT_SET_FLAGS );
  ObjectT objFlag2 = getSecondClosestRelativeInSet( OBJECT_SET_FLAGS );
  ObjectT objLine = getFurthestRelativeInSet( OBJECT_SET_LINES );

  // first determine global neck angle with furthest line (max. error is 0.5)
  if( objLine != OBJECT_ILLEGAL )
  {
    double angGlobalLine = getGlobalAngle  ( objLine );
    AngDeg angRelLine    = getRelativeAngle( objLine );
    *angGlobal           = angGlobalLine - angRelLine;
    *angGlobal           = VecPosition::normalizeAngle( *angGlobal );
  }

  // if two flags were seen in last see message
  //   calculate global position using two closest flags (Cosinus rule)
  //     except when rel angle flags is smaller than 8 (gives bad results)
  //       except when no line is seen
  if( objFlag1 != OBJECT_ILLEGAL && objFlag2 != OBJECT_ILLEGAL &&
      (
        fabs(getRelativeAngle(objFlag1) - getRelativeAngle(objFlag2)) > 8.0
        || objLine == OBJECT_ILLEGAL
      ) )
  {
    *posGlobal = calculatePosAgentWith2Flags( objFlag1, objFlag2 );
  }
  else if( objFlag1 != OBJECT_ILLEGAL && objLine != OBJECT_ILLEGAL )
  {
    // calculate global position as follows:
    //   - get the global position of the closest flag
    //   - get the rel vector to this flag using global neck angle
    //   - global position = global pos flag - relative vector

    VecPosition posGlobalFlag = getGlobalPosition( objFlag1 );
    VecPosition relPosFlag    = VecPosition::getVecPositionFromPolar(
      getRelativeDistance( objFlag1 ),
      *angGlobal + getRelativeAngle( objFlag1 ) );
    *posGlobal                = posGlobalFlag - relPosFlag;
  }
```

```
  else
  {
    cout << "(WorldModel::calculateStateAgent3) cannot determine pos in cycle "
         << getCurrentCycle() << endl;
    return false;
  }

  // calculate global velocity using the absolute neck angle
  *velGlobal = agentObject.getSpeedRelToNeck().rotate(*angGlobal);

  return true;
}

/*! This method calculates the global position of the agent using two flags.
    Using the perceived distance to the two flag, two circles are created with
    as center the global position of the flag. The intersection of these two
    circles is returned as the global position of the agent (note that the
    relative direction to both flags can be used to determine which of the
    two possible intersection points is the correct position).
    It is assumed that the relative information of the specified flags are
    from the last see message.
    \param objFlag1 object type of first flag
    \param objFlag2 object type of second flag
    \return global position of the agent */
VecPosition WorldModel::calculatePosAgentWith2Flags( ObjectT objFlag1,
                                     ObjectT objFlag2 )
{
    double      xA, yA, xB, yB, rA, rB;
    AngDeg      aA, aB;

    // get all information of the two flags
    xA = getGlobalPosition  ( objFlag1 ).getX();
    yA = getGlobalPosition  ( objFlag1 ).getY();
    rA = getRelativeDistance( objFlag1 );
    aA = getRelativeAngle   ( objFlag1 );
    xB = getGlobalPosition  ( objFlag2 ).getX();
    yB = getGlobalPosition  ( objFlag2 ).getY();
    rB = getRelativeDistance( objFlag2 );
    aB = getRelativeAngle   ( objFlag2 );

    double      L, dx, dy, d_par, d_orth;
    double      x, y;
    // Sign is like this 'because' y-axis increases from top to bottom
    double sign = ((aB - aA) > 0.0) ? 1.0 : -1.0;

    // From Cosinus rule: rB rB = L L + rA rA - 2 L rA cos(aB)
```

```
    // with:            rA cos(aB) = d_par
    // and:             d_par^2 + d_orth^2 = rA^2
    // Finally:
    // Position from landmark position and vectors parallel and orthogonal
    // to line from landmark A to B

    dx = xB - xA;
    dy = yB - yA;
    L = sqrt(dx*dx + dy*dy);                // distance between two flags

    dx /= L; dy /= L;                       // normalize

    d_par = (L*L + rA*rA - rB*rB) / (2.0 * L); // dist from flag1 to orth proj
    double arg = rA*rA - d_par*d_par;
    d_orth = (arg > 0.0) ? sqrt(arg) : 0.0;

    x = xA + d_par * dx - sign * d_orth * dy;
    y = yA + d_par * dy + sign * d_orth * dx;

    return VecPosition( x, y );
}

/*! This method calculates the global neck angle of the agent using
    all available flag information and the estimation of the current
    global position of the agent 'pos'. For each perceived flag, the
    global direction is calculated using the known global position of
    the flag and the specified global position 'pos' of the
    agent. Then the global neck direction of the agent is determined
    using the relative direction to this flag.  The average neck angle
    of all perceived flags is returned.

    \param pos current global position of the agent
    \return global neck angle of the agent */
AngDeg WorldModel::calculateAngleAgentWithPos( VecPosition pos )
{
  int    iNrFlags = 0, iIndex;
  double dCosX=0, dSinX=0 ,dAngleNow, xA, yA, aA;

  for( ObjectT obj = iterateObjectStart( iIndex, OBJECT_SET_FLAGS, 1.0 );
       obj != OBJECT_ILLEGAL;
       obj = iterateObjectNext ( iIndex, OBJECT_SET_FLAGS, 1.0 ) )
  {
    xA = getGlobalPosition( obj ).getX();
    yA = getGlobalPosition( obj ).getY();
    aA = getRelativeAngle( obj );
```

```
    // calculate global direction between flag and agent
    // calculate global neck angle by subtracting relative angle to flag
    dAngleNow = atan2Deg( yA - pos.getY(), xA - pos.getX() );
    dAngleNow = VecPosition::normalizeAngle( dAngleNow - aA );

    // add cosine part of angle and sine part separately; this to avoid
    // boundary problem when computing average angle (average of -176 and
    // 178 equals -179 and not 1).
    dCosX += cosDeg( dAngleNow );
    dSinX += sinDeg( dAngleNow );
    iNrFlags++;

  }
  iterateObjectDone( iIndex );

  // calculate average cosine and sine part and determine corresponding angle
  dCosX /= (double)iNrFlags;
  dSinX /= (double)iNrFlags;
  if( iNrFlags == 0 )
    return UnknownAngleValue;

  return VecPosition::normalizeAngle( atan2Deg( dSinX, dCosX  ) )  ;
}


/*! This method returns the velocity from the object o given the perceptions
    from the see message. It uses the soccer server formula directly
    (thus assuming no noise).
    \param o object type for which velocity is determined
    \return global velocity of the ball */
VecPosition WorldModel::calculateVelocityDynamicObject( ObjectT o )
{
  DynamicObject * dobj = (DynamicObject*) getObjectPtrFromType( o );
  if( dobj == NULL )
    return VecPosition( UnknownDoubleValue, UnknownDoubleValue );
  double dDistCh = dobj->getRelativeDistanceChange(   );
  double angCh   = dobj->getRelativeAngleChange   (   );
  double dDist   = getRelativeDistance            ( o );
  double ang     = getRelativeAngle               ( o );

  double velx = dDistCh * cosDeg(ang) - Deg2Rad(angCh) * dDist * sinDeg( ang );
  double vely = dDistCh * sinDeg(ang) + Deg2Rad(angCh) * dDist * cosDeg( ang );

  VecPosition vel( velx, vely );
  return vel.relativeToGlobal( getAgentGlobalVelocity(),
                               getAgentGlobalNeckAngle() );
```

```
}


/*! This method calculates the global position and velocity of the ball using
    the newest visual information.
    \param *posGlobal will be filled with the global position
    \param *velGlobal will be filled with the global velocity
    \return true when calculations were succesful, false otherwise */
bool WorldModel::calculateStateBall( VecPosition *posGlobal,
                                     VecPosition *velGlobal )
{
  // set the global position of the ball as follows:
  //  - get the relative position from the agent to it in world-axis
  //  - add global position agent to this relative position
  VecPosition posRelWorld =
      VecPosition( getRelativeDistance( OBJECT_BALL ),
                   getRelativeAngle( OBJECT_BALL ) + getAgentGlobalNeckAngle(),
                   POLAR );
  *posGlobal = getAgentGlobalPosition() + posRelWorld;

  if( isBeforeKickOff() )
  {
    posGlobal->setVecPosition( 0, 0 );
    velGlobal->setVecPosition( 0, 0 );
    return true;
  }

  *velGlobal = getGlobalVelocity(OBJECT_BALL);
  if( Ball.getTimeChangeInformation( ) == getTimeLastSeen( OBJECT_BALL ) )
  {
    *velGlobal = calculateVelocityDynamicObject( OBJECT_BALL );;
    Log.log( 458, "vel based on change info: (%f,%f)", velGlobal->getX(),
                  velGlobal->getY() );
    // average result with predicted velocity from last see message
    // this has the best result for the ball: see program absvelocity.C
    // only use average when no big chance (kick,turn or dash) has occured
    if( fabs(velGlobal->getX() - getGlobalVelocity(OBJECT_BALL).getX())
                                 <= 2*SS->getBallRand()*getBallSpeed() &&
        fabs(velGlobal->getY() - getGlobalVelocity(OBJECT_BALL).getY())
                                 <= 2*SS->getBallRand()*getBallSpeed() )
    {
      *velGlobal = (*velGlobal + getGlobalVelocity(OBJECT_BALL))/2.0;
      Log.log( 458, "average ball vel to (%f,%f)", velGlobal->getX(),
                    velGlobal->getY() );
    }
  }
```

```
else if( getRelativeDistance(OBJECT_BALL) < SS->getVisibleDistance() &&
        getTimeLastSeeMessage() - Ball.getTimeGlobalPosDerivedFromSee() < 3)
{
  // no change information but have got feel info -> use position based vel.
  // get the difference with the previous known global position
  // and subtract the position difference (this difference is caused by
  // the fact that the global position calculation contains a lot of noise
  // now the noise is filtered, since we compare the velocity as if its
  // position was seen "with the noise" of the last cycle).

  VecPosition posGlobalDiff   = *posGlobal - Ball.getGlobalPositionLastSee()
                              - agentObject.getPositionDifference();
  Log.log( 101, "2 pos: ball(%f,%f), ball_prev(%f,%f), agentdiff(%f,%f)",
     posGlobal->getX(), posGlobal->getY(),
     Ball.getGlobalPositionLastSee().getX(),
     Ball.getGlobalPositionLastSee().getY(),
     agentObject.getPositionDifference().getX(),
     agentObject.getPositionDifference().getY() );

  // difference in global positions is distance traveled, we have to make
  // distinction whether this distance is traveled in one or two cycles.
  // 1 cycle:
  //   distance difference equals last velocity so only have to multiply with
  //   decay
  // 2 cycles:
  //   do not update, since kick can be performed causing large change
  //   of which we have no see information (so thus use sense update)

  if( getTimeLastSeeMessage() - Ball.getTimeGlobalPosDerivedFromSee() == 1 &&
     m_bWasCollision == false )
  {
   *velGlobal = posGlobalDiff*SS->getBallDecay();
   Log.log( 458, "vel based on 2 pos: (%f,%f)", velGlobal->getX(),
                  velGlobal->getY() );
   Log.log( 101, "vel based on 2 pos: (%f,%f)", velGlobal->getX(),
                  velGlobal->getY() );
  }
  else if(getTimeLastSeeMessage()-Ball.getTimeGlobalPosDerivedFromSee()==2 &&
     m_bWasCollision == false )
  {
   VecPosition velTmp, posTmp;
   double      dDist;
   // we have seen ball for the last time two cycles ago
   // name v velocity after first cycle, vel. now is then v*d^2.
   // v can be calculated as v + v * d = diff -> v = diff / ( 1 + d)
```

```
      // velTmp is now distance traveled in previous cycle.
      // get position in previous cycle.
      // if no opponent or I could have shot the ball, update velocity.
      velTmp = (posGlobalDiff/(1+SS->getBallDecay()))*SS->getBallDecay();
      posTmp = *posGlobal - velTmp;
      getClosestInSetTo( OBJECT_SET_PLAYERS, posTmp, &dDist );
      if( dDist > SS->getMaximalKickDist() &&
          m_bPerformedKick == false &&
             (getCurrentTime() - m_timeLastCollision) > 3 )
      {
        *velGlobal = velTmp * SS->getBallDecay();
         Log.log( 458, "vel based on 3 pos: (%f,%f)", velGlobal->getX(),
                       velGlobal->getY() );
      }
     }
     else if( getTimeLastSeeMessage()-Ball.getTimeGlobalPosDerivedFromSee() > 2)
     {
#ifdef WIN32
      Log.log( 20, "(WorldModel:%s) time difference too large" ,
                  "calculateStateBall" );
#else
      Log.log( 20, "(WorldModel:%s) time difference too large" ,__FUNCTION__ );
#endif

     }
   }
   else
    Log.log( 458, "vel ball not updated", velGlobal->getX(),velGlobal->getY());
    // object too far away do not estimate velocity, sense has updated it
    // already

   // change velocity when the ball
   //   has been catched or play mode is not play_on
   //   is in kickable distance opponent
   //   higher than max speed with added noise
   // change position when it is a kick_in: know y coordinate
   // change position when it is a back_pass_[rl]: know positions
   if( getTimeSinceLastCatch() < 2 ||
      (getPlayMode() != PM_PLAY_ON && !isGoalKickUs() && !isGoalKickThem() &&
       !isPenaltyUs() && !isPenaltyThem() ))
     velGlobal->setMagnitude( 0.0 );
   else if( getNrInSetInCircle( OBJECT_SET_OPPONENTS,
                        Circle(*posGlobal,SS->getMaximalKickDist())) > 0
          && getRelativeDistance( OBJECT_BALL ) > SS->getMaximalKickDist() )
     velGlobal->setMagnitude( 0.0 );
   else if( velGlobal->getMagnitude() >
```

```
                    ( 1.0 + SS->getBallRand() )*SS->getBallSpeedMax() )
    velGlobal->setMagnitude( SS->getBallSpeedMax() );

  if( isKickInUs() || isKickInThem() )
    posGlobal->setY( sign( posGlobal->getY() ) * PITCH_WIDTH/2.0 );
  else if( isBackPassUs() )
    posGlobal->setVecPosition( - PENALTY_X,
                       sign(posGlobal->getY())*PENALTY_AREA_WIDTH/2.0);
  else if( isBackPassThem() )
    posGlobal->setVecPosition( + PENALTY_X,
                       sign(posGlobal->getY())*PENALTY_AREA_WIDTH/2.0);

  Log.log( 458, "final ball vel: (%f,%f)",velGlobal->getX(),velGlobal->getY());
  if( getRelativeDistance(OBJECT_BALL) < SS->getVisibleDistance() )
    Log.log( 101, "direction old: %f, new: %f",
           ( getGlobalPosition( OBJECT_BALL ) -
             getAgentGlobalPosition()).getDirection(),
        ( *posGlobal - getAgentGlobalPosition()).getDirection() );
  return true;
}



/*! This method determines the current state of a player and is called after
    a see message has arrived.
    \param o object type of the player
    \param *posGlobal will be filled with global position of the player
    \param *velGlobal will be filled with global velocity of the player
    \return bool indicating whether calculations succeeded. */
bool WorldModel::calculateStatePlayer( ObjectT o, VecPosition *posGlobal,
                               VecPosition *velGlobal )
{
  PlayerObject *pob = (PlayerObject*) getObjectPtrFromType( o );
  if( pob == NULL )
    return false;

  // set the global position of this dynamic object as follows:
  //  - get the relative position from the agent to it in world-axis
  //  - add global position agent to this relative position
  VecPosition posRelWorld =
     VecPosition( getRelativeDistance( o ),
                  getRelativeAngle( o ) + agentObject.getGlobalNeckAngle(),
                  POLAR );
  *posGlobal = getAgentGlobalPosition() + posRelWorld;

  *velGlobal = getGlobalVelocity( o ) ;
```

```
  if( pob->getTimeChangeInformation( ) == getTimeLastSeen( o ) )
  {
    // calculate the global velocity using the distance and angle change
    // with the formula from the soccermanual
    *velGlobal = calculateVelocityDynamicObject( o );
  }
  else
     ; // object too far away do not estimate velocity, sense has updated it
       // already and does not really matter then

  if( velGlobal->getMagnitude() >=
                  ( 1.0 + SS->getPlayerRand())*SS->getPlayerSpeedMax() )
    velGlobal->setMagnitude( SS->getPlayerSpeedMax() );

  return true;
}


/*! This method determines the minimum and maximum input values that
    will produce a quantized (noise used in the soccer server)
    distance of 'dOutput'.  With other words, this is the range of
    values that will have the same quantized value 'dOutput'. The
    quantized steps are defined by x1 and x2.  See the soccer server
    manual for details.

    \param dOutput resulting quantized value
    \param *dMin will be filled with minimum possible value
    \param *dMax will be filled with maximum possible value
    \param x1 value of inner quantize call-0.1 for player/ball, 0.01 for flags
    \param x2 value of outer quantize call (normally 0.1)
    \return bool indicating whether values were filled correctly */
bool WorldModel::getMinMaxDistQuantizeValue( double dOutput, double *dMin,
                                 double *dMax,   double x1, double x2 )
{
  // change output a little bit to circumvent boundaries
  // q = quantize(e^(quantize(ln(V),x1)),x2) with quantize(V,Q) = rint(V/Q)*Q
  // e^(quantize(ln(V),x1)_min = invQuantize( q, x2 )
  // quantize(ln(V),x1) = ln ( invQuantize( q, x2 ) )
  // ln(V) = invQuantize( ln ( invQuantize( q, x2 ) ), x1 )
  // V_min = e^( invQuantize( ln ( invQuantize( q, x2 ) ), x1 ) )
  // apply inverse quantize twice to get correct value
  dOutput -= 1.0e-10;
  if( dOutput < 0 )
    *dMin = 0.0;
  else
    *dMin = exp( invQuantizeMin( log( invQuantizeMin(dOutput,x2) ), x1 ) );
```

```
  dOutput += 2.0e-10;
  *dMax = exp( invQuantizeMax( log( invQuantizeMax(dOutput,x2) ), x1 ) );
  return true;
}

/*! This method determines the minimum and maximum input values that will
    produce a quantized (noise used in the soccer server) for the direction
    change of 'dOutput'.
    See the soccer server manual for details.
    \param dOutput resulting quantized value for direction change
    \param *dMin will be filled with minimum possible value
    \param *dMax will be filled with maximum possible value
    \param x1 value of outer quantize call (normally 0.1)
    \return bool indicating whether values were filled correctly */
bool WorldModel::getMinMaxDirChange( double dOutput, double *dMin,
                                     double *dMax,   double x1     )
{
 *dMin = invQuantizeMin( dOutput, x1 ) ;
 *dMax = invQuantizeMax( dOutput, x1 ) ;
 return true;
}

/*! This method determines the minimum and maximum input values that will
    produce a quantized (noise used in the soccer server) for the distance
    change of 'dOutput'.
    See the soccer server manual for details.
    \param dOutput resulting quantized value for direction change
    \param dDist distance to the perceived object
    \param *dMin will be filled with minimum possible value
    \param *dMax will be filled with maximum possible value
    \param x1 quantize step for distance change
    \param xDist1 value of inner quantize call to determine quantize value
                  corresponding to distance
    \param xDist2 value of outer quantize call to determine quantize value
                  corresponding to distance
    \return bool indicating whether values were filled correctly */
bool WorldModel::getMinMaxDistChange( double dOutput, double dDist,
        double *dMin, double *dMax, double x1, double xDist1, double xDist2)
{
  // Q_dist = quantize(e^(quantize(ln(V),xDist1)),xDist2)
  // q = Q_dist * Quantize( distance_change/distance, x1 )
  // dOutput = q/Q_dist = Quantize( distance_change/distance, x1 )
  // (distance_change/distance)_min = invQmin(q/Q_dist, x1 )
  // real distance is not know so should take into account distance range
  double dMinDist, dMaxDist;
  getMinMaxDistQuantizeValue( dDist, &dMinDist, &dMaxDist, xDist1, xDist2 );
```

```
  dOutput = dOutput/dDist;
  double dMinCh = invQuantizeMin( dOutput, x1 ) ;
  double dMaxCh = invQuantizeMax( dOutput, x1 ) ;
  *dMin = min( dMinDist*dMinCh, dMaxDist*dMinCh );
  *dMax = max( dMinDist*dMaxCh, dMaxDist*dMaxCh );
  return true;
}

/*! This method returns the minimum value that generates dOutput as a quantized
    value when 'dQuantizeStep' is used as the quantized step.
    \param dOutput quantized output
    \param dQuantizeStep quantize step
    \return minimum value that when quantized produces 'dOutput' */
double WorldModel::invQuantizeMin( double dOutput, double dQuantizeStep )
{
  // q = quantize( V, Q ) = rint(V/Q)*Q -> q/Q = rint( V/Q)
  // min = rint(q/Q)-0.5 = V_min/Q -> V_min = (rint(q/Q)-0.5)*Q
  return max(1.0e-10,(rint( dOutput / dQuantizeStep )-0.5 )*dQuantizeStep);
}

/*! This method returns the maximum value that generates dOutput as a quantized
    value when 'dQuantizeStep' is used as the quantized step.
    \param dOutput quantized output
    \param dQuantizeStep quantize step
    \return maximum value that when quantized produces 'dOutput' */
double WorldModel::invQuantizeMax( double dOutput, double dQuantizeStep )
{
  // q = quantize( V, Q ) = rint(V/Q)*Q -> q/Q = rint( V/Q)
  // max = rint(q/Q)+0.5 = V_max/Q -> V_max = (rint(q/Q)-0.5)*Q
  return (rint( dOutput/dQuantizeStep) + 0.5 )*dQuantizeStep;
}

/*! This method maps the information in the array of unknown players
    (players of which we do not know the number and/or team) to the
    player information located in the WorldModel. This is done by
    comparing the predicted position of players we haven't seen this
    cycle and the information of players of which we do not have the
    number and/or teamname.If this difference in the distance is
    smaller than tolerated (see
    PlayerSettings::getPlayerDistTolerance()), this player information
    is updated with the specified information.

    \param time time of the current cycle */
void WorldModel::mapUnknownPlayers( Time time)
{
  double      dDist, dMinDist;
```

```
VecPosition pos, posAgent = getAgentGlobalPosition();
ObjectT     o, o_new, objTmp;
int         index;

// for all unknown players, try to map it to closest teammate or opponent
for( int j = 0; j < iNrUnknownPlayers; j ++ )
{
 pos = posAgent + VecPosition( UnknownPlayers[j].getRelativeDistance(),
         VecPosition::normalizeAngle( getAgentGlobalNeckAngle() +
                                      UnknownPlayers[j].getRelativeAngle() ),
                                    POLAR );
 dMinDist = 1000.0;
 o     = UnknownPlayers[j].getType();
 o_new = OBJECT_ILLEGAL;
 Log.log( 464, "map unknown player: %d %f %f (%f,%f) neck %f",
             o,
             UnknownPlayers[j].getRelativeDistance(),
             UnknownPlayers[j].getRelativeAngle(),
             pos.getX(), pos.getY(),
             getAgentGlobalNeckAngle() );
 if( ! SoccerTypes::isOpponent( o ) ) // TEAMMATE_UNKNOWN or PLAYER_UNKNOWN
 {
  for( int i = 0 ; i < MAX_TEAMMATES ; i++ )
  {
   objTmp = Teammates[i].getType();
   if( isConfidenceGood( objTmp ) && getTimeLastSeen( objTmp ) != time &&
       objTmp != getAgentObjectType()  )
   {
    dDist = pos.getDistanceTo( Teammates[i].getGlobalPosition( ) );
    Log.log( 464, "distance with %d %f (%f,%f)", objTmp, dDist,
               Teammates[i].getRelativeDistance(),
               Teammates[i].getRelativeAngle()  );
    if( dDist < dMinDist )
    {
     o_new    = objTmp;
     dMinDist = dDist;
    }
   }
   else
    Log.log( 464, "not possible: distance with %d (%f,%f) conf %d  \
                  last_seen (%d,%d), now (%d,%d)",
             objTmp,
             Teammates[i].getRelativeDistance(),
             Teammates[i].getRelativeAngle(),
             isConfidenceGood( objTmp ),
             getTimeLastSeen( objTmp ).getTime(),
```

```
            getTimeLastSeen( objTmp ).getTimeStopped(),
            time.getTime(),
            time.getTimeStopped());

  }
 }
 if( ! SoccerTypes::isTeammate( o ) ) // OPPONENT_UNKNOWN or PLAYER_UNKNOWN
 {
  for( int i = 0 ; i < MAX_OPPONENTS ; i++ )
  {
   objTmp = Opponents[i].getType();
   if( isConfidenceGood( objTmp ) && getTimeLastSeen( objTmp ) != time )
   {
    dDist = pos.getDistanceTo( Opponents[i].getGlobalPosition( ) );
    Log.log( 464, "distance with %d %f (%f,%f)", objTmp, dDist,
             Opponents[i].getRelativeDistance(),
             Opponents[i].getRelativeAngle()  );
    if( dDist < dMinDist )
    {
     o_new    = objTmp;
     dMinDist = dDist;
    }
   }
   else
    Log.log( 464, "not possible: distance with %d (%f,%f) conf %d  \
                last_seen (%d,%d), now (%d,%d)",
            objTmp,
            Opponents[i].getRelativeDistance(),
            Opponents[i].getRelativeAngle(),
            isConfidenceGood( objTmp ),
            getTimeLastSeen( objTmp ).getTime(),
            getTimeLastSeen( objTmp ).getTimeStopped(),
            time.getTime(),
            time.getTimeStopped());
  }
 }

 Log.log( 464, "closest obj %d, found %f, max_move %f rel_dist %f type %d",
        o_new, dMinDist, getMaxTraveledDistance( o_new ),
        UnknownPlayers[j].getRelativeDistance(),
        UnknownPlayers[j].getType() );

 // if player found and in tolerated distance update player
 // information.  else if not mapped to player put information in
 // first player position of which we have no info.  if we do not
 // know it is a teammate or opponent only assume opponent when it
```

```
// is very close since then it is probably an opponent trying to
// get ball
if( SoccerTypes::isKnownPlayer(o_new)
    && dMinDist < PS->getPlayerDistTolerance()
    && dMinDist < getMaxTraveledDistance( o_new ) + 2 ) // 2 for the noise
{
  UnknownPlayers[j].setType( o_new );
  if( SoccerTypes::isTeammate( o_new ) )
  {
    index = SoccerTypes::getIndex(o_new);
    UnknownPlayers[j].setHeteroPlayerType(
                        Teammates[index].getHeteroPlayerType( ) );
    Teammates[index] = UnknownPlayers[j];
    Log.log( 464, "map to known teammate %d (%f,%f) conf %f", index + 1,
       Teammates[index].getGlobalPosition().getX(),
       Teammates[index].getGlobalPosition().getY(),
       Teammates[index].getConfidence( getCurrentTime() ) );
  }
  else if( SoccerTypes::isOpponent( o_new ) )
  {
    index = SoccerTypes::getIndex(o_new );
    UnknownPlayers[j].setHeteroPlayerType(
                        Opponents[index].getHeteroPlayerType( ) );
    Opponents[index] = UnknownPlayers[j];
    Log.log( 464, "map to known opponent %d (time %d)",
                  index + 1, UnknownPlayers[j].getTimeLastSeen().getTime() );
  }
}
else if( UnknownPlayers[j].getType() == OBJECT_TEAMMATE_UNKNOWN )
{
  o_new = getFirstEmptySpotInSet( OBJECT_SET_TEAMMATES, j );

  if( o_new != OBJECT_ILLEGAL )
  {
    index = SoccerTypes::getIndex(o_new);
    UnknownPlayers[j].setHeteroPlayerType(
                        Teammates[index].getHeteroPlayerType( ) );
    UnknownPlayers[j].setType( o_new );
    Teammates[index] = UnknownPlayers[j];
    Log.log( 464, "map to unknown teammate %d", index + 1 );
  }
}
else if( UnknownPlayers[j].getType() == OBJECT_OPPONENT_UNKNOWN )
{                                  // could not map info to a player
  o_new = getFirstEmptySpotInSet( OBJECT_SET_OPPONENTS, j );
  Log.log( 464, "map unkown opponent to %d", o_new );
```

```
      if( o_new != OBJECT_ILLEGAL )
      {
       index = SoccerTypes::getIndex(o_new);
       UnknownPlayers[j].setHeteroPlayerType(
                          Opponents[index].getHeteroPlayerType( ) );
       UnknownPlayers[j].setType( o_new );
       Opponents[index] = UnknownPlayers[j];
       Log.log( 464, "map to unknown opponent %d", index + 1 );
      }
      else
      {
       Log.log( 464, "couldn't find empty spot for unk. opponent" );
       if( Log.isInLogLevel( 464 ) )
        show( OBJECT_SET_PLAYERS, Log.getOutputStream() );
//       if( getRelativeDistance( OBJECT_BALL ) < 3.0 )
//       cerr << getPlayerNumber() << ", " <<
//         getCurrentCycle() << " couldn't find empty spot" << endl;
      }
    }
    else if( UnknownPlayers[j].getType() == OBJECT_PLAYER_UNKNOWN &&
        ( UnknownPlayers[j].getRelativeDistance() < SS->getVisibleDistance()
         || ( dMinDist - getMaxTraveledDistance( o_new ) ) > 10   ) )
    {
     o_new = getFirstEmptySpotInSet( OBJECT_SET_OPPONENTS );
     Log.log( 464, "map unkown player to %d", o_new );
     if( o_new != OBJECT_ILLEGAL )
     {
       index = SoccerTypes::getIndex(o_new);
       UnknownPlayers[j].setHeteroPlayerType(
                          Opponents[index].getHeteroPlayerType( ) );
       UnknownPlayers[j].setType( o_new );
       Opponents[index] = UnknownPlayers[j];
       Log.log( 464, "map to unknown close opponent %d", index + 1 );
     }
    }
   }
   Log.log( 464, "end map unknown player" );

   iNrUnknownPlayers  = 0;
}

/*! This method updates the ServerSettings of this agent using the player type
    information of the heterogeneous player at index 'iIndex'. This method is
    usually called when the player type of the agent is changed by the coach.
    It updates its parameters, such that all calculations to determine the
    next action are based on the correct parameters.
```

```
    \param iIndex index of the new player type
    \return bool indicating whether update was succesfull. */
bool WorldModel::updateSSToHeteroPlayerType( int iIndex )
{
  SS->setPlayerSpeedMax( pt[iIndex].dPlayerSpeedMax );
  SS->setStaminaIncMax ( pt[iIndex].dStaminaIncMax  );
  SS->setPlayerDecay   ( pt[iIndex].dPlayerDecay    );
  SS->setInertiaMoment ( pt[iIndex].dInertiaMoment  );
  SS->setDashPowerRate ( pt[iIndex].dDashPowerRate  );
  SS->setPlayerSize    ( pt[iIndex].dPlayerSize     );
  SS->setKickableMargin( pt[iIndex].dKickableMargin );
  SS->setKickRand      ( pt[iIndex].dKickRand       );
  SS->setExtraStamina  ( pt[iIndex].dExtraStamina   );
  SS->setEffortMax     ( pt[iIndex].dEffortMax      );
  SS->setEffortMin     ( pt[iIndex].dEffortMin      );

  return true;
}

/*! This methods resets all the object information stored in the worldmodel.
    This means that the last see time of these objects are set to UnknownTime.
    \return bool indicating whether update was succesfull */
bool WorldModel::resetTimeObjects( )
{
  Ball.setTimeLastSeen             ( Time( -1, 0) );
  for( int i = 0 ; i < MAX_TEAMMATES ; i ++ )
    Teammates[i].setTimeLastSeen   ( Time( -1, 0) );
  for( int i = 0 ; i < MAX_OPPONENTS ; i ++ )
    Opponents[i].setTimeLastSeen   ( Time( -1, 0) );
  for( int i = 0 ; i < MAX_FLAGS     ; i ++ )
    Flags[i].setTimeLastSeen   ( Time( -1, 0) );
  for( int i = 0 ; i < MAX_LINES     ; i ++ )
    Lines[i].setTimeLastSeen   ( Time( -1, 0) );
  agentObject.setTimeLastSeen      ( Time( -1, 0) );
  return true;
}

/*! This method removes ghosts from the WorldModel. Ghosts are objects that
    should have been seen in the last see message but aren't. Currently only
    the ball is removed from the WorldMOdel. */
void WorldModel::removeGhosts( )
{
  AngDeg dAngle=SoccerTypes::getHalfViewAngleValue(agentObject.getViewAngle());
  dAngle -= 0.35*dAngle; // make somewhat smaller for error
  VecPosition posAgent = getAgentGlobalPosition();
```

```
if( fabs( getRelativeAngle( OBJECT_BALL ) ) < dAngle
    && getTimeLastSeen( OBJECT_BALL ) != getTimeLastSeeMessage() )
{
 double dDist;
 ObjectT objOpp=getClosestInSetTo(OBJECT_SET_OPPONENTS,OBJECT_BALL,&dDist);
 // when opp is very close and ball is not seen (and also not felt) set it
 // at the opponent position
 if( dDist < 2.0 &&
       posAgent.getDistanceTo( getGlobalPosition( objOpp ) ) < 4.5  &&
       getTimeLastSeen( objOpp ) != getTimeLastSeeMessage() )
 {
  Log.log( 556, "ball not seen, but opp close, set ball to that pos" );
  Ball.setGlobalPosition( getGlobalPosition( objOpp ),
                          Ball.getTimeGlobalPosition() );
 }
 else
 {
  Log.log( 556, "ball not in cone: set time ball at -1 %f %f",
           fabs( getRelativeAngle( OBJECT_BALL ) ), dAngle );
  Ball.setTimeLastSeen( Time( -1, 0 ) );
 }
}

// ball should be "seen" when it is in visible distance. 0.9 is for noise
if( fabs( getRelativeDistance( OBJECT_BALL ) ) < 0.9*SS->getVisibleDistance()
    && getTimeLastSeen( OBJECT_BALL ) != getTimeLastSeeMessage() )
{
 double dDist;
 ObjectT objOpp=getClosestInSetTo(OBJECT_SET_OPPONENTS,OBJECT_BALL,&dDist);
 if( dDist < 2.0 )
 {
  Log.log( 556, "ball not seen, but opp close, set ball to that pos" );
  Ball.setGlobalPosition( getGlobalPosition( objOpp ),
                          Ball.getTimeGlobalPosition() );
 }
 else
 {
  Log.log( 556, "ball not in vis. dist: set time ball at -1 %f %f",
           fabs( getRelativeAngle( OBJECT_BALL ) ), dAngle );
  Ball.setTimeLastSeen( Time( -1, 0 ) );
 }
}

// now remove all opponents. If an opponent is not seen while you expect him
// somewhere, he has moved a lot. It's no use keeping his current position
// in the world model, since he would still be considered in passing
```

```
  // confidence, etc.
  int iIndex;
  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_OPPONENTS );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_OPPONENTS ) )
  {
    if(  fabs( getRelativeAngle( o ) ) < dAngle
//         || getRelativeDistance( o ) < SS->getVisibleDistance() )
        && getTimeLastSeen( o ) != getTimeLastSeeMessage() &&
        o != getOppGoalieType() )
    {
      Log.log( 556, "opponent %d not in cone: (angle %f>%f) see (%d,%d)",
                SoccerTypes::getIndex( o ) + 1,
              fabs( getRelativeAngle( o ) ), dAngle,
              getTimeLastSeeMessage().getTime(),getTimeLastSeen(o).getTime());

      AngDeg ang =
        SoccerTypes::getHalfViewAngleValue(agentObject.getViewAngle());
      AngDeg angOpp = (getGlobalPosition(o)-posAgent).getDirection();
      AngDeg angNeck = getAgentGlobalNeckAngle();
      // could have move both ways
      if( fabs(VecPosition::normalizeAngle( angOpp - angNeck )) < 10 &&
          getCurrentTime() - getTimeGlobalAngles(o) < 4 )
        angOpp = (getGlobalPosition(o)+
                  VecPosition(2.0, getGlobalBodyAngle(o), POLAR )-posAgent)
          .getDirection();
      if( VecPosition::normalizeAngle( angOpp - angNeck )  > 0 )
        angOpp = angNeck + ang;
      else
        angOpp = angNeck - ang;
      angOpp = VecPosition::normalizeAngle( angOpp );
      DynamicObject *obj = (DynamicObject*) getObjectPtrFromType( o );

      // if at edge change him otherwise delete
      if( fabs( getRelativeAngle( o ) ) + 20 > dAngle )
      {
        VecPosition posNew = posAgent+
          VecPosition(getRelativeDistance(o),angOpp, POLAR);
        if( posNew.getDistanceTo( getGlobalPosition( o ) ) <
            getMaxTraveledDistance( o ) )
        {
          obj->setGlobalPosition( posNew, getCurrentTime());
            //          obj->setGlobalPosition( posNew, getTimeLastSeen(o));
          Log.log( 556, "set opp at angle %f", angOpp );
          updateObjectRelativeFromGlobal( o );
        }
```

```
        else
          Log.log( 556, "do not reposition opp too far dist %f",
                posNew.getDistanceTo( getGlobalPosition( o ) ));
      }
      else
      {
          setTimeLastSeen( o, Time( -1, 0 ) );
          Log.log( 556, "remove opponent not in cone at angle %f %f %f", angOpp,
                fabs(getRelativeAngle(o)), dAngle  );
      }
    }
    if( getRelativeDistance(o) < SS->getVisibleDistance( )
          && getTimeLastSeen( o ) != getTimeLastSeeMessage() )
    {
      Log.log( 556, "opp %d not felt, place him outside vis. dist", o );
      DynamicObject *obj = (DynamicObject*) getObjectPtrFromType( o );
      VecPosition posNew = posAgent + VecPosition( SS->getVisibleDistance(),
                                (getGlobalPosition( o ) - getAgentGlobalPosition()).
                                getDirection(), POLAR );
      obj->setGlobalPosition( posNew, getCurrentTime(  ));
      //      obj->setGlobalPosition( posNew, getTimeLastSeen( o ));
      updateObjectRelativeFromGlobal( o );
    }
  }
  iterateObjectDone( iIndex );

  // now remove all teammates. If a teammate is not seen while you expect him
  // somewhere, he has moved a lot. It's no use keeping his current position
  // in the world model, since he would still be considered in passing
  // confidence, etc.
  for( ObjectT o = iterateObjectStart( iIndex, OBJECT_SET_TEAMMATES );
       o != OBJECT_ILLEGAL;
       o = iterateObjectNext ( iIndex, OBJECT_SET_TEAMMATES ) )
  {
    if( fabs( getRelativeAngle( o ) ) < dAngle
//       ||getRelativeDistance( o ) < SS->getVisibleDistance() )
          && getTimeLastSeen( o ) != getTimeLastSeeMessage() )
    {
      Log.log( 556, "teammate %d not in cone: (angle %f>%f) see (%d,%d)",
        SoccerTypes::getIndex( o ) + 1,
        fabs( getRelativeAngle( o ) ), dAngle,
        getTimeLastSeeMessage().getTime(), getTimeLastSeen( o ).getTime() );
      setTimeLastSeen( o, Time( -1, 0 ) );
    }
  }
  iterateObjectDone( iIndex );
```

```
}

/*! This method initializes all particles that represent the global position
    and global velocity of the ball. This is done by taking the last
    perceived information and generate particles within the range that can
    produce the perceived values. Note that each position is related to the
    velocity with the same index. The state of the ball is implicitly
    represented as a 4-tuple (pos_x, pos_y, vel_x, vel_y).
    \param posArray array that will be filled with the position particles
    \param velArray array that will be filled with the velocity particles
    \param iLength number of particles that have to be initialized. */
void WorldModel::initParticlesBall( VecPosition posArray[],
                                    VecPosition velArray[], int iLength )
{
  // declare a bunch of variables
  double dDistBall, dDistChange = UnknownDoubleValue;
  AngDeg angBall, angChange   = UnknownAngleValue;
  double dMinDist, dMaxDist, dMinCh, dMaxCh, dDistTmp, dDistChTmp, dVelX,dVelY;
  AngDeg angChMin, angChMax,     angTmp,   angChTmp;

  // no information  received -> no initialization
  if( Ball.getTimeRelativePosition() != getTimeLastSeeMessage() )
    return;

  // get perceived values for distance and angle with ball
  dDistBall = getRelativeDistance( OBJECT_BALL );
  angBall   = getRelativeAngle( OBJECT_BALL );

  // get perceived values for distance and direction change
  if( Ball.getTimeChangeInformation( ) == getTimeLastSeeMessage() )
  {
    dDistChange = Ball.getRelativeDistanceChange();
    angChange   = Ball.getRelativeAngleChange();
  }

  // get ranges from which values could originate
  getMinMaxDistQuantizeValue( dDistBall, &dMinDist, &dMaxDist, 0.1, 0.1 );
  getMinMaxDistChange( dDistChange, dDistBall, &dMinCh, &dMaxCh, 0.02, 0.1,0.1);
  getMinMaxDirChange ( angChange, &angChMin, &angChMax, 0.1 );

  for( int i = 0; i < iLength; i ++ )
  {
    // make random distance and angle from range (angle is rounded)
    // and make pos
    dDistTmp   = dMinDist + drand48()*fabs(dMaxDist - dMinDist); // angle->sign
```

```
    angTmp     = angBall  + drand48() - 0.5;

    posArray[i].setVecPosition( dDistTmp, angTmp, POLAR );
    posArray[i].relativeToGlobal( getAgentGlobalPosition(),
                          getAgentGlobalNeckAngle() );

    if( dDistChange != UnknownDoubleValue )
    {
      // make random values for direction and distance change and make velocity
      angChTmp   = angChMin + drand48()*(angChMax-angChMin);
      dDistChTmp = dMinCh   + drand48()*(dMaxCh-dMinCh);

      dVelX=dDistChTmp*
        cosDeg(angTmp)-Deg2Rad(angChTmp)*dDistTmp*sinDeg(angTmp);
      dVelY=dDistChTmp*
        sinDeg(angTmp)+Deg2Rad(angChTmp)*dDistTmp*cosDeg(angTmp);

      velArray[i].setVecPosition( dVelX, dVelY );
      velArray[i].relativeToGlobal( getAgentGlobalVelocity(),
                            getAgentGlobalNeckAngle() );
    }
    else
      velArray[i].setVecPosition( 0, 0 );
  }
}

/*! This method checks all particles that represent the global position
    and global velocity of the ball. This is done by using the last
    perceived information. Using the possible ranges from which these values
    coule originate from, it can be checked which particles are legal. These
    particles are moved to the beginning of the array. Afterwards the number of
    legal particles is stored in 'iNrParticlesLeft'.
    \param posArray array that contains the position particles
    \param velArray array that contains the velocity particles
    \param iLength number of particles that have to be checked.
    \param iNrParticlesLeft will contain the number of legal particles*/
 void WorldModel::checkParticlesBall( VecPosition posArray[],
           VecPosition velArray[], int iLength, int *iNrParticlesLeft )
{
  bool   bIllegal;
  double dMinDist, dMaxDist, dMinCh, dMaxCh, dMag;
  double dDistBall, dDistChange = UnknownDoubleValue;
  AngDeg angBall,  angChange;
  double dDistChTmp;
  AngDeg angChTmp, angChMin, angChMax;
  VecPosition pos_rel, vel_rel;
```

```
// no new perceptions, do not check
if( getTimeLastSeen( OBJECT_BALL ) != getTimeLastSeeMessage() )
  return;

// initialize values distance, direction, distance change and
// direction change and get the associated ranges
dDistBall = getRelativeDistance( OBJECT_BALL );
angBall   = getRelativeAngle( OBJECT_BALL );
getMinMaxDistQuantizeValue( dDistBall, &dMinDist, &dMaxDist, 0.1, 0.1 );

if( getTimeLastSeen( OBJECT_BALL ) == Ball.getTimeChangeInformation( ) )
{
  dDistChange = Ball.getRelativeDistanceChange();
  angChange   = Ball.getRelativeAngleChange();
  getMinMaxDirChange ( angChange, &angChMin, &angChMax, 0.1);
  getMinMaxDistChange( dDistChange,dDistBall, &dMinCh, &dMaxCh,0.02,0.1,0.1);

}

*iNrParticlesLeft = 0;
for( int i = 0; i < iLength; i ++ )
{
  // get particles and make them relative to the agent to compare
  pos_rel = posArray[i];
  vel_rel = velArray[i];
  pos_rel.globalToRelative( getAgentGlobalPosition(),
                            getAgentGlobalNeckAngle() );
  vel_rel.globalToRelative( getAgentGlobalVelocity(),
                            getAgentGlobalNeckAngle() );

  bIllegal = false;

  dMag = pos_rel.getMagnitude();
  if( dMag <  dMinDist || dMag > dMaxDist )
  {
    bIllegal = true;
  }
  if( fabs( VecPosition::normalizeAngle(pos_rel.getDirection() - angBall) )
      > 0.5 )
  {
    bIllegal = true;
  }

  if( dDistChange != UnknownDoubleValue )
  {
```

```
      dDistChTmp = (vel_rel.getX()*(pos_rel.getX()/dMag)) +
                   (vel_rel.getY()*(pos_rel.getY()/dMag));
      angChTmp   = Rad2Deg( ((vel_rel.getY()*pos_rel.getX()/dMag) -
                           (vel_rel.getX()*pos_rel.getY()/dMag)))/dMag ;

      if( angChTmp < angChMin || angChTmp > angChMax )
      {
        bIllegal = true;
      }
      if( dDistChTmp < dMinCh || dDistChTmp > dMaxCh )
      {
        bIllegal = true;
      }
    }

    // if not illegal, save particles and raise iNrParticlesLeft
    if( bIllegal == false )
    {
      posArray[*iNrParticlesLeft]     = posArray[i];
      velArray[(*iNrParticlesLeft)++] = velArray[i];
    }
  }
}

/*! This method updates all particles that represent the global position
    and global velocity of the ball to the next cycle. This is done using the
    same formula as the soccer server.
    \param posArray array that contains the position particles
    \param velArray array that contains the velocity particles
    \param iLength number of particles that have to be checked.
    \param dPower power with which the ball is accelerated.
    \param ang angle (relative to body) to which the ball is accelerated. */
void WorldModel::updateParticlesBall( VecPosition posArray[],
             VecPosition velArray[], int iLength, double dPower, AngDeg ang )
{
  double dRand = SS->getBallRand();
  double dMaxRand;

  for( int i = 0; i < iLength; i ++ )
  {
    // if power supplied, assume ball (and thus particles) are kicked
    if( dPower > EPSILON )
    {
      ang = VecPosition::normalizeAngle(ang + getAgentGlobalBodyAngle() );
      velArray[i] += VecPosition(getActualKickPowerRate()*dPower, ang, POLAR) ;
      if( velArray[i].getMagnitude() > SS->getBallSpeedMax() )
```

```
      velArray[i].setMagnitude( SS->getBallSpeedMax() );
    }

    // add noise in same way server does.
    dMaxRand = dRand * velArray[i].getMagnitude();
    velArray[i] += VecPosition(
                (-1 + 2*drand48())*dMaxRand,
                (-1 + 2*drand48())*dMaxRand );
    posArray[i] += velArray[i];
    velArray[i] *= SS->getBallDecay();
  }
}


/*! This method resamples all particles that represent the global position
    and global velocity of the ball. This is by copying existing legal
    particles at random. Since noise will be added to each particle, they will
    blur each time the particles are updated. The first 'iLeft' particles are
    legal particles and copied to the position at the end of the array.
    \param posArray array that contains the position particles
    \param velArray array that contains the velocity particles
    \param iLength number of particles that have to be checked.
    \param iLeft number of particles that are legal */
void WorldModel::resampleParticlesBall( VecPosition posArray[],
VecPosition velArray[], int iLength, int iLeft )
{
  int iRand ;
  for ( int i = iLeft; i < iLength; i ++ )
  {
    iRand = (int)(drand48()*iLeft);       // pick random particle
    posArray[ i ] = posArray[ iRand ];    // and copy contents
    velArray[ i ] = velArray[ iRand ];
  }
}

ObjectT WorldModel::getMaxRangeUnknownPlayer( ObjectT obj, char* strMsg )
{
  list<ObjectT> l;
  ObjectT o;
  bool    isGoalie, bCont = true;
  int     i, loop;
//  bool    isTeammate = SoccerTypes::isTeammate( obj );
  ObjectT objMax = (getSide()==SIDE_LEFT) ? OBJECT_OPPONENT_11
                                : OBJECT_TEAMMATE_11 ;

  while( bCont == true )
```

```
{
 i = Parse::gotoFirstOccurenceOf( '(', &strMsg );
 if( i == -1 )
   bCont = false;                          // no more objects
 else
 {
   strMsg++;
   o = SoccerTypes::getObjectFromStr(&strMsg,&isGoalie,getTeamName());
   if( SoccerTypes::isPlayer( o ) )
     l.push_back( o );
 }
}
Log.log( 459, "list size %d", l.size() );
while( ! l.empty() )
{
 o = l.back();
 l.pop_back();
 if( SoccerTypes::isKnownPlayer( o ) )      // max range is one lower
   objMax = o;
 else if( SoccerTypes::isOpponent( o ) &&
          SoccerTypes::isTeammate( objMax ) )
   objMax = OBJECT_OPPONENT_11;
 else if( SoccerTypes::isTeammate( o ) &&
          SoccerTypes::isOpponent( objMax ) )
   objMax = OBJECT_TEAMMATE_11;

 if( objMax == getAgentObjectType() )
   loop = 2;
 else
   loop = 1;

 for( int j = 0; j < loop; j ++ )
 {
   i = SoccerTypes::getIndex( objMax );
   if( objMax == OBJECT_TEAMMATE_1 )
     objMax = OBJECT_OPPONENT_11;
   else if( objMax == OBJECT_OPPONENT_1 )
     objMax = OBJECT_TEAMMATE_11;
   else if( SoccerTypes::isTeammate( objMax ) )
     objMax = SoccerTypes::getTeammateObjectFromIndex( i - 1 );
   else if( SoccerTypes::isOpponent( objMax ) )
     objMax = SoccerTypes::getOpponentObjectFromIndex( i - 1 );
 }
 Log.log( 459, "processs %d new obj_max: %d", o, objMax );

}
```

```
  return objMax;
}
```

BasicCoach.h

```
#ifndef _BASICCOACH_
#define _BASICCOACH_

#include "ActHandler.h"

#ifdef WIN32
  DWORD WINAPI stdin_callback( LPVOID v );
#else
  void* stdin_callback( void * v );
#endif

/*! This class starts a simple coach, which actions are defined in the method
    mainLoop. It uses an ActHandler to send actions to the server and can
    receive information from the WorldModel. The declaration of the different
    methods are defined in BasicCoachTest.C and BasicCoach.C. */
class BasicCoach
{
protected:
  ActHandler     *ACT; /*!< ActHandler to which commands can be sent         */
  WorldModel     *WM;  /*!< WorldModel that contains information of world    */
  ServerSettings *SS;  /*!< All parameters used by the server                */

  HeteroPlayerSettings m_player_types[MAX_HETERO_PLAYERS];
  bool bContLoop;      /*!< bool to indicate whether to stop or continue     */

public:
  BasicCoach( ActHandler* a, WorldModel *wm, ServerSettings *ss,
              char* strTeamName, double dVersion, bool isTrainer );
  virtual ~BasicCoach( );

  virtual void mainLoopNormal   ( ); // virtual can be overwritten in subclass
  void         substitutePlayer( int iPlayer, int iPlayerType );

  // methods that deal with user input (from keyboard) to sent commands
  void         handleStdin             (                              );
  void         showStringCommands      ( ostream& out                 );
  bool         executeStringCommand    ( char *str                    );

};

#endif
```

BasicCoach.cpp

```
#include"BasicCoach.h"
#include"Parse.h"
#ifdef WIN32
  #include <windows.h>
#else
  #include <sys/poll.h>
#endif

extern Logger Log; /*!< This is a reference to the Logger to write loginfo to*/

/*!This is the constructor for the BasicCoach class and contains the
   arguments that are used to initialize a coach.
   \param act ActHandler to which the actions can be sent
   \param wm WorldModel which information is used to determine action
   \param ss ServerSettings that contain parameters used by the server
   \param strTeamName team name of this player
   \param dVersion version this basiccoach corresponds to
   \param isTrainer indicates whether the coach is a trainer (offline coach)
         or an online coach (used during the match). */
BasicCoach::BasicCoach( ActHandler* act, WorldModel *wm, ServerSettings *ss,
      char* strTeamName, double dVersion, bool isTrainer )

{
  char str[MAX_MSG];

  ACT       = act;
  WM        = wm;
  SS        = ss;
  bContLoop = true;
  WM->setTeamName( strTeamName );

  if( !isTrainer )
    sprintf( str, "(init %s (version %f))", strTeamName, dVersion );
  else
    sprintf( str, "(init (version %f))", dVersion );

   ACT->sendMessage( str );
}

BasicCoach::~BasicCoach( )
{
}

/*! This method is the main loop of the coach. All sequence of actions are
    located in this method. */
void BasicCoach::mainLoopNormal( )
```

```
{
#ifdef WIN32
  Sleep( 1000 );
#else
  poll( 0, 0, 1000 );
#endif

  bool bSubstituted   = false;
  ACT->sendMessageDirect( "(eye on)" );

#ifdef WIN32
  Sleep( 1000 );
#else
  poll( 0, 0, 1000 );
#endif

  while( WM->getPlayMode() != PM_TIME_OVER  && bContLoop )
  {
    Log.log( 1, "in loop %d %d %f",
             WM->getTimeLastSeeGlobalMessage().getTime(),
             bSubstituted,
             WM->isConfidenceGood( OBJECT_TEAMMATE_11 )) ;
    if(  WM->waitForNewInformation() == false )
    {
      printf( "Shutting down coach\n" );
      bContLoop = false;
    }
    else if( WM->getTimeLastSeeGlobalMessage().getTime() == 0 &&
             bSubstituted == false &&
             WM->isConfidenceGood( OBJECT_TEAMMATE_11 ))
    {
      // read (and write) all player_type information
      for( int i = 0 ; i < MAX_HETERO_PLAYERS; i ++ )
      {
        m_player_types[i] = WM->getInfoHeteroPlayer( i );
//      cout << i << ": " ;
//      m_player_types[i].show( cout );
      }

      // just substitute some players (define your own methods to
      // determine which player types should be substituted )
      substitutePlayer(  2, 1 );  // substitute player 2 to type 1
      substitutePlayer(  3, 1 );
      substitutePlayer(  4, 1 );
      substitutePlayer(  5, 2 );
      substitutePlayer(  6, 2 );
```

```
      substitutePlayer(  7, 2 );
      substitutePlayer(  8, 3 );
      substitutePlayer(  9, 3 );
      substitutePlayer( 10, 3 );
      substitutePlayer( 11, 4 );
      bSubstituted = true;
    }

    if( Log.isInLogLevel(  456 ) )
      WM->logObjectInformation( 456, OBJECT_ILLEGAL);
    if( SS->getSynchMode() == true )
      ACT->sendMessageDirect( "(done)" );
  }

  return;
}


/*! This method substitutes one player to the given player type and sends
    this command (using the ActHandler) to the soccer server. */
void BasicCoach::substitutePlayer( int iPlayer, int iPlayerType )
{
  SoccerCommand soc;
  soc.makeCommand( CMD_CHANGEPLAYER, (double)iPlayer, (double)iPlayerType );
  ACT->sendCommandDirect( soc );
  cerr << "coachmsg: changed player " << iPlayer << " to type " << iPlayerType
       << endl;
}


#ifdef WIN32
DWORD WINAPI stdin_callback( LPVOID v )
#else
void* stdin_callback( void * v )
#endif
{
  Log.log( 1, "Starting to listen for user input" );
  BasicCoach* bc = (BasicCoach*)v;
  bc->handleStdin();
  return NULL;
}

/*!This method listens for input from the keyboard and when it receives this
   input it converts this input to the associated action. See
   showStringCommands for the possible options. This method is used together
   with the SenseHandler class that sends an alarm to indicate that a new
```

```
   command can be sent. This conflicts with the method in this method that
   listens for the user input (fgets) on Linux systems (on Solaris this isn't a
   problem). The only known method is to use the flag SA_RESTART with this
   alarm function, but that does not seem to work under Linux. If each time
   the alarm is sent, this gets function unblocks, it will cause major
   performance problems. This function should not be called when a whole match
   is played! */
void BasicCoach::handleStdin( )
{
  char buf[MAX_MSG];

  while( bContLoop )
  {
#ifdef WIN32
    cin.getline( buf, MAX_MSG );
#else
    fgets( buf, MAX_MSG, stdin ); // does unblock with signal !!!!!
#endif
    printf( "after fgets: %s\n", buf );
    executeStringCommand( buf );
  }
}

/*!This method prints the possible commands that can be entered by the user.
   The whole name can be entered to perform the corresponding command, but
   normally only the first character is sufficient. This is indicated by
   putting brackets around the part of the command that is not needed.
   \param out output stream to which the possible commands are printed */
void BasicCoach::showStringCommands( ostream& out )
{
  out << "Basic commands:"                          << endl <<
         " m(ove) player_nr x y"                    << endl <<
         " q(uit)"                                  << endl;
}

/*!This method executes the command that is entered by the user. For the
   possible command look at the method showStringCommands.
   \param str string that is entered by the user
   \return true when command could be executed, false otherwise */
bool BasicCoach::executeStringCommand( char *str)
{
  switch( str[0] )
  {
    case 'm':                           // move
      sprintf( str, "(move %d %f %f)", Parse::parseFirstInt( &str ),
                                Parse::parseFirstDouble( &str ),
```

```
                                Parse::parseFirstDouble( &str ) );
      break;
    case 'q':                   // quit
      bContLoop = false;
      return true;
    default:                    // default: send entered string
      ;
  }
  printf( "send: %s\n", str );
  ACT->sendMessage( str );
  return true;
}
```

Logger.h

```
#ifndef _LOGGER_
#define _LOGGER_

#include <iostream>   // needed for ostream (logging to output stream)
#include <fstream>    // needed for fstream (logging to file)
#include <string>     // needed for string
#include <iomanip>    // needed for setw
#include <set>        // needed for set

#ifdef WIN32
  #include <windows.h>  // needed for DWORD and GetTickCount() function
  #include <time.h>     // needed for time_t
#else
  #include <sys/time.h> // needed for timeval
#endif

using namespace std;

#define MAX_LOG_LINE 3072 /*!< maximum size of a log message */
#define MAX_HEADER   128  /*!< maximum size of the header    */

/*****************************************************************************
/
/********************** CLASS TIMING
*****************************************/
/*****************************************************************************
/

/*!This class holds a timer. This timer can be set (restartTime) and text can
   be printed with the elapsed time since the timer was restarted.. */
class Timing
{
#ifdef WIN32
  DWORD  time1;                 /*! < the number of milliseconds that have
                              elapsed since the Windows was started */
#else
  struct timeval time1;         /*!< Time the timer has last been restarted.*/
#endif

public:
  // methods to restart the timer, get the elapsed time and print messages
#ifdef WIN32
  static double getTimeDifference    ( DWORD tv1, DWORD tv2                );
#else
  static double getTimeDifference    ( struct   timeval t1,
                               struct   timeval t2                 );
```

```
#endif
  void        printTimeDiffWithText( ostream& os,
                                     char     *str,
                                     int      iFactor = 1000          );
  double       getElapsedTime       ( int      iFactor = 1             );
  void        restartTime          (                                  );
} ;


/*************************************************************************
/
/*************************** LOGGER
****************************************/
/*************************************************************************
/

/*! This class makes it possible to log information on different abstraction
    levels. All messages are passed to the log method 'log' with a level
    indication. When it has been specified that this level should be logged
    using either the 'addLogLevel' or 'addLogRange' method
    the message is logged, otherwise it is ignored. This makes it
    possible to print only  the information you are interested in.
    There is one global Log class which is used by all classes that use the
    Logger. This instantation of the Logger is located in the file Logger.C and
    is called 'Log'. All classes that want use this Logger should make a
    reference to it using the line 'extern Logger Log;' and can then use
    this Logger with the Log.log( ... ) methods. Furthermore the Logger also
    contains a timer with makes it possible to print the time since the timer
    has been restarted. */
class Logger
{
  Timing   m_timing;             /*!< timer to print timing information    */
  char     m_buf[MAX_LOG_LINE];    /*!< buffer needed by different methods   */
  set<int> m_setLogLevels;        /*!< set that contains all log levels     */

  pthread_mutex_t mutex_stream;
  char     m_strHeader[MAX_HEADER];/*!< header string printed before msg     */
  ostream* m_os;                  /*!< output stream to print messages to   */
  string   m_strSignal;           /*!< temporary string for other messages  */

public:
  Logger( ostream& os=cout, int iMinLogLevel=0, int iMaxLogLevel = 0);

  // different methods associated with logging messages
  bool     log            ( int        iLevel, string str            );
  bool     log            ( int        i,      char   *str, ...      );
```

```
  bool    logWithTime     ( int       iLevel, char   *str, ...       );
  bool    logFromSignal   ( int       iLevel, char   *str, ...       );
  bool    logSignal       (                                          );

  void    restartTimer    (                                          );
  Timing  getTiming       (                                          );
  bool    isInLogLevel    ( int       iLevel                         );

  bool    addLogLevel     ( int       iLevel                         );
  bool    addLogRange     ( int       iMin,   int iMax               );

  char*   getHeader       (                                          );
  bool    setHeader       ( char      *str                           );
  bool    setHeader       ( int       i                              );
  bool    setHeader       ( int       i1,     int   i2               );

  bool    setOutputStream ( ostream&   os                            );
  ostream& getOutputStream (                                         );
  void    showLogLevels   ( ostream&   os                            );
};


#endif
```

Logger.cpp

```
#include "Logger.h"
#include <stdio.h>    // needed for vsprintf
#include <string>     // needed for string
#ifdef Solaris
 #include <varargs.h> // needed for va_list and va_start under Solaris
#else
 #include <stdarg.h>
#endif
#include <pthread.h>

Logger Log;         /*!<Logger instantation that can be used by all classes */
Logger LogDraw;     /*!<Drawing logger instantation for all classes        */

/*****************************************************************************
/
/***************************** LOGGER
*****************************************/
/*****************************************************************************
/

/*!This is the constructor for the Logger. The output stream, the minimal and
   maximal log level can all be specified. The timer in this class is also
   restarted.
   \param o outputstream (file or cout) to which information is printed
          (default cout)
   \param iMin minimal log level (default 0)
   \param iMax maximal log level (default 0) */
Logger::Logger( ostream& o, int iMin, int iMax )
{
  pthread_mutex_init( &mutex_stream, NULL );
  strcpy( m_strHeader, "" );
  m_timing.restartTime();
  addLogRange( iMin, iMax );
  m_os = &o;
}

/*!This method can be used to log information. Only when the specified
   level of the message is part of the set of logged levels the
   information is logged. This method receives a a normal string that
           is logged.
   \param iLevel level corresponding to this message
   \param str string that is logged when iLevel is a logging lvel.
   \return bool indicating whether the message was logged or not. */
bool Logger::log( int iLevel, string str)
{
  if( isInLogLevel( iLevel ) )
```

```
  {
    *m_os << m_strHeader << str;
    return true;
  }

  return false;
}

/*!This method can be used to log information. Only when the specified
   level of the message is part of the set of logged values the
   information is logged. This method receives a character string that may
   contain format specifiers that are also available to 'printf' (like %d, %f,
   etc.). The remaining arguments are the variables that have to be filled in
   at the location of the specifiers.
   \param iLevel level corresponding to this message
   \param str character string with (possible) format specifiers
   \param ... variables that define the values of the specifiers.
   \return bool indicating whether the message was logged or not. */
bool Logger::log( int iLevel, char *str, ... )
{
  if( isInLogLevel( iLevel ) )
  {
    logSignal(); // test whether there are no old strings left to log
    va_list ap;
#ifdef Solaris
    va_start( ap );
#else
    va_start( ap, str );
#endif
    if( vsnprintf( m_buf, MAX_LOG_LINE-1, str, ap ) == -1 )
      cerr << "Logger::log, buffer is too small!\n" ;
    va_end(ap);
    *m_os << m_strHeader << m_buf << "\n";
    return true;
  }

  return false;
}

/*!This method can be used to log information. Only when the specified
   level of the message is an element in the set of logged levels the
   information is logged. This method receives a character string that may
   contain format specifiers that are also available to 'printf' (like %d, %f,
   etc.). The remaining arguments are the variables that have to be filled in
   at the location of the specifiers. Before the logged message the elapsed
   time since the timer has been restarted is printed.
```

```
   \param iLevel level corresponding to this message
   \param str character string with (possible) format specifiers
   \param ... variables that define the values of the specifiers.
   \return bool indicating whether the message was logged or not. */
bool Logger::logWithTime( int iLevel, char *str, ... )
{
  if( isInLogLevel( iLevel ) )
  {
    logSignal(); // test whether there are no old strings left to log
    va_list ap;
#ifdef Solaris
    va_start( ap );
#else
    va_start( ap, str );
#endif
    if( vsnprintf( m_buf, MAX_LOG_LINE-1, str, ap ) == -1 )
      cerr << "Logger::log, buffer is too small!" << "\n";
    va_end(ap);

    string s = m_strHeader;
    s.append( m_buf );
    s.copy( m_buf, string::npos );
    m_buf[s.length()] = '\0';
    m_timing.printTimeDiffWithText( *m_os, m_buf );

    return true;
  }

  return false;
}

/*!This method can be used to log information. The main difference
 * with the standard log method is that in this case the string is not
 * printed to the output stream immediately but only at the next call
 * of log (or logWithTime). This is a work-around for specific
 * compiler cases in which a deadlock occurs when something is logged
 * at the same instance by two different threads.

 \param iLevel level corresponding to this message
 \param str character string with (possible) format specifiers
 \param ... variables that define the values of the specifiers.
 \return bool indicating whether the message was logged or not. */
bool Logger::logFromSignal( int iLevel, char   *str, ...        )
{
  char buf[MAX_LOG_LINE];
  if( isInLogLevel( iLevel ) )
```

```
  {
    va_list ap;
#ifdef Solaris
    va_start( ap );
#else
    va_start( ap, str );
#endif
    if( vsnprintf( buf, MAX_LOG_LINE-1, str, ap ) == -1 )
      cerr << "Logger::log, buffer is too small!" << "\n";
    va_end(ap);

    char str[16];
    sprintf( str, "%2.2f: ", m_timing.getElapsedTime()*1000 );
    m_strSignal.append( str );
    m_strSignal.append( m_strHeader );
    m_strSignal.append( buf );
    m_strSignal.append(  "\n\0" );

    return true;
  }

  return false;

}

/*! This method actually writes the string that was created with
 *  logFromSignal to the specified output stream.
 \return bool indicating whether the message was logged or not. */
bool Logger::logSignal( )
{
  if( ! m_strSignal.empty() )
  {
    *m_os << "\n" << m_strSignal ;
    m_strSignal = "";
    return true;
  }
  return false;
}

/*!This method restarts the timer associated with this Logger. */
void Logger::restartTimer()
{
  return m_timing.restartTime();
}

/*! Return the instance of the timing class that denotes the time the counter
```

```
    is running. */
Timing Logger::getTiming( )
{
  return m_timing;
}


/*!This method returns whether the supplied log level is recorded, thus
   within the interval [min_log_level..max_log_level] or equal to the extra
   log level.
   \param iLevel log level that should be checked
   \return bool indicating whether the supplied log level is logged. */
bool Logger::isInLogLevel( int iLevel )
{
  return m_setLogLevels.find( iLevel ) != m_setLogLevels.end() ;
}

/*! This method inserts the log level 'iLevel' to the set of logged levels.
    Information from this log level will be printed.
                    \param iLevel level that will be added to the set
    \return bool indicating whether the update was successfull. */
bool  Logger::addLogLevel( int iLevel )
{
  m_setLogLevels.insert( iLevel );
  return true;
}

/*! This method inserts all the log levels in the interval [iMin..iMax] to
    the set of logged levels.
    \param iMin minimum log level that is added
    \param iMax maximum log level that is added
    \return bool indicating whether the update was successfull. */
bool  Logger::addLogRange( int iMin, int iMax )
{
  bool bReturn = true;
  for( int i = iMin ; i <= iMax;  i++ )
    bReturn &= addLogLevel( i );
  return bReturn;
}

/*! This method returns the current header that is written before the actual
    text that has to be logged.
    \return current header */
char* Logger::getHeader( )
{
  return m_strHeader;
```

```
}

/*! This method sets the header that is written before the actual logging text.
    \param str that represents the character string
    \return bool indicating whether the update was succesfull */
bool Logger::setHeader( char *str )
{
  strcpy( m_strHeader, str );
  return true;
}

/*! This method sets a predefined header that is written before the actual
    logging text. The header is represented by two integers which are
    written between parentheses, i.e. (9, 2401).
    \param i1 first integer
    \param i2 second integer
    \return bool indicating whether the update was succesfull */
bool Logger::setHeader( int i1, int i2 )
{
  sprintf( m_strHeader, "(%d, %d) ", i1, i2 );
  return true;
}

/*! This method sets a predefined header that is written before the actual
    logging text. The header is represented by one integer which is followed by
    a semicolon (":") .
    \param integer
    \return bool indicating whether the update was succesfull */
bool Logger::setHeader( int i )
{
  sprintf( m_strHeader, " %d: ", i );
  return true;
}


/*!This method sets the output stream to which the log information is written.
   This outputstream can be standard output (cout) or a reference to a
   file.
   \param o outputstream to which log information is printed.
   \return bool indicating whether update was succesfull. */
bool Logger::setOutputStream( ostream& o )
{
  m_os = &o;
  return true;
}
```

```
/*!This method returns the output stream to which the log information is
   written. This outputstream can be standard output (cout) or a reference to a
   file.
   \return o outputstream to which log information is printed. */
ostream& Logger::getOutputStream(  )
{
 return *m_os;
}

/*! This method outputs all the log levels that are logged to the output stream
    os.
    \param os output stream to which log levels are printed. */
void Logger::showLogLevels( ostream &os )
{
 set<int>::iterator itr;
 for (itr = m_setLogLevels.begin() ; itr != m_setLogLevels.end() ; itr++)
   os << *itr << " " ;
 os << "\n";
}
/*************************************************************************
/
/********************** CLASS TIMING
*****************************************/
/*************************************************************************
/

/*! This method returns the difference between two timevals in seconds.
    \param tv1 first timeval
    \param tv2 second timeval
    \return double representing the difference between t1 and t2 in seconds */
#ifdef WIN32
double Timing::getTimeDifference( DWORD tv1, DWORD tv2 )
{
 return  ((double)(tv1 - tv2) / 1000.0) ;
}
#else
double Timing::getTimeDifference( struct timeval tv1, struct timeval tv2 )
{

 return  ((double)tv1.tv_sec + (double)tv1.tv_usec/1000000 ) -
        ((double)tv2.tv_sec + (double)tv2.tv_usec/1000000 ) ;
}
#endif

/*! This method prints the time in seconds that elapsed since
    the timer was restarted. It is possible to multiply this time with a
```

```
    factor, such that it can be used for different quantity values. In the
    default case this value is 1000 with the effect that the resulting value
    resembles miliseconds.
    \param os output stream to which output should be written.
    \param str that should be printed
    \param iFactor with which the elapsed time is multiplied (default 1000) */
void Timing::printTimeDiffWithText( ostream &os, char *str, int iFactor )
{
  // set the with to 6 and fill remaining places with '0'.
  os <<setw(6)<< setfill('0')<< getElapsedTime()*iFactor << ":" << str << "\n";
}

/*! This method returns the time (in seconds) since the last time the timer
    was restarted.
    \return double seconds that have passed since last restart of timer */
double Timing::getElapsedTime( int iFactor )
{
#ifdef WIN32
  DWORD time2 = GetTickCount();
#else
  struct timeval time2;
  gettimeofday( &time2, NULL );
#endif
  return getTimeDifference( time2, time1 )*iFactor;
}

/*! This method restarts the timer by setting it to the current time */
void Timing::restartTime( )
{
#ifdef WIN32
  time1 = GetTickCount();
#else
  gettimeofday( &time1, NULL );
#endif
}


/*****************************************************************************
/
/****************************** TESTING PURPOSES
*******************************/
/*****************************************************************************
/

/*
#ifdef WIN32
```

```
  #include <windows.h>
#else
  #include<sys/poll.h>
#endif

int main( void )
{
  ofstream fout( "temp.txt" );
  Logger log( fout, 0, 2 );
  log.log( 0, "hello" );
  log.setHeader( "jelle" );
#ifdef WIN32
  Sleep(1000);
#else
  poll(0,0,1000);
#endif
  log.log( 2, "hello" );
  log.log( 3, "hello" );
  int j = 2;
  double i = 2.324234;
  printf( "hoi: "); fflush(stdout);
  log.logWithTime( 1, "|%f %d|", i, j);
}

*/
```

mainCoach.cpp

```
#include"ActHandler.h"
#include"SenseHandler.h"
#include"BasicCoach.h"
#include"Parse.h"

#include<string.h>   // strcpy
#ifdef WIN32
  #include <windows.h>  // needed for CreateThread
#else
  #include <pthread.h>  // needed for pthread_create
#endif
#include<stdlib.h>   // exit

extern Logger Log; /*!< This is a reference to the Logger to write log info to*/
void printOptions( );

/*! This is the main function and creates and links all the different
    classes.  First it reads in all the parameters from the command
    prompt (<program name> -help)and uses these values to create the
    classes. After all the classes are linked, the mainLoop in the
    Player class is called. */
int main( int argc, char * argv[] )
{
#ifdef WIN32
  HANDLE        listen, sense;
#else
  pthread_t     listen, sense;
#endif
  ServerSettings ss;
  PlayerSettings cs;

  // define variables for command options and initialize with default values
  char    strTeamName[MAX_TEAM_NAME_LENGTH] = "UvA_Trilearn";
  int     iPort                       = 6002;
  int     iMinLogLevel                  = 0 ;
  int     iMaxLogLevel                   = 0;
  char    strHost[128]                 = "localhost";
  double  dVersion                      = 9.3;
  int     iMode                       = 0;
  char    strFormations[128]              = "formations.conf";
  int     iNr                         = 0;
  int     iReconnect                    = -1;
  bool    bInfo                       = false;
  bool    bSuppliedLogFile              = false;
  ofstream os;
```

```
// read in all the command options and change the associated variables
// assume every two values supplied at prompt, form a duo
char * str;
for( int i = 1 ; i < argc ; i = i + 2  )
{
 // help is only option that does not have to have an argument
 if( i + 1 >= argc && strncmp( argv[i], "-help", 3 ) != 0 )
 {
  cout << "Need argument for option: " << argv[i] << endl;
  exit( 0 );
 }
 // read a command option
 if( argv[i][0] == '-' && strlen( argv[i] ) > 1)
 {
  switch( argv[i][1] )
  {
   case 'h':                              // host server or help
    if( strlen( argv [i]) > 2 && argv[i][2] == 'e' )
    {
     printOptions( );
     exit(0);
    }
    else
     strcpy( strHost, argv[i+1] );
    break;
   case 'f':                              // formations file
    strcpy( strFormations, argv[i+1] );
    break;
   case 'c':                              // clientconf file
    if( cs.readValues( argv[i+1], ":" ) == false )
     cerr << "Error in reading client file: " << argv[i+1] << endl;
    break;
   case 'i':                              // info 1 0
    str   = &argv[i+1][0];
    bInfo = (Parse::parseFirstInt( &str ) == 1 ) ? true : false ;
    break;
   case 'l':                              // loglevel int[..int]
    str = &argv[i+1][0];
    iMinLogLevel = Parse::parseFirstInt( &str );
    while( iMinLogLevel != 0 )
    {
     if( *str == '.' ) // '.' indicates range of levels
     {
      iMaxLogLevel = Parse::parseFirstInt( &str );
      if( iMaxLogLevel == 0 ) iMaxLogLevel = iMinLogLevel;
      Log.addLogRange( iMinLogLevel, iMaxLogLevel );
```

```
        }
        else
          Log.addLogLevel( iMinLogLevel );
        iMinLogLevel = Parse::parseFirstInt( &str );
       }
       break;
     case 'm':                          // mode int
       str = &argv[i+1][0];
       iMode = Parse::parseFirstInt( &str );
       break;
     case 'o':                          // output file log info
       os.open( argv[i+1] );
       bSuppliedLogFile = true;
       break;
     case 'p':                          // port
       str = &argv[i+1][0];
       iPort = Parse::parseFirstInt( &str );
       break;
     case 's':                          // serverconf file
       if( ss.readValues( argv[i+1], ":" ) == false )
         cerr << "Error in reading server file: " << argv[i+1] << endl;
       break;
     case 't':                          // teamname name
       strcpy( strTeamName, argv[i+1] );
       break;
     case 'v':                          // version version
       str = &argv[i+1][0];
       dVersion = Parse::parseFirstDouble( &str );
       break;
     default:
       cerr << "(main) Unknown command option: " << argv[i] << endl;
    }
   }
}


if( bInfo == true )
cout << "team        : "  << strTeamName    << endl <<
        "port        : "  << iPort          << endl <<
        "host        : "  << strHost        << endl <<
        "version     : "  << dVersion       << endl <<
        "min loglevel : "  << iMinLogLevel   << endl <<
        "max loglevel : "  << iMaxLogLevel   << endl <<
        "mode        : "  << iMode          << endl <<
        "playernr    : "  << iNr            << endl <<
        "reconnect   : "  << iReconnect     << endl ;
```

```
  if( bSuppliedLogFile == true )
    Log.setOutputStream( os );                  // initialize logger
  else
    Log.setOutputStream( cout );
  Log.restartTimer( );
  Formations fs( strFormations, (FormationT)cs.getInitialFormation(), iNr );
                                    // read formations file
  WorldModel wm( &ss, &cs, &fs );               // create worldmodel
  Connection c( strHost, iPort, MAX_MSG );     // make connection with server
  ActHandler a( &c, &wm, &ss );                 // link actHandler and WM
  SenseHandler s( &c, &wm, &ss, &cs );          // link senseHandler with wm
  bool isTrainer = (iPort == ss.getCoachPort()) ? true : false;
  BasicCoach bp( &a, &wm, &ss, strTeamName, dVersion, isTrainer );
                                    // link acthandler and WM

#ifdef WIN32
  DWORD id1;
  sense = CreateThread(NULL, 0, &sense_callback, &s, 0, &id1);
  if (sense == NULL)
  {
     cerr << "create thread error" << endl;
     return false;
  }
#else
  pthread_create( &sense, NULL, sense_callback  , &s); // start listening
#endif

  if( iMode > 0 && iMode < 9 )       // only listen to sdtdin when not playing
#ifdef WIN32
  {
    DWORD id2;
    listen = CreateThread(NULL, 0, &stdin_callback, &bp, 0, &id2);
    if ( listen == NULL)
    {
       cerr << "create thread error" << endl;
       return false;
    }
  }
#else
    pthread_create( &listen, NULL, stdin_callback, &bp);
#endif

  if( iMode == 0 )
    bp.mainLoopNormal();
```

```
  c.disconnect();
  os.close();
}


/*! This function prints the command prompt options that can be supplied to the
    program. */
void printOptions( )
{
  cout << "Command options:"                                  << endl <<
   " c(lientconf) file     - use file as client conf file"     << endl <<
   " f(ormations) file     - file with formation info"         << endl <<
   " he(lp)                - print this information"           << endl <<
   " h(ost) hostname       - host to connect with"             << endl <<
   " i(nfo) 0/1            - print variables used to start"    << endl <<
   " l(oglevel) int[..int] - level of debug info"              << endl <<
   " m(ode) int            - which mode to start up with"      << endl <<
   " o(utput) file         - write log info to (screen is default)"  << endl <<
   " p(ort)                - port number to connect with"      << endl <<
   " s(erverconf) file     - use file as server conf file"     << endl <<
   " t(eamname) name       - name of your team"                << endl;
}
```

main.cpp

```
#include "SenseHandler.h"
#include "Player.h"
#include "Parse.h"
#include <string.h>   // needed for strcpy
#ifdef WIN32
  #include <windows.h>  // needed for CreateThread
#else
  #include <pthread.h>  // needed for pthread_create
#endif
#include <stdlib.h>   // needed for exit

extern Logger Log;     /*!< This is a reference to the normal Logger class   */
extern Logger LogDraw; /*!< This is a reference to the drawing Logger class  */

void printOptions( );

/*! This is the main function and creates and links all the different classes.
    First it reads in all the parameters from the command prompt
    (<program name> -help) and uses these values to create the classes. After
    all the classes are linked, the mainLoop in the Player class is called. */
int main( int argc, char * argv[] )
{

#ifdef WIN32
  HANDLE        listen, sense;
#else
  pthread_t     listen, sense;
#endif

  ServerSettings ss;
  PlayerSettings cs;

  // define variables for command options and initialize with default values
  char    strTeamName[MAX_TEAM_NAME_LENGTH] = "UvA_Trilearn";
  int     iPort                       = ss.getPort();
  int     iMinLogLevel                      ;
  int     iMaxLogLevel                      ;
  char    strHost[128]                    = "localhost";
  double  dVersion                        = 9.3;
  int     iMode                       = 0;
  char    strFormations[128]                = "formations.conf";
  int     iNr                         = 2;
  int     iReconnect                    = -1;
  bool    bInfo                       = false;
  bool    bSuppliedLogFile                  = false;
  bool    bSuppliedLogDrawFile              = false;
```

```
ofstream os;
ofstream osDraw;

// read in all the command options and change the associated variables
// assume every two values supplied at prompt, form a duo
char * str;
for( int i = 1 ; i < argc ; i = i + 2  )
{
 // help is only option that does not have to have an argument
 if( i + 1 >= argc && strncmp( argv[i], "-help", 3 ) != 0 )
 {
  cout << "Need argument for option: " << argv[i] << endl;
  exit( 0 );
 }
 // read a command option
 if( argv[i][0] == '-' && strlen( argv[i] ) > 1)
 {
  switch( argv[i][1] )
  {
   case '?':                          // print help
    printOptions( );
    exit(0);
    break;
   case 'a':                          // output file drawlog info
    osDraw.open( argv[i+1] );
    bSuppliedLogDrawFile = true;
    break;
   case 'c':                          // clientconf file
    if( cs.readValues( argv[i+1], ":" ) == false )
     cerr << "Error in reading client file: " << argv[i+1] << endl;
    break;
   case 'd':                          // drawloglevel int[..int]
    str = &argv[i+1][0];
    iMinLogLevel = Parse::parseFirstInt( &str );
    while( iMinLogLevel != 0 )
    {
     if( *str == '.' || *str == '-') // '.' or '-' indicates range
     {
      *str += 1 ;
      iMaxLogLevel = Parse::parseFirstInt( &str );
      if( iMaxLogLevel == 0 ) iMaxLogLevel = iMinLogLevel;
      LogDraw.addLogRange( iMinLogLevel, iMaxLogLevel );
     }
     else
      LogDraw.addLogLevel( iMinLogLevel );
     iMinLogLevel = Parse::parseFirstInt( &str );
```

```
  }
  break;
case 'f':                              // formations file
  strcpy( strFormations, argv[i+1] );
  break;
case 'h':                               // host server or help
  if( strlen( argv [i]) > 2 && argv[i][2] == 'e' )
  {
    printOptions( );
    exit(0);
  }
  else
    strcpy( strHost, argv[i+1] );
  break;
case 'i':                              // info 1 0
  str   = &argv[i+1][0];
  bInfo = (Parse::parseFirstInt( &str ) == 1 ) ? true : false ;
  break;
case 'l':                              // loglevel int[..int]
  str = &argv[i+1][0];
  iMinLogLevel = Parse::parseFirstInt( &str );
  while( iMinLogLevel != 0 )
  {
    if( *str == '.' || *str == '-') // '.' or '-' indicates range
    {
      *str += 1 ;
      iMaxLogLevel = Parse::parseFirstInt( &str );
      if( iMaxLogLevel == 0 ) iMaxLogLevel = iMinLogLevel;
      Log.addLogRange( iMinLogLevel, iMaxLogLevel );
    }
    else
      Log.addLogLevel( iMinLogLevel );
    iMinLogLevel = Parse::parseFirstInt( &str );
  }
  break;
case 'm':                               // mode int
  str = &argv[i+1][0];
  iMode = Parse::parseFirstInt( &str );
  break;
case 'n':                              // number in formation int
  str = &argv[i+1][0];
  iNr = Parse::parseFirstInt( &str );
  break;
case 'o':                              // output file log info
  os.open( argv[i+1] );
  bSuppliedLogFile = true;
```

```
        break;
      case 'p':                          // port
        str = &argv[i+1][0];
        iPort = Parse::parseFirstInt( &str );
        break;
      case 'r':                          // reconnect 1 0
        str = &argv[i+1][0];
        iReconnect = Parse::parseFirstInt( &str );
        break;
      case 's':                          // serverconf file
        if( ss.readValues( argv[i+1], ":" ) == false )
          cerr << "Error in reading server file: " << argv[i+1] << endl;
        break;
      case 't':                          // teamname name
        strcpy( strTeamName, argv[i+1] );
        break;
      case 'v':                          // version version
        str = &argv[i+1][0];
        dVersion = Parse::parseFirstDouble( &str );
        break;
      default:
        cerr << "(main) Unknown command option: " << argv[i] << endl;
     }
   }
  }
  if( bInfo == true )
  {
   cout << "team         : "  << strTeamName    << endl <<
           "port         : "  << iPort          << endl <<
           "host         : "  << strHost        << endl <<
           "version      : "  << dVersion       << endl <<
           "mode         : "  << iMode          << endl <<
           "playernr     : "  << iNr            << endl <<
           "reconnect    : "  << iReconnect     << endl ;
   Log.showLogLevels( cout );
   LogDraw.showLogLevels( cout );
  }
  if( bSuppliedLogFile == true )
   Log.setOutputStream( os );                // initialize logger
  else
   Log.setOutputStream( cout );
  if( bSuppliedLogDrawFile == true )
   LogDraw.setOutputStream( osDraw );        // initialize drawing logger
  else
   LogDraw.setOutputStream( cout );
```

```
  Log.restartTimer( );

  Formations fs( strFormations, (FormationT)cs.getInitialFormation(), iNr-1 );
                                  // read formations file
  WorldModel wm( &ss, &cs, &fs );             // create worldmodel
  Connection c( strHost, iPort, MAX_MSG );     // make connection with server
  ActHandler a( &c, &wm, &ss );               // link actHandler and worldmodel
  SenseHandler s( &c, &wm, &ss, &cs );         // link senseHandler with wm
  Player bp( &a, &wm, &ss, &cs, &fs, strTeamName, dVersion, iReconnect );
                                  // create player
#ifdef WIN32
  DWORD id1;
  sense = CreateThread(NULL, 0, &sense_callback, &s, 0, &id1);
  if (sense == NULL)
  {
     cerr << "creat thread error" << endl;
     return false;
  }
#else
  pthread_create( &sense, NULL, sense_callback  , &s); // start listening
#endif

  if( iMode > 0 && iMode < 9 ) // only listen to stdin when not playing
#ifdef WIN32
  {
    DWORD id2;
    listen = CreateThread(NULL, 0, &stdin_callback, &bp, 0, &id2);
    if ( listen == NULL)
    {
       cerr << "create thread error" << endl;
       return false;
    }
  }
#else
    pthread_create( &listen, NULL, stdin_callback, &bp);
#endif

  if( iMode == 0 )
    bp.mainLoop();

  c.disconnect();
  os.close();
}

/*! This function prints the command prompt options that can be supplied to the
    program. */
```

```
void printOptions( )
{
  cout << "Command options:"                                    << endl <<
    " a file               - write drawing log info to "        << endl <<


    " c(lientconf) file     - use file as client conf file"     << endl <<
    " d(rawloglevel) int[..int] - level(s) of drawing debug info"  << endl <<
    " f(ormations) file     - file with formation info"         << endl <<
    " he(lp)               - print this information"           << endl <<
    " h(ost) hostname       - host to connect with"            << endl <<
    " i(nfo) 0/1            - print variables used to start"   << endl <<
    " l(oglevel) int[..int] - level of debug info"             << endl <<
    " m(ode) int            - which mode to start up with"     << endl <<
    " n(umber) int          - player number in formation"      << endl <<
    " o(utput) file         - write log info to (screen is default)"  << endl <<
    " p(ort)               - port number to connect with"      << endl <<
    " r(econnect) int       - reconnect as player nr"          << endl <<
    " s(erverconf) file     - use file as server conf file"    << endl <<
    " t(eamname) name       - name of your team"               << endl;
}
```

formations.conf

```
# We have the following player types:
        # ---------------------------------
        #
        # 0 = PT_UNKNOWN
        # 1 = PT_GOALKEEPER
        # 2 = PT_DEFENDER_CENTRAL
        # 3 = PT_DEFENDER_SWEEPER
        # 4 = PT_DEFENDER_WING
        # 5 = PT_MIDFIELDER_CENTER
        # 6 = PT_MIDFIELDER_WING
        # 7 = PT_ATTACKER_WING
        # 8 = PT_ATTACKER_CENTRAL
        #
        # WE HAVE THE FOLLOWING FORMATIONS:
        # ---------------------------------
        #
        # 0 = FT_UNKNOWN = 000
        # 1 = FT_INITIAL = 111
        # 2 = FT_433_OFFENSIVE
        # 3 = FT_334_OFFENSIVE
        # 4 = FT_DEFENSIVE = 442
        # 5 = FT_OPEN_DEFENSIVE = 442
        # 6 = FT_343_ATTACKING = 244
        #
        # Layout of information per formation:
        # ------------------------------------
        # Formation number
        # Formation type number
        # X-position of eleven players
        # Y-position of eleven players
        # Player types for eleven players
        # X-attraction factors to ball for each player type
        # Y-attraction factors to ball for each player type
        # Booleans denoting for each player type whether to always remain behind ball
        # Minimal X-coordinate for each player type
        # Maximal X-coordinate for each player type
        #
        # Formation 0 = FT_unknown = 000
        0
          0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0       # X_pos
          0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0       # Y_pos
           0     0     0     0     0     0     0     0     0     0     0         # P_type
          0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0                   # X_attr
          0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0                   # Y_attr
           0     0     0     0     0     0     0     0     0                     # Behind_ball
          0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0                   # X_min
```

```
 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0                    # X_max


# Formation 1 = FT_initial = 111
1
-50.0 -16.0 -17.0 -17.0 -16.0  -8.0  -5.0  -5.0  -2.0  -1.0  -1.0  # X_pos
  0.0  16.0   5.0  -5.0 -16.0   0.0  10.0 -10.0   0.0  22.0 -22.0   # Y_pos
   1    4    2    2    4    5    6    6    8    7    7              # P_type
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0              # X_attr
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0              # Y_attr
   0    0    0    0    0    0    0    0    0                       # Behind_ball
 -49.0  -45.0   -45.0   -45.0   -45.0   -45.0   -45.0 -45.0 -45.0   # X_min
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0              # X_max


# Formation 2 = FT_433_offensive
2
-50.0 -16.5 -21.0 -15.0 -16.5   0.0   0.0  -3.0  15.0  18.0  18.0  # X_pos
  0.0  10.0   0.0  0.0  -10.0 -11.0  11.0   0.5  -0.5  19.0 -19.0   # Y_pos
   1    4    3    2    4    6    6    5    8    7    7              # P_type
  0.0   0.1   0.7  0.65   0.7  0.65   0.7   0.5  0.6              # X_attr
  0.0   0.1   0.2   0.4 0.25  0.3  0.25   0.3  0.25              # Y_attr
   0    1    1    1    0    0    0    0    0                      # Behind_ball
  0.0 -50.5 -42.0 -47.0 -45.0 -36.0 -36.0  -2.0  -2.0            # X_min
  0.0 -30.0   0.0   2.0   2.0  42.0  42.0   44.0 44.0             # X_max


# Formation 3 = FT_334_offensive
3
-50.0 -16.0 -21.0  -5.0 -16.0   3.0   3.0  15.0  15.0  18.0  18.0  # X_pos
  0.0  10.0   0.0   0.5 -10.0 -11.0  11.0  -5.0   5.0  19.0 -19.0   # Y_pos
   1    4    2    5    4    6    6    8    8    7    7              # P_type
  0.0   0.1   0.6   0.6   0.6   0.6   0.6   0.5   0.6              # X_attr
  0.0   0.1   0.25  0.25  0.25  0.25  0.3  0.2  0.2               # Y_attr
   0    1    1    1    0    0    0    0    0                      # Behind_ball
  0.0 -50.5 -45.0 -45.0 -45.0 -30.0 -30.0 -45.0   -2.0           # X_min
  0.0 -30.0   0.0   0.0  2.0   20.0  42.0  44.0   45.0            # X_max


# Formation 4 = FT_defensive = 442
4
-45.0 -12.0 -14.0 -14.0 -12.0   1.0  -5.0  -5.0   1.0  18.0  18.0  # X_pos
  0.0  21.0   5.0  -5.0 -21.0  24.0   8.0  -8.0 -24.0   7.0  -7.0    # Y_pos
   1    4    2    2    4    6    5    5    6    8    8              # P_type
  0.0   0.1   0.5   0.5   0.4   0.6   0.5   0.5   0.6              # X_attr
  0.0   0.1   0.25  0.25  0.2   0.3   0.15  0.25  0.3              # Y_attr
```

```
  0    1    1     1   0    0    0    0     0                          # Behind_bal
 0.0 -52.5 -45.0 -45.0  -45.0 -30.0 -30.0 -45.0   10.0               # X_min
 0.0 -30.0   0.0   0.0   35.0  42.0  42.0  47.0   47.0               # X_max


# Formation 5 = FT_open_defensive = 442
5
-45.0 -11.0 -12.0 -12.0 -11.0   8.0   2.0   2.0   8.0  20.0  20.0   # X_pos
 0.0  18.0   7.0  -7.0 -18.0  25.0  10.0 -10.0 -25.0  10.0 -10.0  # Y_pos
  1    4    2     2    4    6    5     5    6    7     7                # P_type
 0.0   0.1   0.4   0.4   0.6   0.5   0.5   0.5                         # X_attr
 0.0   0.1   0.25  0.25  0.25  0.25  0.25  0.25                        # Y_attr
  0    1    1     1    0    0    0    0                               # Behind_bal
 0.0 -52.5 -45.0 -45.0   -45.0 -30.0 -30.0 -45.0                      # X_min
 0.0 -30.0   0.0  0.0  35.0  42.0  42.0  47.0                         # X_max

# Formation 6 = FT_343_attacking
6
-50.0 -16.0 -21.0  -3.0 -16.0   3.0   3.0  -3.0  15.0  18.0  18.0  # X_pos
 0.0  10.0   0.0   5.0 -10.0 -11.0  11.0  -5.0  -0.5  19.0 -19.0   # Y_pos
  1    4    2     5    4    6    6    5    8    7     7                # P_type
 0.0   0.1   0.6   0.6   0.6   0.6   0.6   0.5   0.6                 # X_attr
 0.0   0.1   0.25  0.25  0.25  0.25  0.3  0.2  0.2                   # Y_attr
  0    1    1     1   0    0   0    0    0                          # Behind_ball
 0.0 -50.5 -45.0 -45.0 -45.0 -30.0 -30.0 -45.0   -2.0               # X_min
 0.0 -30.0   0.0   0.0  2.0   20.0  42.0  44.0   45.0               # X_max
```

player_conf_thr:0.85
player_high_conf_thr:0.94
ball_conf_thr:0.90
player_dist_tolerance:10.0
player_when_to_turn:7.0
player_when_to_kick:0.90
player_when_to_intercept:40
clear_ball_dist:5.0
clear_ball_opp_max_dist:30.0
clear_ball_side_angle:17.0
cone_width:0.5
pass_end_speed:1.7
fast_pass_end_speed:1.9
pass_extra_x:0.0
wait_no_see:0.61
wait_see_begin:0.7
wait_see_end:0.85
mark_distance:5.0
strat_area_radius:5.0
shoot_risk_prob:0.88
cycles_catch_wait:20
server_time_out:5
dribble_ang_thr:20.0
turn_with_ball_ang_thr:45.0
turn_with_ball_freeze_thr:0.2
initial_formation:2
max_y_percentage:0.8

serverparam.h

```
#ifndef _SERVER_PARAM_H_
#define _SERVER_PARAM_H_

#include <sys/param.h> /* needed for htonl, htons, ... */
#include <netinet/in.h>
#include <sstream>

#include "paramreader.h"
#include "utility.h"
#include "types.h"

class ServerParam
  : public ParamReader
{
protected:

  ServerParam ();
private:

  ServerParam( const ServerParam& )
    : ParamReader()
  {}

protected:

public:

  inline
  static
  ServerParam&
  instance()
  { static ServerParam rval; return rval; }

  inline
  static
  ServerParam&
  init( const int& argc, const char * const *argv )
  {
    instance().getOptions( argc, argv );
    return instance();
  }

public:

  virtual ~ServerParam ();
```

```
  server_params_t convertToStruct ();



protected:
  virtual void getOptions(const int& argc, const char * const *argv );
  virtual void writeConfig ( std::ostream& o );

  virtual
  void
  setDefaults();

  virtual
  void
  createMaps();

public:

                Value gwidth ;                          /* goal width */
                Value inertia_moment ;                  /* intertia moment for turn */
                Value psize ;                           /* player size */
                Value pdecay ;                          /* player decay */
                Value prand ;                           /* player rand */
                Value pweight ;                                 /* player weight */
                Value pspeed_max ;                      /* player speed max */
                // th 6.3.00
                Value paccel_max ;                      /* player acceleration max */
                //
                Value stamina_max ;                     /* player stamina max */
                Value stamina_inc ;                     /* player stamina inc */
                Value recover_init ;            /* player recovery init */
                Value recover_dthr ;            /* player recovery decriment threshold */
                Value recover_min ;                     /* player recovery min */
                Value recover_dec ;                     /* player recovery decriment */
                Value effort_init ;                     /* player dash effort init */
                Value effort_dthr ;                     /* player dash effort decriment
threshold */
                Value effort_min ;                      /* player dash effrot min */
                Value effort_dec ;                      /* player dash effort decriment */
                Value effort_ithr ;                     /* player dash effort incriment
threshold */
                Value effort_inc ;                      /* player dash effort incriment */
                // pfr 8/14/00: for RC2000 evaluation
                Value kick_rand;                /* noise added directly to kicks */
                bool team_actuator_noise;               /* flag whether to use team specific
actuator noise */
```

```
        Value prand_factor_l;           /* factor to multiple prand for left team */
        Value prand_factor_r;           /* factor to multiple prand for right team */
        Value kick_rand_factor_l;       /* factor to multiple kick_rand for left
team */
        Value kick_rand_factor_r;       /* factor to multiple kick_rand for right
team */

        Value bsize ;                   /* ball size */
        Value bdecay ;                  /* ball decay */
        Value brand ;                   /* ball rand */
        Value bweight ;                 /* ball weight */
        Value bspeed_max ;              /* ball speed max */
        // th 6.3.00
        Value baccel_max;               /* ball acceleration max */
        //
        Value dprate ;                  /* dash power rate */
        Value kprate ;                  /* kick power rate */
        Value kmargin ;                 /* kickable margin */
        Value ctlradius ;               /* control radius */
        Value ctlradius_width ;         /* (control radius) - (plyaer size) */
        Value maxp ;                    /* max power */
        Value minp ;                    /* min power */
        Value maxm ;                    /* max moment */
        Value minm ;                    /* min moment */
        Value maxnm ;                   /* max neck moment */
        Value minnm ;                   /* min neck moment */
        Value maxn ;                    /* max neck angle */
        Value minn ;                    /* min neck angle */
private:
        Value visangle ;                /* visible angle */
public:
        Value visdist ;                 /* visible distance */
        Angle windir ;                  /* wind direction */
        Value winforce ;                /* wind force */
        Value winang ;                  /* wind angle for rand */
        Value winrand ;                 /* wind force for force */
        Value kickable_area ;       /* kickable_area */
        Value catch_area_l ;        /* goalie catchable area length */
        Value catch_area_w ;        /* goalie catchable area width */
        Value catch_prob ;              /* goalie catchable possibility */
    int   goalie_max_moves;     /* goalie max moves after a catch */
    bool kaway ;                /* keepaway mode on/off */
    Value ka_length ;           /* keepaway region length */
    Value ka_width ;            /* keepaway region width */
        Value ckmargin ;                /* corner kick margin */
        Value offside_area ;        /* offside active area size */
```

```
        bool win_no ;                   /* wind factor is none */
        bool win_random ;               /* wind factor is random */
        int portnum ;                   /* port number */
        int coach_pnum ;                /* coach port number */
        int olcoach_pnum ;              /* online coach port number */
        int say_cnt_max ;               /* max count of coach SAY
(freeform) */
        int SayCoachMsgSize ;                   /* max length of coach SAY
(freeform) */
        int clang_win_size;           /* coach language: time window size */
        int clang_define_win;          /* coach language: define messages per
window */
        int clang_meta_win;           /* coach language: meta messages per
window */
        int clang_advice_win;          /* coach language: advice messages per
window */
        int clang_info_win;           /* coach language: info messages per window
*/
        int clang_mess_delay;          /* coach language: delay between receive and
send */
        int clang_mess_per_cycle;        /* coach language: number of messages to
flush per cycle */

        int half_time ;                 /* half time */
        int drop_time;                /* cycles for dropping


    the ball after a free kick,


    corner kick message and


    noone kicking the ball.


    0 means don't drop


    automatically  */

        int sim_st ;                    /* simulator step interval msec */
        int send_st ;                   /* udp send step interval msec */
        int recv_st ;                   /* udp recv step interval msec */
        int sb_step ;                   /* sense_body interval step msec */
        int lcm_st ;                  /* lcm of all the above steps msec */
```

```
        int cban_cycle ;                    /* goalie catch ban cycle */
        int slow_down_factor ;              /* factor to slow down simulator and send
intervals */
        bool useoffside ;                   /* flag for using off side rule */
        bool kickoffoffside ;               /* flag for permit kick off offside */
        Value offside_kick_margin ;         /* offside kick margin */
        Value audio_dist ;                  /* audio cut off distance */
        Value dist_qstep ;                  /* quantize step of distance */
        Value land_qstep ;                  /* quantize step of distance for
landmark */
#ifdef NEW_QSTEP
        Value dir_qstep ;                   /* quantize step of direction */
        Value dist_qstep_l ;                /* team right quantize step of distance */
        Value dist_qstep_r ;                /* team left quantize step of distance */
        Value land_qstep_l ;                /* team right quantize step of distance for
landmark */
        Value land_qstep_r ;                /* team left quantize step of distance for
landmark */
        Value dir_qstep_l ;                 /* team left quantize step of direction
*/
        Value dir_qstep_r ;                 /* team right quantize step of
direction */
#endif
        bool    verbose ;                           /* flag for verbose mode */

        //   char logfile[MAX_FILE_LEN] ; /* server log file name */
        //  char recfile[MAX_FILE_LEN] ;  /* log file name for recording */
        //  int  rec_ver ;                          /* version for recording */
        //  int  rec_log ;                          /* flag for log recording */
        //  int  rec_msg_mode ;                     /* flag for whether to record
msg_mode structures */
        //   int  send_log ;                        /* flag for log sending */
        // th 6.3.00
        //  int  log_times ;                    /* flag for logging server cycle times
in the log file */
        //
//      string replay ;         /* log file name for replay */

        bool CoachMode ;                    /* coach mode */
        bool CwRMode ;                          /* coach with referee mode */
        bool old_hear ;                         /* old format for hear
command (coach) */
        int sv_st ;                         /* online coach's look interval step */

        bool synch_mode;                /* pfr:SYNCH whether to run in
synchronized mode */
```

```
        int synch_offset;               /* pfr:SYNCH the offset from the start of the
cycle to tell players to run */
        int synch_micro_sleep;          /* pfr:SYNCH the # of microseconds to sleep
while waiting for players */

        int start_goal_l;               /* The starting score of the left team */
        int start_goal_r;               /* The starting score of the right team */

        bool fullstate_l;               /* flag to send fullstate messages to left team;
supresses visual info */
        bool fullstate_r;               /* flag to send fullstate messages to right team;
supresses visual info */

        double slowness_on_top_for_left_team; /* Kinda self explanatory */
        double slowness_on_top_for_right_team; /* ditto */


private:
        std::string landmark_file;

        bool send_comms;
        bool text_logging;
        bool game_logging;
        int game_log_version;
        std::string text_log_dir;
        std::string game_log_dir;
        std::string text_log_fixed_name;
        std::string game_log_fixed_name;
        bool text_log_fixed;
        bool game_log_fixed;
        bool text_log_dated;
        bool game_log_dated;
        std::string log_date_format;
        bool log_times;
        bool record_messages;
        int text_log_compression;
        int game_log_compression;
        bool M_profile;

  bool kaway_logging;
  std::string kaway_log_dir;
  std::string kaway_log_fixed_name;
  bool kaway_log_fixed;
  bool kaway_log_dated;

  int kaway_start;
```

```
int M_point_to_ban;
int M_point_to_duration;

int M_say_msg_size;                 /* string size of say message */
int M_hear_max;                         /* player hear_capacity_max */
int M_hear_inc;                         /* player hear_capacity_inc */
int M_hear_decay;                   /* player hear_capacity_decay */

double M_tackle_dist;
double M_tackle_back_dist;
double M_tackle_width;
double M_tackle_exponent;
int M_tackle_cycles;
double M_tackle_power_rate;

int M_freeform_wait_period;
int M_freeform_send_period;

bool M_free_kick_faults;
bool M_back_passes;

bool M_proper_goal_kicks;
double M_stopped_ball_vel;
int M_max_goal_kicks;

int M_clang_del_win;
int M_clang_rule_win;

bool M_auto_mode;
int M_kick_off_wait;
int M_connect_wait;
int M_game_over_wait;

std::string M_team_l_start;
std::string M_team_r_start;

static const std::string LANDMARK_FILE;
static const std::string SERVER_CONF;

static const int SEND_COMMS;
static const int TEXT_LOGGING;
static const int GAME_LOGGING;
static const int GAME_LOG_VERSION;
static const std::string TEXT_LOG_DIR;
static const std::string GAME_LOG_DIR;
```

```
static const std::string TEXT_LOG_FIXED_NAME;
static const std::string GAME_LOG_FIXED_NAME;
static const int TEXT_LOG_FIXED;
static const int GAME_LOG_FIXED;
static const int TEXT_LOG_DATED;
static const int GAME_LOG_DATED;
static const std::string LOG_DATE_FORMAT;
static const int LOG_TIMES;
static const int RECORD_MESSAGES;
static const int TEXT_LOG_COMPRESSION;
static const int GAME_LOG_COMPRESSION;
static const bool PROFILE;

static const int KAWAY_LOGGING;
static const std::string KAWAY_LOG_DIR;
static const std::string KAWAY_LOG_FIXED_NAME;
static const int KAWAY_LOG_FIXED;
static const int KAWAY_LOG_DATED;

static const int KAWAY_START;

static const int POINT_TO_BAN;
static const int POINT_TO_DURATION;

static const unsigned int SAY_MSG_SIZE;
static const unsigned int HEAR_MAX;
static const unsigned int HEAR_INC;
static const unsigned int HEAR_DECAY;

static const double TACKLE_DIST;
static const double TACKLE_BACK_DIST;
static const double TACKLE_WIDTH;
static const double TACKLE_EXPONENT;
static const unsigned int TACKLE_CYCLES;
static const double TACKLE_POWER_RATE;


static const unsigned int FREEFORM_WAIT_PERIOD;
static const unsigned int FREEFORM_SEND_PERIOD;

static const bool FREE_KICK_FAULTS;
static const bool BACK_PASSES;

static const bool PROPER_GOAL_KICKS;
static const double STOPPED_BALL_VEL;
static const int MAX_GOAL_KICKS;
```

```
  static const int CLANG_DEL_WIN;
  static const int CLANG_RULE_WIN;

  static const bool S_AUTO_MODE;
  static const int S_KICK_OFF_WAIT;
  static const int S_CONNECT_WAIT;
  static const int S_GAME_OVER_WAIT;

  static const std::string S_TEAM_L_START;
  static const std::string S_TEAM_R_START;


public:

                std::string landmarkFile () const { return landmark_file; }

                int sendComms () const { return send_comms; }
                int textLogging () const { return text_logging; }
                int gameLogging () const { return game_logging; }
                int gameLogVersion () const { return game_log_version; }
                std::string textLogDir () const { return text_log_dir; }
                std::string gameLogDir () const { return game_log_dir; }
                std::string textLogFixedName () const { return text_log_fixed_name; }
                std::string gameLogFixedName () const { return game_log_fixed_name; }
                int textLogFixed () const { return text_log_fixed; }
                int gameLogFixed () const { return game_log_fixed; }
                int textLogDated () const { return text_log_dated; }
                int gameLogDated () const { return game_log_dated; }
                std::string logDateFormat () const { return log_date_format; }
                int logTimes () const { return log_times; }
                int recordMessages () const { return record_messages; }
                int textLogCompression () const { return text_log_compression; }
                int gameLogCompression () const { return game_log_compression; }

                bool profile () const { return M_profile; }

  int kawayLogging () const { return kaway_logging; }
  std::string kawayLogDir () const { return kaway_log_dir; }
  std::string kawayLogFixedName () const { return kaway_log_fixed_name; }
  int kawayLogFixed () const { return kaway_log_fixed; }
  int kawayLogDated () const { return kaway_log_dated; }

  int kawayStart () const { return kaway_start; }

  unsigned int
```

```
pointToBan () const
{ return (unsigned int)M_point_to_ban; }

unsigned int
pointToDuration () const
{ return (unsigned int)M_point_to_duration; }

unsigned int
sayMsgSize() const
{ return (unsigned int)M_say_msg_size; }

unsigned int
hearMax() const
{ return (unsigned int)M_hear_max; }

unsigned int
hearInc() const
{ return (unsigned int)M_hear_inc; }

unsigned int
hearDecay() const
{ return (unsigned int)M_hear_decay; }

double
tackleDist() const
{ return M_tackle_dist; }

double
tackleBackDist() const
{ return M_tackle_back_dist; }

double
tackleWidth() const
{ return M_tackle_width; }

double
tackleExponent() const
{ return M_tackle_exponent; }

unsigned int
tackleCycles() const
{ return (unsigned int)M_tackle_cycles; }

double
tacklePowerRate() const
{ return M_tackle_power_rate; }
```

```
unsigned int
freeformWaitPeriod() const
{ return (unsigned int)M_freeform_wait_period; }

unsigned int
freeformSendPeriod() const
{ return (unsigned int)M_freeform_send_period; }

double
visAngle() const
{
  static double rval = Deg2Rad(visangle);
  return rval;
}

double
visAngleDeg() const
{ return visangle; }

bool
freeKickFaults() const
{ return M_free_kick_faults; }

bool
backPasses() const
{ return M_back_passes; }

bool
properGoalKicks() const
{ return M_proper_goal_kicks; }

double
stoppedBallVel() const
{ return M_stopped_ball_vel; }

int
maxGoalKicks() const
{ return M_max_goal_kicks; }

int
clangDelWin() const
{ return M_clang_del_win; }

        int
```

```
 clangRuleWin() const
 { return M_clang_rule_win; }

 bool
 autoMode() const
 { return M_auto_mode; }

 int
 kickOffWait() const
 { return M_kick_off_wait * sim_st / recv_st; }

 int
 connectWait() const
 { return M_connect_wait * sim_st / recv_st; }

 int
 gameOverWait() const
 { return M_game_over_wait; }

 std::string
 teamLeftStart() const
 { return M_team_l_start; }

 std::string
 teamRightStart() const
 { return M_team_r_start; }

   double
   goalPostRadius() const
   { return 0.06; }
};


class ServerParamSensor
{
};

class ServerParamSensor_v7
        : public ServerParamSensor
{
public:
        struct data_t
        {
                double M_gwidth;
                double M_inertia_moment ;
                double M_psize;
```

```
double M_pdecay;
double M_prand;
double M_pweight;
double M_pspeed_max;
double M_paccel_max;
double M_stamina_max;
double M_stamina_inc;
double M_recover_init;
double M_recover_dthr;
double M_recover_min;
double M_recover_dec;
double M_effort_init;
double M_effort_dthr;
double M_effort_min;
double M_effort_dec;
double M_effort_ithr;
double M_effort_inc;
double M_kick_rand;
bool M_team_actuator_noise;
double M_prand_factor_l;
double M_prand_factor_r;
double M_kick_rand_factor_l;
double M_kick_rand_factor_r;
double M_bsize;
double M_bdecay;
double M_brand;
double M_bweight;
double M_bspeed_max;
double M_baccel_max;
double M_dprate;
double M_kprate;
double M_kmargin;
double M_ctlradius;
double M_ctlradius_width;
double M_maxp;
double M_minp;
double M_maxm;
double M_minm;
double M_maxnm;
double M_minnm;
double M_maxn;
double M_minn;
double M_visangle;
double M_visdist;
double M_windir;
double M_winforce;
```

```
double M_winang;
double M_winrand;
double M_kickable_area;
double M_catch_area_l;
double M_catch_area_w;
double M_catch_prob;
int M_goalie_max_moves;
double M_ckmargin;
double M_offside_area;
bool M_win_no;
bool M_win_random;
int M_say_cnt_max;
int M_SayCoachMsgSize;
int M_clang_win_size;
int M_clang_define_win;
int M_clang_meta_win;
int M_clang_advice_win;
int M_clang_info_win;
int M_clang_mess_delay;
int M_clang_mess_per_cycle;
int M_half_time;
int M_sim_st;
int M_send_st;
int M_recv_st;
int M_sb_step;
int M_lcm_st;
int M_say_msg_size;
int M_hear_max;
int M_hear_inc;
int M_hear_decay;
int M_cban_cycle;
int M_slow_down_factor;
bool M_useoffside;
bool M_kickoffoffside;
double M_offside_kick_margin;
double M_audio_dist;
double M_dist_qstep;
double M_land_qstep;
double M_dir_qstep;
double M_dist_qstep_l;
double M_dist_qstep_r;
double M_land_qstep_l;
double M_land_qstep_r;
double M_dir_qstep_l;
double M_dir_qstep_r;
bool M_CoachMode;
```

```
bool M_CwRMode;
bool M_old_hear;
int M_sv_st;
int M_start_goal_l;
int M_start_goal_r;
bool M_fullstate_l;
bool M_fullstate_r;
int M_drop_time;



data_t ( const ServerParam& sp )
        : M_gwidth ( sp.gwidth ),
                M_inertia_moment  ( sp.inertia_moment ),
                M_psize ( sp.psize ),
                M_pdecay ( sp.pdecay ),
                M_prand ( sp.prand ),
                M_pweight ( sp.pweight ),
                M_pspeed_max ( sp.pspeed_max ),
                M_paccel_max ( sp.paccel_max ),
                M_stamina_max ( sp.stamina_max ),
                M_stamina_inc ( sp.stamina_inc ),
                M_recover_init ( sp.recover_init ),
                M_recover_dthr ( sp.recover_dthr ),
                M_recover_min ( sp.recover_min ),
                M_recover_dec ( sp.recover_dec ),
                M_effort_init ( sp.effort_init ),
                M_effort_dthr ( sp.effort_dthr ),
                M_effort_min ( sp.effort_min ),
                M_effort_dec ( sp.effort_dec ),
                M_effort_ithr ( sp.effort_ithr ),
                M_effort_inc ( sp.effort_inc ),
                M_kick_rand ( sp.kick_rand ),
                M_team_actuator_noise ( sp.team_actuator_noise ),
                M_prand_factor_l ( sp.prand_factor_l ),
                M_prand_factor_r ( sp.prand_factor_r ),
                M_kick_rand_factor_l ( sp.kick_rand_factor_l ),
                M_kick_rand_factor_r ( sp.kick_rand_factor_r ),
                M_bsize ( sp.bsize ),
                M_bdecay ( sp.bdecay ),
                M_brand ( sp.brand ),
                M_bweight ( sp.bweight ),
                M_bspeed_max ( sp.bspeed_max ),
                M_baccel_max ( sp.baccel_max ),
                M_dprate ( sp.dprate ),
                M_kprate ( sp.kprate ),
```

M_kmargin (sp.kmargin),
M_ctlradius (sp.ctlradius),
M_ctlradius_width (sp.ctlradius_width),
M_maxp (sp.maxp),
M_minp (sp.minp),
M_maxm (sp.maxm),
M_minm (sp.minm),
M_maxnm (sp.maxnm),
M_minnm (sp.minnm),
M_maxn (sp.maxn),
M_minn (sp.minn),
M_visangle (sp.visAngleDeg()),
M_visdist (sp.visdist),
M_windir (sp.windir),
M_winforce (sp.winforce),
M_winang (sp.winang),
M_winrand (sp.winrand),
M_kickable_area (sp.kickable_area),
M_catch_area_l (sp.catch_area_l),
M_catch_area_w (sp.catch_area_w),
M_catch_prob (sp.catch_prob),
M_goalie_max_moves (sp.goalie_max_moves),
M_ckmargin (sp.ckmargin),
M_offside_area (sp.offside_area),
M_win_no (sp.win_no),
M_win_random (sp.win_random),
M_say_cnt_max (sp.say_cnt_max),
M_SayCoachMsgSize (sp.SayCoachMsgSize),
M_clang_win_size (sp.clang_win_size),
M_clang_define_win (sp.clang_define_win),
M_clang_meta_win (sp.clang_meta_win),
M_clang_advice_win (sp.clang_advice_win),
M_clang_info_win (sp.clang_info_win),
M_clang_mess_delay (sp.clang_mess_delay),
M_clang_mess_per_cycle (sp.clang_mess_per_cycle),
M_half_time (sp.half_time),
M_sim_st (sp.sim_st),
M_send_st (sp.send_st),
M_recv_st (sp.recv_st),
M_sb_step (sp.sb_step),
M_lcm_st (sp.lcm_st),
M_say_msg_size (sp.sayMsgSize()),
M_hear_max (sp.hearMax()),
M_hear_inc (sp.hearInc()),
M_hear_decay (sp.hearDecay()),
M_cban_cycle (sp.cban_cycle),

```
                                M_slow_down_factor ( sp.slow_down_factor ),
                                M_useoffside ( sp.useoffside ),
                                M_kickoffoffside ( sp.kickoffoffside ),
                                M_offside_kick_margin ( sp.offside_kick_margin ),
                                M_audio_dist ( sp.audio_dist ),
                                M_dist_qstep ( sp.dist_qstep ),
                                M_land_qstep ( sp.land_qstep ),
#ifdef NEW_QSTEP
                                M_dir_qstep ( sp.dir_qstep ),
                                M_dist_qstep_l ( sp.dist_qstep_l ),
                                M_dist_qstep_r ( sp.dist_qstep_r ),
                                M_land_qstep_l ( sp.land_qstep_l ),
                                M_land_qstep_r ( sp.land_qstep_r ),
                                M_dir_qstep_l ( sp.dir_qstep_l ),
                                M_dir_qstep_r ( sp.dir_qstep_r ),
#else
                                M_dir_qstep ( -1 ),
                                M_dist_qstep_l ( -1 ),
                                M_dist_qstep_r ( -1 ),
                                M_land_qstep_l ( -1 ),
                                M_land_qstep_r ( -1 ),
                                M_dir_qstep_l ( -1 ),
                                M_dir_qstep_r ( -1 ),
#endif
                                M_CoachMode ( sp.CoachMode ),
                                M_CwRMode ( sp.CwRMode ),
                                M_old_hear ( sp.old_hear ),
                                M_sv_st ( sp.sv_st ),
                                M_start_goal_l ( sp.start_goal_l ),
                                M_start_goal_r ( sp.start_goal_r ),
                                M_fullstate_l ( sp.fullstate_l ),
                                M_fullstate_r ( sp.fullstate_r ),
                                M_drop_time ( sp.drop_time )
                        {
                        }
        };

        virtual void send ( const ServerParamSensor_v7::data_t& data ) = 0;
};

std::ostream& toStr ( std::ostream& o, const ServerParamSensor_v7::data_t& data );

class ServerParamSensor_v8
        : public ServerParamSensor_v7
{
public:
```

```
        struct data_t
        {
    const std::map< std::string, ServerParam::DataHolder< int* > >&                int_map;
    const std::map< std::string, ServerParam::DataHolder< std::string* > >&        str_map;
    const std::map< std::string, ServerParam::DataHolder< bool* > >&       bool_map;
    const std::map< std::string, ServerParam::DataHolder< bool* > >&       onoff_map;
    const std::map< std::string, ServerParam::DataHolder< double* > >&    double_map;

    data_t( const ServerParam& sp )
      : int_map( sp.intMap() ),
        str_map( sp.strMap() ),
        bool_map( sp.boolMap() ),
        onoff_map( sp.onOffMap() ),
        double_map( sp.doubleMap() )
    {}

    //  double M_slowness_on_top_for_left_team;
//      double M_slowness_on_top_for_right_team;
//                  int M_portnum;
//                  int M_coach_pnum;
//                  int M_olcoach_pnum;
//                  bool M_verbose;
//                  std::string M_replay;
//                  bool M_synch_mode;
//                  int M_synch_offset;
//                  int M_synch_micro_sleep;
//                  bool M_send_comms;
//                  bool M_text_logging;
//                  bool M_game_logging;
//                  int M_game_log_version;
//                  std::string M_text_log_dir;
//                  std::string M_game_log_dir;
//                  std::string M_text_log_fixed_name;
//                  std::string M_game_log_fixed_name;
//                  bool M_text_log_fixed;
//                  bool M_game_log_fixed;
//                  bool M_text_log_dated;
//                  bool M_game_log_dated;
//                  std::string M_log_date_format;
//                  bool M_log_times;
//                  bool M_record_messages;
//                  int M_text_log_compression;
//                  int M_game_log_compression;
//                  int M_profile;
//                  int M_point_to_ban;
//                  int M_point_to_duration;
```

```
//                  data_t ( const ServerParam& sp )
//                          : ServerParamSensor_v7::data_t ( sp ),
//                                  M_slowness_on_top_for_left_team (
sp.slowness_on_top_for_left_team ),
//                                   M_slowness_on_top_for_right_team (
sp.slowness_on_top_for_right_team ),
//                                  M_portnum ( sp.portnum ),
//                                  M_coach_pnum ( sp.coach_pnum ),
//                                  M_olcoach_pnum ( sp.olcoach_pnum ),
//                                  M_verbose ( sp.verbose ),
//                                  M_replay ( sp.replay ),
//                                  M_synch_mode ( sp.synch_mode ),
//                                  M_synch_offset ( sp.synch_offset ),
//                                  M_synch_micro_sleep ( sp.synch_micro_sleep ),
//                                  M_send_comms ( sp.sendComms () ),
//                                  M_text_logging ( sp.textLogging () ),
//                                  M_game_logging ( sp.gameLogging () ),
//                                  M_game_log_version ( sp.gameLogVersion () ),
//                                  M_text_log_dir ( sp.textLogDir () ),
//                                  M_game_log_dir ( sp.gameLogDir () ),
//                                  M_text_log_fixed_name ( sp.textLogFixedName () ),
//                                  M_game_log_fixed_name ( sp.gameLogFixedName () ),
//                                  M_text_log_fixed ( sp.textLogFixed () ),
//                                  M_game_log_fixed ( sp.gameLogFixed () ),
//                                  M_text_log_dated ( sp.textLogDated () ),
//                                  M_game_log_dated ( sp.gameLogDated () ),
//                                  M_log_date_format ( sp.logDateFormat () ),
//                                  M_log_times ( sp.logTimes () ),
//                                  M_record_messages ( sp.recordMessages () ),
//                                  M_text_log_compression ( sp.textLogCompression () ),
//                                  M_game_log_compression ( sp.gameLogCompression () ),
//          M_profile ( sp.gameLogCompression () ),
//          M_point_to_ban ( sp.pointToBan () ),
//          M_point_to_duration ( sp.pointToDuration () )
//                          {
//                          }
          };

          virtual void send ( const ServerParamSensor_v8::data_t& data ) = 0;


};

std::ostream& toStr ( std::ostream& o, const ServerParamSensor_v8::data_t& data );
```

```
#endif
```

serverparam.cpp

```
#include <cstring>

#include "serverparam.h"
#include <string>
#include "config.h"

#define name_of(var) (#var)
#define str_of(var)      (string(#var))


const std::string ServerParam::LANDMARK_FILE = "~/.rcssserver-landmark.xml";
const std::string ServerParam::SERVER_CONF = "~/.rcssserver-server.conf";

const int ServerParam::SEND_COMMS = false;

const int ServerParam::TEXT_LOGGING = true;
const int ServerParam::GAME_LOGGING = true;
const int ServerParam::GAME_LOG_VERSION = 3;
const std::string ServerParam::TEXT_LOG_DIR = "./";
const std::string ServerParam::GAME_LOG_DIR = "./";
const std::string ServerParam::TEXT_LOG_FIXED_NAME = "rcssserver";
const std::string ServerParam::GAME_LOG_FIXED_NAME = "rcssserver";
const int ServerParam::TEXT_LOG_FIXED = false;
const int ServerParam::GAME_LOG_FIXED = false;
const int ServerParam::TEXT_LOG_DATED = true;
const int ServerParam::GAME_LOG_DATED = true;
const std::string ServerParam::LOG_DATE_FORMAT = "%Y%m%d%H%M-";
const int ServerParam::LOG_TIMES = false;
const int ServerParam::RECORD_MESSAGES = false;
const int ServerParam::TEXT_LOG_COMPRESSION = 0;
const int ServerParam::GAME_LOG_COMPRESSION = 0;
const bool ServerParam::PROFILE = false;

const int ServerParam::KAWAY_LOGGING = true;
const std::string ServerParam::KAWAY_LOG_DIR = "./";
const std::string ServerParam::KAWAY_LOG_FIXED_NAME = "rcssserver";
const int ServerParam::KAWAY_LOG_FIXED = false;
const int ServerParam::KAWAY_LOG_DATED = true;

const int ServerParam::KAWAY_START = -1;

const int ServerParam::POINT_TO_BAN = 5;
const int ServerParam::POINT_TO_DURATION= 20;
const unsigned int ServerParam::SAY_MSG_SIZE = 10;
const unsigned int ServerParam::HEAR_MAX = 1;
const unsigned int ServerParam::HEAR_INC = 1;
```

```
const unsigned int ServerParam::HEAR_DECAY = 1;

const double ServerParam::TACKLE_DIST = 2.5;
const double ServerParam::TACKLE_BACK_DIST = 0.5;
const double ServerParam::TACKLE_WIDTH = 1.25;
const double ServerParam::TACKLE_EXPONENT = 6.0;
const unsigned int ServerParam::TACKLE_CYCLES = 10;
const double ServerParam::TACKLE_POWER_RATE = 0.027;

const unsigned int ServerParam::FREEFORM_WAIT_PERIOD = 600;
const unsigned int ServerParam::FREEFORM_SEND_PERIOD = 20;

const bool ServerParam::FREE_KICK_FAULTS = true;
const bool ServerParam::BACK_PASSES = true;

const bool ServerParam::PROPER_GOAL_KICKS = false;
const double ServerParam::STOPPED_BALL_VEL = 0.01;
const int ServerParam::MAX_GOAL_KICKS = 3;

const int ServerParam::CLANG_DEL_WIN = 1;
const int ServerParam::CLANG_RULE_WIN = 1;

const bool ServerParam::S_AUTO_MODE = false;
const int ServerParam::S_KICK_OFF_WAIT = 100;
const int ServerParam::S_CONNECT_WAIT = 300;
const int ServerParam::S_GAME_OVER_WAIT = 100;

const std::string ServerParam::S_TEAM_L_START = "";
const std::string ServerParam::S_TEAM_R_START = "";

ServerParam::ServerParam()
{
  setDefaults();
  createMaps();
}

ServerParam::~ServerParam()
{
}

void
ServerParam::createMaps()
{
  add2Maps ( "goal_width", group( &gwidth, 7 ) );
  add2Maps ( "player_size", group( &psize, 7 ) );
  add2Maps ( "player_decay", group( &pdecay, 7 ) );
```

```
add2Maps ( "player_rand", group( &prand, 7 ) );
add2Maps ( "player_weight", group( &pweight, 7 ) );
add2Maps ( "player_speed_max", group( &pspeed_max, 7 ) );
// th 6.3.00
add2Maps ( "player_accel_max", group( &paccel_max, 7 ) );
//
add2Maps ( "stamina_max", group( &stamina_max, 7 ) );
add2Maps ( "stamina_inc_max", group( &stamina_inc, 7 ) );
add2Maps ( "recover_dec_thr", group( &recover_dthr, 7 ) );
add2Maps ( "recover_min", group( &recover_min, 7 ) );
add2Maps ( "recover_dec", group( &recover_dec, 7 ) );
add2Maps ( "effort_init", group( &effort_init, 7 ) );
add2Maps ( "effort_dec_thr", group( &effort_dthr, 7 ) );
add2Maps ( "effort_min", group( &effort_min, 7 ) );
add2Maps ( "effort_dec", group( &effort_dec, 7 ) );
add2Maps ( "effort_inc_thr", group( &effort_ithr, 7 ) );
add2Maps ( "effort_inc", group( &effort_inc, 7 ) );
// pfr 8/14/00: for RC2000 evaluation
add2Maps ( "kick_rand",           group( &kick_rand, 7 ) );
add2Maps ( "team_actuator_noise", group( &team_actuator_noise, 7 ), true );
add2Maps ( "prand_factor_l",      group( &prand_factor_l, 7 ) );
add2Maps ( "prand_factor_r",      group( &prand_factor_r, 7 ) );
add2Maps ( "kick_rand_factor_l",  group( &kick_rand_factor_l, 7 ) );
add2Maps ( "kick_rand_factor_r",  group( &kick_rand_factor_r, 7 ) );

add2Maps ( "ball_size", group( &bsize, 7 ) );
add2Maps ( "ball_decay", group( &bdecay, 7 ) );
add2Maps ( "ball_rand", group( &brand, 7 ) );
add2Maps ( "ball_weight", group( &bweight, 7 ) );
add2Maps ( "ball_speed_max", group( &bspeed_max, 7 ) );
// th 6.3.00
add2Maps ( "ball_accel_max", group( &baccel_max, 7 ) );
//
add2Maps ( "dash_power_rate", group( &dprate, 7 ) );
add2Maps ( "kick_power_rate", group( &kprate, 7 ));
add2Maps ( "kickable_margin", group( &kmargin, 7 ) );
add2Maps ( "control_radius", group( &ctlradius, 7 ) );

add2Maps ( "catch_probability", group( &catch_prob, 7 ) );
add2Maps ( "catchable_area_l",    group( &catch_area_l, 7 ) );
add2Maps ( "catchable_area_w",    group( &catch_area_w, 7 ) );
add2Maps ( "goalie_max_moves",    group( &goalie_max_moves, 7 ) );

add2Maps ( "maxpower", group( &maxp, 7 ) );
add2Maps ( "minpower", group( &minp, 7 ) );
add2Maps ( "maxmoment", group( &maxm, 7 ) );
```

```
  add2Maps ( "minmoment", group( &minm, 7 ) );
  add2Maps ( "maxneckmoment", group( &maxnm, 7 ) );
  add2Maps ( "minneckmoment", group( &minnm, 7 ) );
  add2Maps ( "maxneckang", group( &maxn, 7 ) );
  add2Maps ( "minneckang", group( &minn, 7 ) );
  add2Maps ( "visible_angle", group( &visangle, 7 ) );
  add2Maps ( "visible_distance", group( &visdist, 7 ) );
  add2Maps ( "audio_cut_dist", group( &audio_dist, 7 ) );
  add2Maps ( "quantize_step", group( &dist_qstep, 7 ) );
  add2Maps ( "quantize_step_l", group( &land_qstep, 7 ) );
#ifdef NEW_QSTEP
  add2Maps ( "quantize_step_dir", group( &dir_qstep, 7 ) );
  add2Maps ( "quantize_step_dist_team_l", group( &dist_qstep_l, 7 ) );
  add2Maps ( "quantize_step_dist_team_r", group( &dist_qstep_r, 7 ) );
  add2Maps ( "quantize_step_dist_l_team_l", group( &land_qstep_l, 7 ) );
  add2Maps ( "quantize_step_dist_l_team_r", group( &land_qstep_r, 7 ) );
  add2Maps ( "quantize_step_dir_team_l", group( &dir_qstep_l, 7 ) );
  add2Maps ( "quantize_step_dir_team_r", group( &dir_qstep_r, 7 ) );
#endif
  add2Maps ( "ckick_margin", group( &ckmargin, 7 ) );
  add2Maps ( "wind_dir", group( &windir, 7 ) );
  add2Maps ( "wind_force", group( &winforce, 7 ) );
  add2Maps ( "wind_ang", group( &winang, 7 ) );
  add2Maps ( "wind_rand", group( &winrand, 7 ) );
  add2Maps ( "inertia_moment", group( &inertia_moment, 7 ) );
  add2Maps ( "wind_none", group( &win_no, 7 ) );
  add2Maps ( "wind_random", group( &win_random, 7 ) );
  add2Maps ( "half_time", group( &half_time, 7 ) );
  add2Maps ( "drop_ball_time", group( &drop_time, 7 ) );
  add2Maps ( "port", group( &portnum, 8 ) );
  add2Maps ( "coach_port", group( &coach_pnum, 8 ) );
  add2Maps ( "olcoach_port", group( &olcoach_pnum, 8 ) );
  add2Maps ( "say_coach_cnt_max", group( &say_cnt_max, 7 ) );
  add2Maps ( "say_coach_msg_size", group( &SayCoachMsgSize, 7 ) );
  add2Maps ( "simulator_step", group( &sim_st, 7 ) );
  add2Maps ( "send_step", group( &send_st, 7 ) );
  add2Maps ( "recv_step", group( &recv_st, 7 ) );
  add2Maps ( "sense_body_step", group( &sb_step, 7 ) );
  add2Maps ( "say_msg_size", group( &M_say_msg_size, 7 ) );
  add2Maps ( "clang_win_size",      group( &clang_win_size, 7 ) );
  add2Maps ( "clang_define_win",    group( &clang_define_win, 7 ) );
  add2Maps ( "clang_meta_win",      group( &clang_meta_win, 7 ) );
  add2Maps ( "clang_advice_win",    group( &clang_advice_win, 7 ) );
  add2Maps ( "clang_info_win",      group( &clang_info_win, 7 ) );
  add2Maps ( "clang_del_win",      group( &M_clang_del_win, 8 ) );
  add2Maps ( "clang_rule_win",      group( &M_clang_rule_win, 8 ) );
```

```
  add2Maps ( "clang_mess_delay",    group( &clang_mess_delay, 7 ) );
  add2Maps ( "clang_mess_per_cycle",      group( &clang_mess_per_cycle, 7 ) );
  add2Maps ( "hear_max", group( &M_hear_max, 7 ) );
  add2Maps ( "hear_inc", group( &M_hear_inc, 7 ) );
  add2Maps ( "hear_decay", group( &M_hear_decay, 7 ) );
  add2Maps ( "catch_ban_cycle", group( &cban_cycle, 7 ) );
  add2Maps ( "coach", group( &CoachMode, 7 ) );
  add2Maps ( "coach_w_referee", group( &CwRMode, 7 ) );
  add2Maps ( "old_coach_hear", group( &old_hear, 7 ) );
  add2Maps ( "send_vi_step",        group( &sv_st, 7 ) );
  add2Maps ( "use_offside", group( &useoffside, 7 ), true );
  add2Maps ( "offside_active_area_size", group( &offside_area, 7 ) );
  add2Maps ( "forbid_kick_off_offside", group( &kickoffoffside, 7 ), true );
  add2Maps ( "verbose", group( &verbose, 8 ), true );
//    add2Maps ( "replay", group( &replay, 8 ) );
  add2Maps ( "offside_kick_margin", group( &offside_kick_margin, 7 ) );
  add2Maps ( "slow_down_factor",        group( &slow_down_factor, 7 ) );
  add2Maps ( "synch_mode",              group( &synch_mode, 8 ), true ); //pfr:SYNCH
  add2Maps ( "synch_offset",            group( &synch_offset, 8 ) ); //pfr:SYNCH
  add2Maps ( "synch_micro_sleep",       group( &synch_micro_sleep, 8 ) ); //pfr:SYNCH

  add2Maps ( "start_goal_l",            group( &start_goal_l, 7 ) );
  add2Maps ( "start_goal_r",            group( &start_goal_r, 7 ) );

  add2Maps ( "fullstate_l",             group( &fullstate_l, 7 ), true );
  add2Maps ( "fullstate_r",             group( &fullstate_r, 7 ), true );
  add2Maps ( "slowness_on_top_for_left_team", group( &slowness_on_top_for_left_team, 8 ) );
  add2Maps ( "slowness_on_top_for_right_team", group( &slowness_on_top_for_right_team, 8 )
);
  add2Maps ( "landmark_file",           group( &landmark_file, 8 ) );
  add2Maps ( "send_comms", group( &send_comms, 8 ), true );
  add2Maps ( "text_logging", group( &text_logging, 8 ), true );
  add2Maps ( "game_logging", group( &game_logging, 8 ), true );
  add2Maps ( "game_log_version", group( &game_log_version, 8 ) );
  add2Maps ( "text_log_dir", group( &text_log_dir, 8 ) );
  add2Maps ( "game_log_dir", group( &game_log_dir, 8 ) );
  add2Maps ( "text_log_fixed_name", group( &text_log_fixed_name, 8 ) );
  add2Maps ( "game_log_fixed_name", group( &game_log_fixed_name, 8 ) );
  add2Maps ( "text_log_fixed", group( &text_log_fixed, 8 ), true );
  add2Maps ( "game_log_fixed", group( &game_log_fixed, 8 ), true );
  add2Maps ( "text_log_dated", group( &text_log_dated, 8 ), true );
  add2Maps ( "game_log_dated", group( &game_log_dated, 8 ), true );
  add2Maps ( "log_date_format", group( &log_date_format, 8 ) );
  add2Maps ( "log_times", group( &log_times, 8 ), true );
  add2Maps ( "record_messages", group( &record_messages, 8 ), true );
  add2Maps ( "text_log_compression", group( &text_log_compression, 8 ) );
```

```
  add2Maps ( "game_log_compression", group( &game_log_compression, 8 ) );
  add2Maps ( "profile", group( &M_profile, 8 ), true );
  add2Maps ( "point_to_ban", group( &M_point_to_ban, 8 ) );
  add2Maps ( "point_to_duration", group( &M_point_to_duration, 8 ) );

  add2Maps ( "tackle_dist", group( &M_tackle_dist, 8 ) );
  add2Maps ( "tackle_back_dist", group( &M_tackle_back_dist, 8 ) );
  add2Maps ( "tackle_width", group( &M_tackle_width, 8 ) );
  add2Maps ( "tackle_exponent", group( &M_tackle_exponent, 8 ) );
  add2Maps ( "tackle_cycles", group( &M_tackle_cycles, 8 ) );
  add2Maps ( "tackle_power_rate", group( &M_tackle_power_rate, 8 ) );

  add2Maps ( "freeform_wait_period", group( &M_freeform_wait_period, 8 ) );
  add2Maps ( "freeform_send_period", group( &M_freeform_send_period, 8 ) );

  add2Maps ( "free_kick_faults", group( &M_free_kick_faults, 8 ), true );
  add2Maps ( "back_passes", group( &M_back_passes, 8 ), true );

  add2Maps ( "proper_goal_kicks", group( &M_proper_goal_kicks, 8 ), true );
  add2Maps ( "stopped_ball_vel", group( &M_stopped_ball_vel, 8 ) );
  add2Maps ( "max_goal_kicks", group( &M_max_goal_kicks, 8 ) );

  add2Maps( "auto_mode", group( &M_auto_mode, 9 ), true );
  add2Maps( "kick_off_wait", group( &M_kick_off_wait, 9 ) );
  add2Maps( "connect_wait", group( &M_connect_wait, 9 ) );
  add2Maps( "game_over_wait", group( &M_game_over_wait, 9 ) );

  add2Maps( "team_l_start", group( &M_team_l_start, 9 ) );
  add2Maps( "team_r_start", group( &M_team_r_start, 9 ) );

  add2Maps ( "keepaway", group( &kaway, 9 ), true );
  add2Maps ( "keepaway_length", group( &ka_length, 9 ) );
  add2Maps ( "keepaway_width", group( &ka_width, 9 ) );

  add2Maps ( "keepaway_logging", group( &kaway_logging, 9 ), true );
  add2Maps ( "keepaway_log_dir", group( &kaway_log_dir, 9 ) );
  add2Maps ( "keepaway_log_fixed_name", group( &kaway_log_fixed_name, 9 ) );
  add2Maps ( "keepaway_log_fixed", group( &kaway_log_fixed, 9 ), true );
  add2Maps ( "keepaway_log_dated", group( &kaway_log_dated, 9 ), true );

  add2Maps ( "keepaway_start", group( &kaway_start, 9 ) );

};

void
ServerParam::setDefaults()
```

```
{
 /* set default parameter */
 gwidth = GOAL_WIDTH ;
 psize = PLAYER_SIZE ;
 pdecay = PLAYER_DECAY ;
 prand = PLAYER_RAND ;
 pweight = PLAYER_WEIGHT ;
 stamina_max = STAMINA_MAX ;
 stamina_inc = STAMINA_INC_MAX ;
 recover_dthr = RECOVERY_DEC_THR ;
 recover_min = RECOVERY_MIN ;
 recover_dec = RECOVERY_DEC ;
 effort_dthr = EFFORT_DEC_THR ;
 effort_min = EFFORT_MIN ;
 effort_dec = EFFORT_DEC ;
 effort_ithr = EFFORT_INC_THR ;
 effort_inc= EFFORT_INC ;
 inertia_moment = IMPARAM ;
 // pfr 8/14/00: for RC2000 evaluation
 kick_rand = KICK_RAND;
 team_actuator_noise = FALSE;
 prand_factor_l = PRAND_FACTOR_L;
 prand_factor_r = PRAND_FACTOR_R;
 kick_rand_factor_l = KICK_RAND_FACTOR_L;
 kick_rand_factor_r = KICK_RAND_FACTOR_R;

 bsize = BALL_SIZE ;
 bdecay = BALL_DECAY ;
 brand = BALL_RAND ;
 bweight = BALL_WEIGHT ;
 bspeed_max = BALL_SPEED_MAX ;
 pspeed_max = PLAYER_SPEED_MAX ;
 // th 6.3.00
 baccel_max = BALL_ACCEL_MAX ;
 paccel_max = PLAYER_ACCEL_MAX ;
 //
 dprate = DASHPOWERRATE ;
 kprate = KICKPOWERRATE ;
 kmargin = KICKABLE_MARGIN ;
 ctlradius = CONTROL_RADIUS ;
 ckmargin = CORNER_KICK_MARGIN ;
 catch_prob = GOALIE_CATCHABLE_POSSIBILITY ;
 catch_area_l = GOALIE_CATCHABLE_AREA_LENGTH ;
 catch_area_w = GOALIE_CATCHABLE_AREA_WIDTH ;
 goalie_max_moves = GOALIE_MAX_MOVES;
 maxp = MAXPOWER ;
```

```
  minp = MINPOWER ;
  maxm = MAXMOMENT ;
  minm = MINMOMENT ;
  maxnm = MAX_NECK_MOMENT ;
  minnm = MIN_NECK_MOMENT ;
  maxn = MAX_NECK_ANGLE ;
  minn = MIN_NECK_ANGLE ;
  visangle = VisibleAngle ;
  visdist = VisibleDistance ;
  half_time = HALF_TIME ;
  drop_time = DROP_TIME ;
  portnum = DEFAULT_PORT_NUMBER ;
  coach_pnum = COACH_PORT_NUMBER ;
  olcoach_pnum = OLCOACH_PORT_NUMBER ;
  say_cnt_max = DEF_SAY_COACH_CNT_MAX ;
  SayCoachMsgSize = DEF_SAY_COACH_MSG_SIZE ;
  sim_st = SIMULATOR_STEP_INTERVAL_MSEC ;
  send_st = UDP_SEND_STEP_INTERVAL_MSEC ;
  recv_st = UDP_RECV_STEP_INTERVAL_MSEC ;
  sb_step = SENSE_BODY_INTERVAL_MSEC ;
  clang_win_size = 300;
  clang_define_win = 1;
  clang_meta_win = 1;
  clang_advice_win = 1;
  clang_info_win = 1;
  M_clang_del_win = CLANG_DEL_WIN;
  M_clang_rule_win = CLANG_RULE_WIN;
  clang_mess_delay = 50;
  clang_mess_per_cycle = 1;
  CoachMode = FALSE ;
  CwRMode = FALSE ;
  old_hear = FALSE ;
  sv_st = SEND_VISUALINFO_INTERVAL_MSEC ;
//    replay[0] = NULLCHAR ;
  audio_dist = AUDIO_CUT_OFF_DIST ;
  cban_cycle = GOALIE_CATCH_BAN_CYCLE ;
  slow_down_factor = 1;
  verbose = FALSE ;

  dist_qstep = DIST_QSTEP ;
  land_qstep = LAND_QSTEP ;
#ifdef NEW_QSTEP
  dir_qstep  = DIR_QSTEP ;
#endif

  windir = WIND_DIR ;
```

```
  winforce = WIND_FORCE ;
  winrand = WIND_RAND ;
  win_no = FALSE ;
  win_random = FALSE ;
  useoffside = TRUE ;
  offside_area = OFFSIDE_ACTIVE_AREA_SIZE ;
  kickoffoffside = TRUE;
  offside_kick_margin = OFFSIDE_KICK_MARGIN ;

  kaway = FALSE ;
  ka_length = KEEPAWAY_LENGTH ;
  ka_width = KEEPAWAY_WIDTH ;

//pfr:SYNCH
  synch_mode = false;
  synch_offset = 60;
  synch_micro_sleep = 1;

  start_goal_l = 0;
  start_goal_r = 0;

  fullstate_l = FALSE;
  fullstate_r = FALSE;

  /*  // skip command name */
  /*  int tmp_argc = argc-1; */
  /*  argv++ ; argc-- ; */

#ifdef NEW_QSTEP
  bool defined_qstep_l = false;
  bool defined_qstep_r = false;
  bool defined_qstep_l_l = false;
  bool defined_qstep_l_r = false;
  bool defined_qstep_dir_l = false;
  bool defined_qstep_dir_r = false;
#endif

  slowness_on_top_for_left_team = 1.0;
  slowness_on_top_for_right_team = 1.0;

  landmark_file = ServerParam::LANDMARK_FILE;


  send_comms = ServerParam::SEND_COMMS;
  text_logging = ServerParam::TEXT_LOGGING;
  game_logging = ServerParam::GAME_LOGGING;
```

```
game_log_version = ServerParam::GAME_LOG_VERSION;
text_log_dir = ServerParam::TEXT_LOG_DIR;
game_log_dir = ServerParam::GAME_LOG_DIR;
text_log_fixed_name = ServerParam::TEXT_LOG_FIXED_NAME;
game_log_fixed_name = ServerParam::GAME_LOG_FIXED_NAME;
text_log_fixed = ServerParam::TEXT_LOG_FIXED;
game_log_fixed = ServerParam::GAME_LOG_FIXED;
text_log_dated = ServerParam::TEXT_LOG_DATED;
game_log_dated = ServerParam::GAME_LOG_DATED;
log_date_format = ServerParam::LOG_DATE_FORMAT;
log_times = ServerParam::LOG_TIMES;
record_messages = ServerParam::RECORD_MESSAGES;
text_log_compression = ServerParam::TEXT_LOG_COMPRESSION;
game_log_compression = ServerParam::GAME_LOG_COMPRESSION;
M_profile = ServerParam::PROFILE;

kaway_logging = ServerParam::KAWAY_LOGGING;
kaway_log_dir = ServerParam::KAWAY_LOG_DIR;
kaway_log_fixed_name = ServerParam::KAWAY_LOG_FIXED_NAME;
kaway_log_fixed = ServerParam::KAWAY_LOG_FIXED;
kaway_log_dated = ServerParam::KAWAY_LOG_DATED;

kaway_start = ServerParam::KAWAY_START;

M_point_to_ban = ServerParam::POINT_TO_BAN;
M_point_to_duration = ServerParam::POINT_TO_DURATION;
M_say_msg_size = ServerParam::SAY_MSG_SIZE ;
M_hear_max = ServerParam::HEAR_MAX ;
M_hear_inc = ServerParam::HEAR_INC ;
M_hear_decay = ServerParam::HEAR_DECAY ;

M_tackle_dist = TACKLE_DIST;
M_tackle_back_dist = TACKLE_BACK_DIST;
M_tackle_width = TACKLE_WIDTH;
M_tackle_exponent = TACKLE_EXPONENT;
M_tackle_cycles = TACKLE_CYCLES;
M_tackle_power_rate = TACKLE_POWER_RATE;

M_freeform_wait_period = FREEFORM_WAIT_PERIOD;
M_freeform_send_period = FREEFORM_SEND_PERIOD;

M_free_kick_faults = FREE_KICK_FAULTS;
M_back_passes = BACK_PASSES;

M_proper_goal_kicks = PROPER_GOAL_KICKS;
M_stopped_ball_vel = STOPPED_BALL_VEL;
```

```
  M_max_goal_kicks = MAX_GOAL_KICKS;

  M_auto_mode = S_AUTO_MODE;
  M_kick_off_wait = S_KICK_OFF_WAIT;
  M_connect_wait = S_CONNECT_WAIT;
  M_game_over_wait = S_GAME_OVER_WAIT;

  M_team_l_start = S_TEAM_L_START;
  M_team_r_start = S_TEAM_R_START;
}

void ServerParam::getOptions(const int& argc, const char * const *argv )
{
  /* first, search option '-sfile' */
  bool file_specified = false;

  for(int i = 1 ; i < argc ; i++)
    {
      if ( strcmp( argv [ i ], "-sfile" ) == 0)
        {
          file_specified = true;
          i++;
          readParams ( argv [ i ] );
        }
    }

  if (!file_specified)
    readParams ( ServerParam::SERVER_CONF );


  readOptions ( argc, argv );

#ifdef NEW_QSTEP
  if ( !defined_qstep_l )
    dist_qstep_l = dist_qstep ;
  if ( !defined_qstep_r )
    dist_qstep_r = dist_qstep ;
  if ( !defined_qstep_l_l )
    land_qstep_l = land_qstep ;
  if ( !defined_qstep_l_r )
    land_qstep_r = land_qstep ;
  if ( !defined_qstep_dir_l )
    dir_qstep_l = dir_qstep ;
  if ( !defined_qstep_dir_r )
    dir_qstep_r = dir_qstep ;
#endif
```

```
  half_time = half_time * 1000 / sim_st ;
  /*send_st /= 4 ; */
  kickable_area = kmargin + bsize + psize ;
  ctlradius_width = ctlradius - psize ;
  /* apply the slow down factor */
  sim_st  *= slow_down_factor;
  sb_step *= slow_down_factor;
  sv_st   *= slow_down_factor;
  send_st *= slow_down_factor;
  synch_offset *= slow_down_factor; //pfr:SYNCH
  lcm_st = lcm (sim_st, lcm (send_st, lcm (recv_st, lcm (sb_step, sv_st) ) ) );
  lcm_st = lcm (lcm_st, (synch_mode ? synch_offset : 1)); //pfr:SYNCH

  try
  {
     text_log_dir = tildeExpand ( text_log_dir );
     game_log_dir = tildeExpand ( game_log_dir );
     kaway_log_dir = tildeExpand ( kaway_log_dir );
  }
  catch( const std::string& e )
  {
     std::cerr << "Could not expand log directory paths: " << e << std::endl
               << "Please set the USER environment variable and try again"
               << std::endl;
     exit( EXIT_FAILURE );
  }
}

void ServerParam::writeConfig ( std::ostream& o )
{
  o << "# Configurations for soccerserver\n";
  o << "# Lines that start '#' are comment lines.\n";
  o << "\n";
  o << "# width of goals\n";
  o << "goal_width: " << gwidth << std::endl;
  o << "\n";
  o << "# size, decay, random parameter, weight, maximum speed of players\n";
  o << "player_size: " << psize << std::endl;
  o << "player_decay: " << pdecay << std::endl;
  o << "player_rand: " << prand << std::endl;
  o << "player_weight: " << pweight << std::endl;
  o << "player_speed_max: " << pspeed_max << std::endl;
  o << "player_accel_max: " << paccel_max << std::endl;
  o << "\n";
  o << "# maximum and recovery step of player's stamina\n";
```

```
o << "stamina_max: " << stamina_max << std::endl;
o << "stamina_inc_max        : " << stamina_inc << std::endl;
o << "\n";
o << "# decriment threshold ,decriment step and minimum of player's recovery\n";
o << "recover_dec_thr        : " << recover_dthr << std::endl;
o << "recover_dec: " << recover_dec << std::endl;
o << "recover_min: " << recover_min << std::endl;
o << "\n";
o << "# decriment threshold, decriment step, incriment threshold, incriment step\n";
o << "# and minimum of player's effort\n";
o << "effort_dec_thr: " << effort_dthr << std::endl;
o << "effort_dec: " << effort_dec << std::endl;
o << "effort_inc_thr: " << effort_ithr << std::endl;
o << "effort_inc: " << effort_inc << std::endl;
o << "effort_min: " << effort_min << std::endl;
o << "\n";
o << "# maximum, recovery step and decay of player's hear capacity\n";
o << "hear_max: " << M_hear_max << std::endl;
o << "hear_inc: " << M_hear_inc << std::endl;
o << "hear_decay: " << M_hear_decay << std::endl;
o << "\n";
o << "# inertia moment of player\n";
o << "inertia_moment: " << inertia_moment << std::endl;
o << "\n";
o << "# interval of sense_body\n";
o << "sense_body_step: " << sb_step << std::endl;
o << "\n";
o << "# goalie catchable area length, width\n";
o << "catchable_area_l        : " << catch_area_l << std::endl;
o << "catchable_area_w        : " << catch_area_w << std::endl;
o << "\n";
o << "# goalie catchbale probability\n";
o << "catch_probability       : " << catch_prob << std::endl;
o << "\n";
o << "# goalie catch ban cycle\n";
o << "catch_ban_cycle         : " << cban_cycle << std::endl;
o << "\n";
o << "# goalie max moves after catch\n";
o << "goalie_max_moves        : " << goalie_max_moves << std::endl;
o << "\n";
o << "# size, decay, random parameter, weight and maximum speed of a ball\n";
o << "ball_size: " << bsize << std::endl;
o << "ball_decay: " << bdecay << std::endl;
o << "ball_rand: " << brand << std::endl;
o << "ball_weight: " << bweight << std::endl;
o << "ball_speed_max: " << bspeed_max << std::endl;
```

```
o << "ball_accel_max: " << baccel_max << std::endl;
o << "\n";
o << "# force, direction and random parameter of wind\n";
o << "wind_force: " << winforce << std::endl;
o << "wind_dir: " << windir << std::endl;
o << "wind_rand: " << winrand << std::endl;
o << "\n";
o << "# kickable margin. kickable_area = kickable_margin + bsize + psize\n";
o << "kickable_margin        : " << kmargin << std::endl;
o << "\n";
o << "# corner kick margin\n";
o << "ckick_margin: " << ckmargin << std::endl;
o << "\n";
o << "# magnification of power in dash, kick \n";
o << "dash_power_rate        : " << dprate << std::endl;
o << "kick_power_rate        : " << kprate << std::endl;
o << "\n";
o << "# kick randomeness\n";
o << "kick_rand: " << kick_rand << std::endl;
o << "\n";
o << "# angle of view corn [unit: degree]\n";
o << "visible_angle        : " << visangle << std::endl;
o << "\n";
o << "# audio cut off distance\n";
o << "audio_cut_dist        : " << audio_dist << std::endl;
o << "\n";
o << "# quantize step of distance for move_object, landmark\n";
o << "quantize_step: " << dist_qstep << std::endl;
o << "quantize_step_l        : " << land_qstep << std::endl;
o << "\n";
o << "# max and min of power in dash and kick\n";
o << "maxpower: " << maxp << std::endl;
o << "minpower: " << minp << std::endl;
o << "\n";
o << "# max and min of power in turn\n";
o << "maxmoment: " << maxm << std::endl;
o << "minmoment: " << minm << std::endl;
o << "\n";
o << "# max and min of power in turn_neck\n";
o << "maxneckmoment: " << maxnm << std::endl;
o << "minneckmoment: " << minnm << std::endl;
o << "\n";
o << "# max and min of neck angle\n";
o << "maxneckang: " << maxn << std::endl;
o << "minneckang: " << minn << std::endl;
o << "\n";
```

```
o << "# Default port number\n";
o << "port: " << portnum << std::endl;
o << "\n";
o << "# Default coach port number\n";
o << "coach_port: " << coach_pnum << std::endl;
o << "\n";
o << "# Default oline coach port number\n";
o << "olcoach_port: " << olcoach_pnum << std::endl;
o << "\n";
o << "# Default upper limit of the number of online coach's message\n";
o << "say_coach_cnt_max: " << say_cnt_max << std::endl;
o << "\n";
o << "# Default upper limit of length of online coach's message\n";
o << "say_coach_msg_size: " << SayCoachMsgSize << std::endl;
o << "\n";
o << "# Parameters for the new coach language\n";
o << "# time window which controls how many messages can be sent\n";
o << "clang_win_size          : " << clang_win_size << std::endl;
o << "# number of messages per window\n";
o << "clang_define_win        : " << clang_define_win << std::endl;
o << "clang_meta_win          : " << clang_meta_win << std::endl;
o << "clang_advice_win        : " << clang_advice_win << std::endl;
o << "clang_info_win          : " << clang_info_win << std::endl;
o << "clang_del_win           : " << M_clang_del_win << std::endl;
o << "clang_rule_win          : " << M_clang_rule_win << std::endl;
o << "# delay between receipt of message and send to players\n";
o << "clang_mess_delay        : " << clang_mess_delay << std::endl;
o << "# maximum number of coach messages sent per cycle\n";
o << "clang_mess_per_cycle    : " << clang_mess_per_cycle << std::endl;
o << "# How long after playon before coaches can send freeform\n";
o << "freeform_wait_period    : " << M_freeform_wait_period << std::endl;
o << "# How long coach can send freeform for after freeform_wait_period\n";
o << "freeform_send_period    : " << M_freeform_send_period << std::endl;
o << "\n";
o << "# Default interval of online coach's look\n";
o << "send_vi_step        : " << sv_st << std::endl;
o << "\n";
o << "# time step of simulation [unit:msec]\n";
o << "simulator_step        : " << sim_st << std::endl;
o << "\n";
o << "# time step of visual information [unit:msec]\n";
o << "send_step: " << send_st << std::endl;
o << "\n";
o << "# time step of acception of command [unit: msec]\n";
o << "recv_step: " << recv_st << std::endl;
o << "\n";
```

```
o << "# length of half of game [unit:sec]\n";
o << "# (if negative, a game does not stop automatically)\n";
o << "half_time: " << half_time << std::endl;
o << "\n";
o << "# number of cycles to wait until dropping the ball automatically\n";
o << "# for free kicks, corner kicks and so on\n";
o << "# (0 means do not drop automatically)\n";
o << "drop_ball_time        : " << drop_time << std::endl;
o << "\n";
o << "# string size of say message [unit:byte]\n";
o << "say_msg_size        : " << M_say_msg_size << std::endl;
o << "\n";
o << "# flag for using off side rule. [on/off]\n";
o << "use_offside: " << ( useoffside ? "on" : "off" ) << std::endl;
o << "\n";
o << "# offside active area size\n";
o << "offside_active_area_size: " << offside_area << std::endl;
o << "\n";
o << "# forbid kick off offside. [on/off] \n";
o << "forbid_kick_off_offside: " << ( kickoffoffside ? "on" : "off" ) << std::endl;
o << "\n";
o << "# flag for verbose mode. [on/off]\n";
o << "verbose: " << ( verbose ? "on" : "off" ) << std::endl;
o << "\n";
o << "# offside kick margin. \n";
o << "offside_kick_margin: " << offside_kick_margin << std::endl;
o << "\n";
o << "\n";
o << "#coach_w_referee\n";
o << "\n";
o << "# how much to slow a player down when it's on the top half of the field\n";
o << "#\n";
o << "# 1.0 means no change\n";
o << "# 2.0 means twice as slow, 10.0 means ten times as slow, etc\n";
o << "# 0.5 means twice as fast\n";
o << "# 0.0 and less is invalid.  This is not error checked\n";
o << "slowness_on_top_for_left_team  : "
  << slowness_on_top_for_left_team << std::endl;
o << "slowness_on_top_for_right_team : "
  << slowness_on_top_for_right_team << std::endl;
o << "\n";
o << "synch_mode              : " << ( synch_mode ? "on" : "off" ) << std::endl;
o << "#synch_micro_sleep      : " << synch_micro_sleep << std::endl;
o << "\n";
o << "# Send client commands to monitors. [on/off]\n";
o << "send_comms: " << ( send_comms ? "on" : "off" ) << std::endl;
```

```
o << "\n";
o << "# LOGGING\n";
o << "# game logging. [on/off]\n";
o << "text_logging: " << ( text_logging ? "on" : "off" ) << std::endl;
o << "game_logging: " << ( game_logging ? "on" : "off" ) << std::endl;
o << "# game_log version.\n";
o << "game_log_version        : " << game_log_version << std::endl;
o << "# Output directories\n";
o << "text_log_dir            : " << text_log_dir << std::endl;
o << "game_log_dir            : " << game_log_dir << std::endl;
o << "# Fixed output names\n";
o << "text_log_fixed_name     : " << text_log_fixed_name << std::endl;
o << "game_log_fixed_name     : " << game_log_fixed_name << std::endl;
o << "# Use fixed output names\n";
o << "text_log_fixed          : " << ( text_log_fixed ? "on\n" : "off\n" );
o << "game_log_fixed          : " << ( game_log_fixed ? "on\n" : "off\n" );
o << "# Add date stamp to output names\n";
o << "text_log_dated          : " << ( text_log_dated ? "on\n" : "off\n" );
o << "game_log_dated          : " << ( game_log_dated ? "on\n" : "off\n" );
o << "log_date_format         : " << log_date_format << std::endl;
o << "# Write ms between cycles in text log. [on/off]\n";
o << "log_times: " << ( log_times ? "on\n" : "off\n" );
o << "# Record messages in game log\n";
o << "record_messages         : " << ( record_messages ? "on\n" : "off\n" );
o << "# Log file compression\n";
o << "text_log_compression    : " << text_log_compression << std::endl;
o << "game_log_compression    : " << game_log_compression << std::endl;
o << "# Write ms taken for each part of a cycle in text log. [on/off]\n";
o << "profile                 : " << ( M_profile ? "on\n" : "off\n" );
o << "\n";
o << "# Point to parameters\n";
o << "point_to_ban            : " << M_point_to_ban << std::endl;
o << "point_to_duration       : " << M_point_to_duration << std::endl;
o << "\n";
o << "# Tackle parameters\n";
o << "tackle_dist             : " << M_tackle_dist << std::endl;
o << "tackle_back_dist        : " << M_tackle_back_dist << std::endl;
o << "tackle_width            : " << M_tackle_width << std::endl;
o << "tackle_exponent         : " << M_tackle_exponent << std::endl;
o << "tackle_cycles           : " << M_tackle_cycles << std::endl;
o << "tackle_power_rate       : " << M_tackle_power_rate << std::endl;
o << "\n";
o << "# Enable free kick fault and back pass detection\n";
o << "free_kick_faults        : "
  << ( M_free_kick_faults ? "on\n" : "off\n" );
o << "back_passes             : "
```

```
  << ( M_back_passes ? "on\n" : "off\n" );
o << "\n";
o << "# Force proper goal kicks\n";
o << "proper_goal_kicks       : "
  << ( M_proper_goal_kicks ? "on\n" : "off\n" );
o << "stopped_ball_vel        : " << M_stopped_ball_vel << std::endl;
o << "max_goal_kicks          : " << M_max_goal_kicks << std::endl;
o << "\n";
o << "# Enable automatic kick off of games and exit when game is over\n";
o << "auto_mode               : " << ( M_auto_mode ? "on\n" : "off\n" );
o << "# Kick off wait is the number of cycles to wait after all the\n";
o << "# players are connected before the game or half time is kicked off\n";
o << "# -1 indicates wait forever\n";
o << "kick_off_wait           : " << M_kick_off_wait << std::endl;
o << "# Connect wait is the number of cycles to wait for all the players\n";
o << "# to connect before kicking off\n";
o << "# -1 indicates wait forever\n";
o << "connect_wait            : " << M_connect_wait << std::endl;
o << "# Game over wait is the number of cycles to wait after the end of\n";
o << "# game before exiting\n";
o << "# -1 indicates wait forever\n";
o << "game_over_wait          : " << M_game_over_wait << std::endl;
o << "\n";
o << "# Use the following commands to automatically start the teams\n";
o << "team_l_start            : \"" << M_team_l_start << "\"" << std::endl;
o << "team_r_start            : \"" << M_team_r_start << "\"" << std::endl;
o << "\n";
o << "# Enable fullstate information\n";
o << "fullstate_l             : " << ( fullstate_l ? "on\n" : "off\n" );
o << "fullstate_r             : " << ( fullstate_r ? "on\n" : "off\n" );
o << "\n";
o << "# Enable keepaway mode\n";
o << "keepaway                : " << ( kaway ? "on\n" : "off\n" );
o << "# Keepaway region size\n";
o << "keepaway_length         : " << ka_length << std::endl;
o << "keepaway_width          : " << ka_width << std::endl;
o << "# Automatically start Keepaway match after delay (seconds)\n";
o << "keepaway_start          : " << kaway_start << std::endl;
o << "# Keepaway logging. [on/off]\n";
o << "keepaway_logging        : " << ( kaway_logging ? "on" : "off" ) << std::endl;
o << "# Output directory\n";
o << "keepaway_log_dir        : " << kaway_log_dir << std::endl;
o << "# Fixed output name\n";
o << "keepaway_log_fixed_name  : " << kaway_log_fixed_name << std::endl;
o << "# Use fixed output name\n";
o << "keepaway_log_fixed      : " << ( kaway_log_fixed ? "on\n" : "off\n" );
```

```
  o << "# Add date stamp to output name\n";
  o << "keepaway_log_dated       : " << ( kaway_log_dated ? "on\n" : "off\n" );
  o << "\n";
}



server_params_t ServerParam::convertToStruct ()
{
  server_params_t tmp;

  tmp.gwidth = htonl( (long)(SHOWINFO_SCALE2 * gwidth) );
  tmp.inertia_moment = htonl( (long)(SHOWINFO_SCALE2 * inertia_moment) );
  tmp.psize = htonl( (long)(SHOWINFO_SCALE2 * psize) );
  tmp.pdecay = htonl( (long)(SHOWINFO_SCALE2 * pdecay) );
  tmp.prand = htonl( (long)(SHOWINFO_SCALE2 * prand) );
  tmp.pweight = htonl( (long)(SHOWINFO_SCALE2 * pweight) );
  tmp.pspeed_max = htonl( (long)(SHOWINFO_SCALE2 * pspeed_max) );
  tmp.paccel_max = htonl( (long)(SHOWINFO_SCALE2 * paccel_max) );
  tmp.stamina_max = htonl( (long)(SHOWINFO_SCALE2 * stamina_max) );
  tmp.stamina_inc = htonl( (long)(SHOWINFO_SCALE2 * stamina_inc) );
  tmp.recover_init = htonl( (long)(SHOWINFO_SCALE2 * recover_init) );
  tmp.recover_dthr = htonl( (long)(SHOWINFO_SCALE2 * recover_dthr) );
  tmp.recover_min = htonl( (long)(SHOWINFO_SCALE2 * recover_min) );
  tmp.recover_dec = htonl( (long)(SHOWINFO_SCALE2 * recover_dec) );
  tmp.effort_init = htonl( (long)(SHOWINFO_SCALE2 * effort_init) );
  tmp.effort_dthr = htonl( (long)(SHOWINFO_SCALE2 * effort_dthr) );
  tmp.effort_min = htonl( (long)(SHOWINFO_SCALE2 * effort_min) );
  tmp.effort_dec = htonl( (long)(SHOWINFO_SCALE2 * effort_dec) );
  tmp.effort_ithr = htonl( (long)(SHOWINFO_SCALE2 * effort_ithr) );
  tmp.effort_inc = htonl( (long)(SHOWINFO_SCALE2 * effort_inc) );
  tmp.kick_rand = htonl( (long)(SHOWINFO_SCALE2 * kick_rand) );
  tmp.team_actuator_noise = htons( (short) team_actuator_noise );
  tmp.prand_factor_l = htonl( (long)(SHOWINFO_SCALE2 * prand_factor_l ) );
  tmp.prand_factor_r = htonl( (long)(SHOWINFO_SCALE2 * prand_factor_r ) );
  tmp.kick_rand_factor_l = htonl( (long)(SHOWINFO_SCALE2 * kick_rand_factor_l ) );
  tmp.kick_rand_factor_r = htonl( (long)(SHOWINFO_SCALE2 * kick_rand_factor_r ) );
  tmp.bsize = htonl( (long)(SHOWINFO_SCALE2 * bsize) );
  tmp.bdecay = htonl( (long)(SHOWINFO_SCALE2 * bdecay) );
  tmp.brand = htonl( (long)(SHOWINFO_SCALE2 * brand) );
  tmp.bweight = htonl( (long)(SHOWINFO_SCALE2 * bweight) );
  tmp.bspeed_max = htonl( (long)(SHOWINFO_SCALE2 * bspeed_max) );
  tmp.baccel_max = htonl( (long)(SHOWINFO_SCALE2 * baccel_max ) );
  tmp.dprate = htonl( (long)(SHOWINFO_SCALE2 * dprate) );
  tmp.kprate = htonl( (long)(SHOWINFO_SCALE2 * kprate) );
  tmp.kmargin = htonl( (long)(SHOWINFO_SCALE2 * kmargin) );
```

```
tmp.ctlradius = htonl( (long)(SHOWINFO_SCALE2 * ctlradius) );
tmp.ctlradius_width = htonl( (long)(SHOWINFO_SCALE2 * ctlradius_width) );
tmp.maxp = htonl( (long)(SHOWINFO_SCALE2 * maxp) );
tmp.minp = htonl( (long)(SHOWINFO_SCALE2 * minp) );
tmp.maxm = htonl( (long)(SHOWINFO_SCALE2 * maxm) );
tmp.minm = htonl( (long)(SHOWINFO_SCALE2 * minm) );
tmp.maxnm = htonl( (long)(SHOWINFO_SCALE2 * maxnm) );
tmp.minnm = htonl( (long)(SHOWINFO_SCALE2 * minnm) );
tmp.maxn = htonl( (long)(SHOWINFO_SCALE2 * maxn) );
tmp.minn = htonl( (long)(SHOWINFO_SCALE2 * minn) );
tmp.visangle = htonl( (long)(SHOWINFO_SCALE2 * visangle) );
tmp.visdist = htonl( (long)(SHOWINFO_SCALE2 * visdist) );
tmp.windir = htonl( (long)(SHOWINFO_SCALE2 * windir) );
tmp.winforce = htonl( (long)(SHOWINFO_SCALE2 * winforce) );
tmp.winang = htonl( (long)(SHOWINFO_SCALE2 * winang) );
tmp.winrand = htonl( (long)(SHOWINFO_SCALE2 * winrand) );
tmp.kickable_area = htonl( (long)(SHOWINFO_SCALE2 * kickable_area) );
tmp.catch_area_l = htonl( (long)(SHOWINFO_SCALE2 * catch_area_l) );
tmp.catch_area_w = htonl( (long)(SHOWINFO_SCALE2 * catch_area_w) );
tmp.catch_prob = htonl( (long)(SHOWINFO_SCALE2 * catch_prob) );
tmp.goalie_max_moves = htons( (short) goalie_max_moves );
tmp.ckmargin = htonl( (long)(SHOWINFO_SCALE2 * ckmargin) );
tmp.offside_area = htonl( (long)(SHOWINFO_SCALE2 * offside_area) );
tmp.win_no = htons( (short) win_no);
tmp.win_random = htons( (short) win_random) ;
tmp.say_cnt_max = htons( (short) say_cnt_max) ;
tmp.SayCoachMsgSize = htons( (short) SayCoachMsgSize) ;
tmp.clang_win_size = htons( (short) clang_win_size );
tmp.clang_define_win = htons( (short) clang_define_win );
tmp.clang_meta_win = htons( (short) clang_meta_win );
tmp.clang_advice_win = htons( (short) clang_advice_win );
tmp.clang_info_win = htons( (short) clang_info_win );
tmp.clang_mess_delay = htons( (short) clang_mess_delay );
tmp.clang_mess_per_cycle = htons( (short) clang_mess_per_cycle );
tmp.half_time = htons( (short) half_time) ;
tmp.sim_st = htons( (short) sim_st) ;
tmp.send_st = htons( (short) send_st) ;
tmp.recv_st = htons( (short) recv_st) ;
tmp.sb_step = htons( (short) sb_step) ;
tmp.lcm_st = htons( (short) lcm_st) ;
tmp.M_say_msg_size = htons( (short) M_say_msg_size) ;
tmp.M_hear_max = htons( (short) M_hear_max) ;
tmp.M_hear_inc = htons( (short) M_hear_inc) ;
tmp.M_hear_decay = htons( (short) M_hear_decay) ;
tmp.cban_cycle = htons( (short) cban_cycle) ;
tmp.slow_down_factor = htons( (short) slow_down_factor) ;
```

```
  tmp.useoffside = htons( (short) useoffside) ;
  tmp.kickoffoffside = htons( (short) kickoffoffside) ;
  tmp.offside_kick_margin = htonl( (long)(SHOWINFO_SCALE2 * offside_kick_margin) );
  tmp.audio_dist = htonl( (long)(SHOWINFO_SCALE2 * audio_dist) );
  tmp.dist_qstep = htonl( (long)(SHOWINFO_SCALE2 * dist_qstep) );
  tmp.land_qstep = htonl( (long)(SHOWINFO_SCALE2 * land_qstep) );
#ifdef NEW_QSTEP
  tmp.dir_qstep = htonl( (long)(SHOWINFO_SCALE2 * dir_qstep) );
  tmp.dist_qstep_l = htonl( (long)(SHOWINFO_SCALE2 * dist_qstep_l) );
  tmp.dist_qstep_r = htonl( (long)(SHOWINFO_SCALE2 * dist_qstep_r) );
  tmp.land_qstep_l = htonl( (long)(SHOWINFO_SCALE2 * land_qstep_l) );
  tmp.land_qstep_r = htonl( (long)(SHOWINFO_SCALE2 * land_qstep_r) );
  tmp.dir_qstep_l = htonl( (long)(SHOWINFO_SCALE2 * dir_qstep_l) );
  tmp.dir_qstep_r = htonl( (long)(SHOWINFO_SCALE2 * dir_qstep_r) );
#else
  tmp.dir_qstep = htonl( (long)(SHOWINFO_SCALE2 * -1) );
  tmp.dist_qstep_l = htonl( (long)(SHOWINFO_SCALE2 * -1) );
  tmp.dist_qstep_r = htonl( (long)(SHOWINFO_SCALE2 * -1) );
  tmp.land_qstep_l = htonl( (long)(SHOWINFO_SCALE2 * -1) );
  tmp.land_qstep_r = htonl( (long)(SHOWINFO_SCALE2 * -1) );
  tmp.dir_qstep_l = htonl( (long)(SHOWINFO_SCALE2 * -1) );
  tmp.dir_qstep_r = htonl( (long)(SHOWINFO_SCALE2 * -1) );
#endif
  tmp.CoachMode = htons( (short) CoachMode) ;
  tmp.CwRMode = htons( (short) CwRMode) ;
  tmp.old_hear = htons( (short) old_hear) ;
  tmp.sv_st = htons( (short) sv_st) ;

  tmp.synch_mode   = htons( (short)synch_mode);//pfr:SYNCH
  tmp.synch_offset = htons( (short)synch_offset);//pfr:SYNCH
  tmp.synch_micro_sleep = htons( (short)synch_micro_sleep);//pfr:SYNCH

  tmp.start_goal_l = htons( (short)start_goal_l );
  tmp.start_goal_r = htons( (short)start_goal_r );

  tmp.fullstate_l = htons( (short)fullstate_l );
  tmp.fullstate_r = htons( (short)fullstate_r );

  tmp.drop_time = htons( (short) drop_time) ;

  tmp.slowness_on_top_for_left_team = htonl( (long)(SHOWINFO_SCALE2 *
slowness_on_top_for_left_team) );
  tmp.slowness_on_top_for_right_team = htonl( (long)(SHOWINFO_SCALE2 *
slowness_on_top_for_left_team) );

  tmp.ka_length = htonl( (long)(SHOWINFO_SCALE2 * ka_length) );
```

```
  tmp.ka_width = htonl( (long)(SHOWINFO_SCALE2 * ka_width) );
  //tmp.kaway = htons( (short) kaway);

  tmp.point_to_ban =  htons( (short)M_point_to_ban );
  tmp.point_to_duration =  htons( (short)M_point_to_duration );
  return tmp;
}


std::ostream& toStr ( std::ostream& o, const ServerParamSensor_v7::data_t& data )
{
  return o << "(server_param "
           << data.M_gwidth << " "
           << data.M_inertia_moment << " "
           << data.M_psize << " "
           << data.M_pdecay << " "
           << data.M_prand << " "
           << data.M_pweight << " "
           << data.M_pspeed_max << " "
           << data.M_paccel_max << " "
           << data.M_stamina_max << " "
           << data.M_stamina_inc << " "
           << data.M_recover_init << " "
           << data.M_recover_dthr << " "
           << data.M_recover_min << " "
           << data.M_recover_dec << " "
           << data.M_effort_init << " "
           << data.M_effort_dthr << " "
           << data.M_effort_min << " "
           << data.M_effort_dec << " "
           << data.M_effort_ithr << " "
           << data.M_effort_inc << " "
           << data.M_kick_rand << " "
           << data.M_team_actuator_noise << " "
           << data.M_prand_factor_l << " "
           << data.M_prand_factor_r << " "
           << data.M_kick_rand_factor_l << " "
           << data.M_kick_rand_factor_r << " "
           << data.M_bsize << " "
           << data.M_bdecay << " "
           << data.M_brand << " "
           << data.M_bweight << " "
           << data.M_bspeed_max << " "
           << data.M_baccel_max << " "
           << data.M_dprate << " "
           << data.M_kprate << " "
```

```
<< data.M_kmargin << " "
<< data.M_ctlradius << " "
<< data.M_ctlradius_width << " "
<< data.M_maxp << " "
<< data.M_minp << " "
<< data.M_maxm << " "
<< data.M_minm << " "
<< data.M_maxnm << " "
<< data.M_minnm << " "
<< data.M_maxn << " "
<< data.M_minn << " "
<< data.M_visangle << " "
<< data.M_visdist << " "
<< data.M_windir << " "
<< data.M_winforce << " "
<< data.M_winang << " "
<< data.M_winrand << " "
<< data.M_kickable_area << " "
<< data.M_catch_area_l << " "
<< data.M_catch_area_w << " "
<< data.M_catch_prob << " "
<< data.M_goalie_max_moves << " "
<< data.M_ckmargin << " "
<< data.M_offside_area << " "
<< data.M_win_no << " "
<< data.M_win_random << " "
<< data.M_say_cnt_max << " "
<< data.M_SayCoachMsgSize << " "
<< data.M_clang_win_size << " "
<< data.M_clang_define_win << " "
<< data.M_clang_meta_win << " "
<< data.M_clang_advice_win << " "
<< data.M_clang_info_win << " "
<< data.M_clang_mess_delay << " "
<< data.M_clang_mess_per_cycle << " "
<< data.M_half_time << " "
<< data.M_sim_st << " "
<< data.M_send_st << " "
<< data.M_recv_st << " "
<< data.M_sb_step << " "
<< data.M_lcm_st << " "
<< data.M_say_msg_size << " "
<< data.M_hear_max << " "
<< data.M_hear_inc << " "
<< data.M_hear_decay << " "
<< data.M_cban_cycle << " "
```

```
        << data.M_slow_down_factor << " "
        << data.M_useoffside << " "
        << data.M_kickoffoffside << " "
        << data.M_offside_kick_margin << " "
        << data.M_audio_dist << " "
        << data.M_dist_qstep << " "
        << data.M_land_qstep << " "
        << data.M_dir_qstep << " "
        << data.M_dist_qstep_l << " "
        << data.M_dist_qstep_r << " "
        << data.M_land_qstep_l << " "
        << data.M_land_qstep_r << " "
        << data.M_dir_qstep_l << " "
        << data.M_dir_qstep_r << " "
        << data.M_CoachMode << " "
        << data.M_CwRMode << " "
        << data.M_old_hear << " "
        << data.M_sv_st << " "
        << data.M_start_goal_l << " "
        << data.M_start_goal_r << " "
        << data.M_fullstate_l << " "
        << data.M_fullstate_r << " "
        << data.M_drop_time << ")";
}



std::ostream& toStr ( std::ostream& o, const ServerParamSensor_v8::data_t& data )
{
  o << "(server_param ";
  std::for_each( data.int_map.begin(), data.int_map.end(), ServerParam::Printer( o, 8 ) );
  std::for_each( data.str_map.begin(), data.str_map.end(), ServerParam::QuotedPrinter( o, 8 ) );
  std::for_each( data.bool_map.begin(), data.bool_map.end(), ServerParam::Printer( o, 8 ) );
  std::for_each( data.onoff_map.begin(), data.onoff_map.end(), ServerParam::Printer( o, 8 ) );
  std::for_each( data.double_map.begin(), data.double_map.end(), ServerParam::Printer( o, 8 ) );
  o << ")";

  return o;
//    return o << "(server_param "
//             << "(goal_width " << data.M_gwidth
//             << ") (player_size "  << data.M_psize
//             << ") (player_decay " << data.M_pdecay
//             << ") (player_rand " << data.M_prand
//             << ") (player_weight " << data.M_pweight
//             << ") (player_speed_max " << data.M_pspeed_max
//             << ") (player_accel_max " << data.M_paccel_max
```

```
//            << ") (stamina_max " << data.M_stamina_max
//            << ") (stamina_inc_max " << data.M_stamina_inc
//            << ") (recover_init " << data.M_recover_init
//            << ") (recover_dec_thr " << data.M_recover_dthr
//            << ") (recover_min " << data.M_recover_min
//            << ") (recover_dec " << data.M_recover_dec
//            << ") (effort_init " << data.M_effort_init
//            << ") (effort_dec_thr " << data.M_effort_dthr
//            << ") (effort_min " << data.M_effort_min
//            << ") (effort_dec " << data.M_effort_dec
//            << ") (effort_inc_thr " << data.M_effort_ithr
//            << ") (effort_inc " << data.M_effort_inc
//            << ") (kick_rand " << data.M_kick_rand
//            << ") (team_actuator_noise " << data.M_team_actuator_noise
//            << ") (prand_factor_l " << data.M_prand_factor_l
//            << ") (prand_factor_r " << data.M_prand_factor_r
//            << ") (kick_rand_factor_l " << data.M_kick_rand_factor_l
//            << ") (kick_rand_factor_r " << data.M_kick_rand_factor_r
//            << ") (ball_size " << data.M_bsize
//            << ") (ball_decay " << data.M_bdecay
//            << ") (ball_rand " << data.M_brand
//            << ") (ball_weight " << data.M_bweight
//            << ") (ball_speed_max " << data.M_bspeed_max
//            << ") (ball_accel_max " << data.M_baccel_max
//            << ") (dash_power_rate " << data.M_dprate
//            << ") (kick_power_rate " << data.M_kprate
//            << ") (kickable_margin " << data.M_kmargin
//            << ") (control_radius " << data.M_ctlradius
//            << ") (control_radius_width " << data.M_ctlradius_width
//            << ") (catch_probability " << data.M_catch_prob
//            << ") (catchable_area_l " << data.M_catch_area_l
//            << ") (catchable_area_w " << data.M_catch_area_w
//            << ") (goalie_max_moves " << data.M_goalie_max_moves
//            << ") (maxpower " << data.M_maxp
//            << ") (minpower " << data.M_minp
//            << ") (maxmoment " << data.M_maxm
//            << ") (minmoment " << data.M_minm
//            << ") (maxneckmoment " << data.M_maxnm
//            << ") (minneckmoment " << data.M_minnm
//            << ") (maxneckang " << data.M_maxn
//            << ") (minneckang " << data.M_minn
//            << ") (visible_angle " << data.M_visangle
//            << ") (visible_distance " << data.M_visdist
//            << ") (audio_cut_dist " << data.M_audio_dist
//            << ") (quantize_step " << data.M_dist_qstep
//            << ") (quantize_step_l " << data.M_land_qstep
```

```
//            << ") (quantize_step_dir " << data.M_dir_qstep
//            << ") (quantize_step_dist_team_l " << data.M_dist_qstep_l
//            << ") (quantize_step_dist_team_r " << data.M_dist_qstep_r
//            << ") (quantize_step_dist_l_team_l " << data.M_land_qstep_l
//            << ") (quantize_step_dist_l_team_r " << data.M_land_qstep_r
//            << ") (quantize_step_dir_team_l " << data.M_dir_qstep_l
//            << ") (quantize_step_dir_team_r " << data.M_dir_qstep_r
//            << ") (ckick_margin " << data.M_ckmargin
//            << ") (wind_dir " << data.M_windir
//            << ") (wind_force " << data.M_winforce
//            << ") (wind_ang " << data.M_winang
//            << ") (wind_rand " << data.M_winrand
//            << ") (kickable_area " << data.M_kickable_area
//            << ") (inertia_moment " << data.M_inertia_moment
//            << ") (wind_none " << data.M_win_no
//            << ") (wind_random " << data.M_win_random
//            << ") (half_time " << data.M_half_time
//            << ") (drop_ball_time " << data.M_drop_time
//            << ") (port " << data.M_portnum
//            << ") (coach_port " << data.M_coach_pnum
//            << ") (olcoach_port " << data.M_olcoach_pnum
//            << ") (say_coach_cnt_max " << data.M_say_cnt_max
//            << ") (say_coach_msg_size " << data.M_SayCoachMsgSize
//            << ") (simulator_step " << data.M_sim_st
//            << ") (send_step " << data.M_send_st
//            << ") (recv_step " << data.M_recv_st
//            << ") (sense_body_step " << data.M_sb_step
//            << ") (lcm_step " << data.M_lcm_st
//            << ") (say_msg_size " << data.M_say_msg_size
//            << ") (clang_win_size " << data.M_clang_win_size
//            << ") (clang_define_win " << data.M_clang_define_win
//            << ") (clang_meta_win " << data.M_clang_meta_win
//            << ") (clang_advice_win " << data.M_clang_advice_win
//            << ") (clang_info_win " << data.M_clang_info_win
//            << ") (clang_mess_delay " << data.M_clang_mess_delay
//            << ") (clang_mess_per_cycle " << data.M_clang_mess_per_cycle
//            << ") (hear_max " << data.M_hear_max
//            << ") (hear_inc " << data.M_hear_inc
//            << ") (hear_decay " << data.M_hear_decay
//            << ") (catch_ban_cycle " << data.M_cban_cycle
//            << ") (coach " << data.M_CoachMode
//            << ") (coach_w_referee " << data.M_CwRMode
//            << ") (old_coach_hear " << data.M_old_hear
//            << ") (send_vi_step " << data.M_sv_st
//            << ") (use_offside " << data.M_useoffside
//            << ") (offside_active_area_size " << data.M_offside_area
```

```
//            << ") (forbid_kick_off_offside " << data.M_kickoffoffside
//            << ") (verbose " << data.M_verbose
//            << ") (replay " << ( data.M_replay.size() == 0 ? "0" : data.M_replay )
//            << ") (offside_kick_margin " << data.M_offside_kick_margin
//            << ") (slow_down_factor " << data.M_slow_down_factor
//            << ") (synch_mode " << data.M_synch_mode
//            << ") (synch_offset " << data.M_synch_offset
//            << ") (synch_micro_sleep " << data.M_synch_micro_sleep
//            << ") (start_goal_l " << data.M_start_goal_l
//            << ") (start_goal_r " << data.M_start_goal_r
//            << ") (fullstate_l " << data.M_fullstate_l
//            << ") (fullstate_r " << data.M_fullstate_r
//            << ") (slowness_on_top_for_left_team "
//            << data.M_slowness_on_top_for_left_team
//            << ") (slowness_on_top_for_right_team "
//            << data.M_slowness_on_top_for_right_team
//            << ") (send_comms " << data.M_send_comms
//            << ") (text_logging " << data.M_text_logging
//            << ") (game_logging " << data.M_game_logging
//            << ") (game_log_version " << data.M_game_log_version
//            << ") (text_log_dir " << data.M_text_log_dir
//            << ") (game_log_dir " << data.M_game_log_dir
//            << ") (text_log_fixed_name " << data.M_text_log_fixed_name
//            << ") (game_log_fixed_name " << data.M_game_log_fixed_name
//            << ") (text_log_fixed " << data.M_text_log_fixed
//            << ") (game_log_fixed " << data.M_game_log_fixed
//            << ") (text_log_dated " << data.M_text_log_dated
//            << ") (game_log_dated " << data.M_game_log_dated
//            << ") (log_date_format " << data.M_log_date_format
//            << ") (log_times " << data.M_log_times
//            << ") (record_messages " << data.M_record_messages
//            << ") (text_log_compression " << data.M_text_log_compression
//            << ") (game_log_compression " << data.M_game_log_compression
//            << ") (profile " << data.M_profile
//            << ") (point_to_ban " << data.M_point_to_ban
//            << ") (point_to_duration " << data.M_point_to_duration
//            << "))";
}
```

```
srcdir = .
top_srcdir = ..

pkgdatadir = $(datadir)/trilearn_base_sources
pkglibdir = $(libdir)/trilearn_base_sources
pkgincludedir = $(includedir)/trilearn_base_sources
top_builddir = ..

am__cd = CDPATH="$${ZSH_VERSION+.}$(PATH_SEPARATOR)" && cd
INSTALL = /usr/bin/install -c
install_sh_DATA = $(install_sh) -c -m 644
install_sh_PROGRAM = $(install_sh) -c
install_sh_SCRIPT = $(install_sh) -c
INSTALL_HEADER = $(INSTALL_DATA)
transform = $(program_transform_name)
NORMAL_INSTALL = :
PRE_INSTALL = :
POST_INSTALL = :
NORMAL_UNINSTALL = :
PRE_UNINSTALL = :
POST_UNINSTALL = :
host_triplet = i686-pc-linux-gnu
ACLOCAL = ${SHELL} /home/kamran/teams/uva1/missing --run aclocal-1.7
AMDEP_FALSE = #
AMDEP_TRUE =
AMTAR = ${SHELL} /home/kamran/teams/uva1/missing --run tar
AR = ar
AUTOCONF = ${SHELL} /home/kamran/teams/uva1/missing --run autoconf
AUTOHEADER = ${SHELL} /home/kamran/teams/uva1/missing --run autoheader
AUTOMAKE = ${SHELL} /home/kamran/teams/uva1/missing --run automake-1.7
AWK = gawk
CC = gcc
CCDEPMODE = depmode=gcc3
CFLAGS = -g -O2
CPP = gcc -E
CPPFLAGS =
CXX = g++
CXXCPP = g++ -E
CXXDEPMODE = depmode=gcc3
CXXFLAGS = -g -O2
CYGPATH_W = echo
DEFS = -DHAVE_CONFIG_H
DEPDIR = .deps
ECHO = echo
ECHO_C =
ECHO_N = -n
ECHO_T =
EGREP = grep -E
EXEEXT =
F77 = g77
FFLAGS = -g -O2
INSTALL_DATA = ${INSTALL} -m 644
INSTALL_PROGRAM = ${INSTALL}
INSTALL_SCRIPT = ${INSTALL}
INSTALL_STRIP_PROGRAM = ${SHELL} $(install_sh) -c -s
```

```
LDFLAGS =
LIBOBJS =
LIBS = -lpthread
LIBTOOL = $(SHELL) $(top_builddir)/libtool
LN_S = ln -s
LTLIBOBJS =
MAKEINFO = ${SHELL} /home/kamran/teams/uva1/missing --run makeinfo
OBJEXT = o
PACKAGE = trilearn_base_sources
PACKAGE_BUGREPORT = jellekok@science.uva.nl
PACKAGE_NAME = trilearn__base_sources
PACKAGE_STRING = trilearn__base_sources 3.3
PACKAGE_TARNAME = trilearn__base_sources
PACKAGE_VERSION = 3.3
PATH_SEPARATOR = :
RANLIB = ranlib
SET_MAKE =
SHELL = /bin/sh
STRIP = strip
VERSION = 3.3
ac_ct_AR = ar
ac_ct_CC = gcc
ac_ct_CXX = g++
ac_ct_F77 = g77
ac_ct_RANLIB = ranlib
ac_ct_STRIP = strip
am__fastdepCC_FALSE = #
am__fastdepCC_TRUE =
am__fastdepCXX_FALSE = #
am__fastdepCXX_TRUE =
am__include = include
am__leading_dot = .
am__quote =
bindir = ${exec_prefix}/bin
build = i686-pc-linux-gnu
build_alias =
build_cpu = i686
build_os = linux-gnu
build_vendor = pc
datadir = ${prefix}/share
exec_prefix = ${prefix}
host = i686-pc-linux-gnu
host_alias =
host_cpu = i686
host_os = linux-gnu
host_vendor = pc
includedir = ${prefix}/include
infodir = ${prefix}/info
install_sh = /home/kamran/teams/uva1/install-sh
libdir = ${exec_prefix}/lib
libexecdir = ${exec_prefix}/libexec
localstatedir = ${prefix}/var
mandir = ${prefix}/man
oldincludedir = /usr/include
prefix = /usr/local
program_transform_name = s,x,x,
sbindir = ${exec_prefix}/sbin
```

```
sharedstatedir = ${prefix}/com
sysconfdir = ${prefix}/etc
target_alias =

bin_PROGRAMS = trilearn_player trilearn_coach

SOURCES = Connection.cpp \
        SenseHandler.cpp \
        ActHandler.cpp \
        SoccerTypes.cpp \
        Objects.cpp \
        WorldModel.cpp \
        WorldModelHighLevel.cpp \
        WorldModelPredict.cpp \
        WorldModelUpdate.cpp \
        ServerSettings.cpp \
        PlayerSettings.cpp \
        GenericValues.cpp \
        Formations.cpp \
        Geometry.cpp \
        Parse.cpp \
        Logger.cpp \
      Connection.h \
        SenseHandler.h \
        ActHandler.h \
        SoccerTypes.h \
        Objects.h \
        WorldModel.h \
        ServerSettings.h \
        PlayerSettings.h \
        GenericValues.h \
        Formations.h \
        Geometry.h \
        Parse.h \
        Logger.h


trilearn_player_SOURCES = ${SOURCES}    \
        BasicPlayer.cpp \
        Player.cpp \
        PlayerTeams.cpp \
        main.cpp \
        BasicPlayer.h \
        Player.h


trilearn_coach_SOURCES = ${SOURCES} \
        BasicCoach.cpp \
        BasicCoach.h \
        mainCoach.cpp


EXTRA_DIST = formations.conf player.conf
subdir = src
ACLOCAL_M4 = $(top_srcdir)/aclocal.m4
mkinstalldirs = $(SHELL) $(top_srcdir)/mkinstalldirs
CONFIG_HEADER = $(top_builddir)/config.h
```

```
CONFIG_CLEAN_FILES =
bin_PROGRAMS = trilearn_player$(EXEEXT) trilearn_coach$(EXEEXT)
PROGRAMS = $(bin_PROGRAMS)

am__objects_1 = Connection.$(OBJEXT) SenseHandler.$(OBJEXT) \
      ActHandler.$(OBJEXT) SoccerTypes.$(OBJEXT) Objects.$(OBJEXT) \
      WorldModel.$(OBJEXT) WorldModelHighLevel.$(OBJEXT) \
      WorldModelPredict.$(OBJEXT) WorldModelUpdate.$(OBJEXT) \
      ServerSettings.$(OBJEXT) PlayerSettings.$(OBJEXT) \
      GenericValues.$(OBJEXT) Formations.$(OBJEXT) Geometry.$(OBJEXT) \
      Parse.$(OBJEXT) Logger.$(OBJEXT)
am_trilearn_coach_OBJECTS = $(am__objects_1) BasicCoach.$(OBJEXT) \
      mainCoach.$(OBJEXT)
trilearn_coach_OBJECTS = $(am_trilearn_coach_OBJECTS)
trilearn_coach_LDADD = $(LDADD)
trilearn_coach_DEPENDENCIES =
trilearn_coach_LDFLAGS =
am_trilearn_player_OBJECTS = $(am__objects_1) BasicPlayer.$(OBJEXT) \
      Player.$(OBJEXT) PlayerTeams.$(OBJEXT) main.$(OBJEXT)
trilearn_player_OBJECTS = $(am_trilearn_player_OBJECTS)
trilearn_player_LDADD = $(LDADD)
trilearn_player_DEPENDENCIES =
trilearn_player_LDFLAGS =

DEFAULT_INCLUDES =  -I. -I$(srcdir) -I$(top_builddir)
depcomp = $(SHELL) $(top_srcdir)/depcomp
am__depfiles_maybe = depfiles
DEP_FILES = ./$(DEPDIR)/ActHandler.Po \
      ./$(DEPDIR)/BasicCoach.Po \
      ./$(DEPDIR)/BasicPlayer.Po \
      ./$(DEPDIR)/Connection.Po ./$(DEPDIR)/Formations.Po \
      ./$(DEPDIR)/GenericValues.Po \
      ./$(DEPDIR)/Geometry.Po ./$(DEPDIR)/Logger.Po \
      ./$(DEPDIR)/Objects.Po ./$(DEPDIR)/Parse.Po \
      ./$(DEPDIR)/Player.Po ./$(DEPDIR)/PlayerSettings.Po \
      ./$(DEPDIR)/PlayerTeams.Po \
      ./$(DEPDIR)/SenseHandler.Po \
      ./$(DEPDIR)/ServerSettings.Po \
      ./$(DEPDIR)/SoccerTypes.Po \
      ./$(DEPDIR)/WorldModel.Po \
      ./$(DEPDIR)/WorldModelHighLevel.Po \
      ./$(DEPDIR)/WorldModelPredict.Po \
      ./$(DEPDIR)/WorldModelUpdate.Po ./$(DEPDIR)/main.Po \
      ./$(DEPDIR)/mainCoach.Po
CXXCOMPILE = $(CXX) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) \
      $(AM_CPPFLAGS) $(CPPFLAGS) $(AM_CXXFLAGS) $(CXXFLAGS)
LTCXXCOMPILE = $(LIBTOOL) --mode=compile $(CXX) $(DEFS) \
      $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) \
      $(AM_CXXFLAGS) $(CXXFLAGS)
CXXLD = $(CXX)
CXXLINK = $(LIBTOOL) --mode=link $(CXXLD) $(AM_CXXFLAGS) $(CXXFLAGS) \
      $(AM_LDFLAGS) $(LDFLAGS) -o $@
COMPILE = $(CC) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS)
\
      $(CPPFLAGS) $(AM_CFLAGS) $(CFLAGS)
LTCOMPILE = $(LIBTOOL) --mode=compile $(CC) $(DEFS) $(DEFAULT_INCLUDES)
\
```

```
      $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) $(AM_CFLAGS) $(CFLAGS)
CCLD = $(CC)
LINK = $(LIBTOOL) --mode=link $(CCLD) $(AM_CFLAGS) $(CFLAGS) \
      $(AM_LDFLAGS) $(LDFLAGS) -o $@
DIST_SOURCES = $(trilearn_coach_SOURCES) $(trilearn_player_SOURCES)
DIST_COMMON = Makefile.am Makefile.in
SOURCES = Connection.cpp \
        SenseHandler.cpp \
        ActHandler.cpp \
        SoccerTypes.cpp \
        Objects.cpp \
        WorldModel.cpp \
        WorldModelHighLevel.cpp \
        WorldModelPredict.cpp \
        WorldModelUpdate.cpp \
        ServerSettings.cpp \
        PlayerSettings.cpp \
        GenericValues.cpp \
        Formations.cpp \
        Geometry.cpp \
        Parse.cpp \
        Logger.cpp \
      Connection.h \
        SenseHandler.h \
        ActHandler.h \
        SoccerTypes.h \
        Objects.h \
        WorldModel.h \
        ServerSettings.h \
        PlayerSettings.h \
        GenericValues.h \
        Formations.h \
        Geometry.h \
        Parse.h \
        Logger.h


all: all-am

.SUFFIXES:
.SUFFIXES: .cpp .lo .o .obj
$(srcdir)/Makefile.in:  Makefile.am  $(top_srcdir)/configure.ac
$(ACLOCAL_M4)
      cd $(top_srcdir) && \
        $(AUTOMAKE) --gnu  src/Makefile
Makefile:  $(srcdir)/Makefile.in  $(top_builddir)/config.status
      cd $(top_builddir) && $(SHELL) ./config.status $(subdir)/$@
$(am__depfiles_maybe)
binPROGRAMS_INSTALL = $(INSTALL_PROGRAM)
install-binPROGRAMS: $(bin_PROGRAMS)
      @$(NORMAL_INSTALL)
      $(mkinstalldirs) $(DESTDIR)$(bindir)
      @list='$(bin_PROGRAMS)'; for p in $$list; do \
        p1=`echo $$p|sed 's/$(EXEEXT)$$//'`; \
        if test -f $$p \
           || test -f $$p1 \
        ; then \
```

```
          f=`echo "$$p1" | sed
's,^.*/,,;$(transform);s/$$/$(EXEEXT)/'`; \
          echo " $(INSTALL_PROGRAM_ENV) $(LIBTOOL) --mode=install
$(binPROGRAMS_INSTALL) $$p $(DESTDIR)$(bindir)/$$f"; \
          $(INSTALL_PROGRAM_ENV) $(LIBTOOL) --mode=install
$(binPROGRAMS_INSTALL) $$p $(DESTDIR)$(bindir)/$$f || exit 1; \
        else :; fi; \
      done

uninstall-binPROGRAMS:
      @$(NORMAL_UNINSTALL)
      @list='$(bin_PROGRAMS)'; for p in $$list; do \
        f=`echo "$$p" | sed
's,^.*/,,;s/$(EXEEXT)$$//;$(transform);s/$$/$(EXEEXT)/'`; \
        echo " rm -f $(DESTDIR)$(bindir)/$$f"; \
        rm -f $(DESTDIR)$(bindir)/$$f; \
      done

clean-binPROGRAMS:
      @list='$(bin_PROGRAMS)'; for p in $$list; do \
        f=`echo $$p|sed 's/$(EXEEXT)$$//'`; \
        echo " rm -f $$p $$f"; \
        rm -f $$p $$f ; \
      done
trilearn_coach$(EXEEXT): $(trilearn_coach_OBJECTS)
$(trilearn_coach_DEPENDENCIES)
      @rm -f trilearn_coach$(EXEEXT)
      $(CXXLINK) $(trilearn_coach_LDFLAGS) $(trilearn_coach_OBJECTS)
$(trilearn_coach_LDADD) $(LIBS)
trilearn_player$(EXEEXT): $(trilearn_player_OBJECTS)
$(trilearn_player_DEPENDENCIES)
      @rm -f trilearn_player$(EXEEXT)
      $(CXXLINK) $(trilearn_player_LDFLAGS) $(trilearn_player_OBJECTS)
$(trilearn_player_LDADD) $(LIBS)

mostlyclean-compile:
      -rm -f *.$(OBJEXT) core *.core

distclean-compile:
      -rm -f *.tab.c

include ./$(DEPDIR)/ActHandler.Po
include ./$(DEPDIR)/BasicCoach.Po
include ./$(DEPDIR)/BasicPlayer.Po
include ./$(DEPDIR)/Connection.Po
include ./$(DEPDIR)/Formations.Po
include ./$(DEPDIR)/GenericValues.Po
include ./$(DEPDIR)/Geometry.Po
include ./$(DEPDIR)/Logger.Po
include ./$(DEPDIR)/Objects.Po
include ./$(DEPDIR)/Parse.Po
include ./$(DEPDIR)/Player.Po
include ./$(DEPDIR)/PlayerSettings.Po
include ./$(DEPDIR)/PlayerTeams.Po
include ./$(DEPDIR)/SenseHandler.Po
include ./$(DEPDIR)/ServerSettings.Po
include ./$(DEPDIR)/SoccerTypes.Po
```

```
include ./$(DEPDIR)/WorldModel.Po
include ./$(DEPDIR)/WorldModelHighLevel.Po
include ./$(DEPDIR)/WorldModelPredict.Po
include ./$(DEPDIR)/WorldModelUpdate.Po
include ./$(DEPDIR)/main.Po
include ./$(DEPDIR)/mainCoach.Po

distclean-depend:
	-rm -rf ./$(DEPDIR)

.cpp.o:
	if $(CXXCOMPILE) -MT $@ -MD -MP -MF "$(DEPDIR)/$*.Tpo" \
	  -c -o $@ `test -f '$<' || echo '$(srcdir)/'`$<; \
	then mv -f "$(DEPDIR)/$*.Tpo" "$(DEPDIR)/$*.Po"; \
	else rm -f "$(DEPDIR)/$*.Tpo"; exit 1; \
	fi
#	source='$<' object='$@' libtool=no \
#	depfile='$(DEPDIR)/$*.Po' tmpdepfile='$(DEPDIR)/$*.TPo' \
#	$(CXXDEPMODE) $(depcomp) \
#	$(CXXCOMPILE) -c -o $@ `test -f '$<' || echo '$(srcdir)/'`$<

.cpp.obj:
	if $(CXXCOMPILE) -MT $@ -MD -MP -MF "$(DEPDIR)/$*.Tpo" \
	  -c -o $@ `if test -f '$<'; then $(CYGPATH_W) '$<'; else
$(CYGPATH_W) '$(srcdir)/$<'; fi`; \
	then mv -f "$(DEPDIR)/$*.Tpo" "$(DEPDIR)/$*.Po"; \
	else rm -f "$(DEPDIR)/$*.Tpo"; exit 1; \
	fi
#	source='$<' object='$@' libtool=no \
#	depfile='$(DEPDIR)/$*.Po' tmpdepfile='$(DEPDIR)/$*.TPo' \
#	$(CXXDEPMODE) $(depcomp) \
#	$(CXXCOMPILE) -c -o $@ `if test -f '$<'; then $(CYGPATH_W) '$<';
else $(CYGPATH_W) '$(srcdir)/$<'; fi`

.cpp.lo:
	if $(LTCXXCOMPILE) -MT $@ -MD -MP -MF "$(DEPDIR)/$*.Tpo" \
	  -c -o $@ `test -f '$<' || echo '$(srcdir)/'`$<; \
	then mv -f "$(DEPDIR)/$*.Tpo" "$(DEPDIR)/$*.Plo"; \
	else rm -f "$(DEPDIR)/$*.Tpo"; exit 1; \
	fi
#	source='$<' object='$@' libtool=yes \
#	depfile='$(DEPDIR)/$*.Plo' tmpdepfile='$(DEPDIR)/$*.TPlo' \
#	$(CXXDEPMODE) $(depcomp) \
#	$(LTCXXCOMPILE) -c -o $@ `test -f '$<' || echo '$(srcdir)/'`$<

mostlyclean-libtool:
	-rm -f *.lo

clean-libtool:
	-rm -rf .libs _libs

distclean-libtool:
	-rm -f libtool
uninstall-info-am:

ETAGS = etags
ETAGSFLAGS =
```

```
CTAGS = ctags
CTAGSFLAGS =

tags: TAGS

ID: $(HEADERS) $(SOURCES) $(LISP) $(TAGS_FILES)
      list='$(SOURCES) $(HEADERS) $(LISP) $(TAGS_FILES)'; \
      unique=`for i in $$list; do \
          if test -f "$$i"; then echo $$i; else echo $(srcdir)/$$i; fi;
\
        done | \
        $(AWK) '    { files[$$0] = 1; } \
             END { for (i in files) print i; }'`; \
      mkid -fID $$unique

TAGS:  $(HEADERS) $(SOURCES)  $(TAGS_DEPENDENCIES) \
            $(TAGS_FILES) $(LISP)
      tags=; \
      here=`pwd`; \
      list='$(SOURCES) $(HEADERS)  $(LISP) $(TAGS_FILES)'; \
      unique=`for i in $$list; do \
          if test -f "$$i"; then echo $$i; else echo $(srcdir)/$$i; fi;
\
        done | \
        $(AWK) '    { files[$$0] = 1; } \
             END { for (i in files) print i; }'`; \
      test -z "$(ETAGS_ARGS)$$tags$$unique" \
        || $(ETAGS) $(ETAGSFLAGS) $(AM_ETAGSFLAGS) $(ETAGS_ARGS) \
           $$tags $$unique

ctags: CTAGS
CTAGS:  $(HEADERS) $(SOURCES)  $(TAGS_DEPENDENCIES) \
            $(TAGS_FILES) $(LISP)
      tags=; \
      here=`pwd`; \
      list='$(SOURCES) $(HEADERS)  $(LISP) $(TAGS_FILES)'; \
      unique=`for i in $$list; do \
          if test -f "$$i"; then echo $$i; else echo $(srcdir)/$$i; fi;
\
        done | \
        $(AWK) '    { files[$$0] = 1; } \
             END { for (i in files) print i; }'`; \
      test -z "$(CTAGS_ARGS)$$tags$$unique" \
        || $(CTAGS) $(CTAGSFLAGS) $(AM_CTAGSFLAGS) $(CTAGS_ARGS) \
           $$tags $$unique

GTAGS:
      here=`$(am__cd) $(top_builddir) && pwd` \
        && cd $(top_srcdir) \
        && gtags -i $(GTAGS_ARGS) $$here

distclean-tags:
      -rm -f TAGS ID GTAGS GRTAGS GSYMS GPATH tags
DISTFILES = $(DIST_COMMON) $(DIST_SOURCES) $(TEXINFOS) $(EXTRA_DIST)

top_distdir = ..
```

```
distdir = $(top_distdir)/$(PACKAGE)-$(VERSION)

distdir: $(DISTFILES)
      @srcdirstrip=`echo "$(srcdir)" | sed 's|.|.|g'`; \
      topsrcdirstrip=`echo "$(top_srcdir)" | sed 's|.|.|g'`; \
      list='$(DISTFILES)'; for file in $$list; do \
        case $$file in \
          $(srcdir)/*) file=`echo "$$file" | sed
"s|^$$srcdirstrip/||"`;; \
          $(top_srcdir)/*) file=`echo "$$file" | sed
"s|^$$topsrcdirstrip/|$(top_builddir)/|"`;; \
        esac; \
        if test -f $$file || test -d $$file; then d=.; else
d=$(srcdir); fi; \
        dir=`echo "$$file" | sed -e 's,/[^/]*$$,,'`; \
        if test "$$dir" != "$$file" && test "$$dir" != "."; then \
          dir="/$$dir"; \
          $(mkinstalldirs) "$(distdir)$$dir"; \
        else \
          dir=''; \
        fi; \
        if test -d $$d/$$file; then \
          if test -d $(srcdir)/$$file && test $$d != $(srcdir); then \
            cp -pR $(srcdir)/$$file $(distdir)$$dir || exit 1; \
          fi; \
          cp -pR $$d/$$file $(distdir)$$dir || exit 1; \
        else \
          test -f $(distdir)/$$file \
          || cp -p $$d/$$file $(distdir)/$$file \
          || exit 1; \
        fi; \
      done
check-am: all-am
check: check-am
all-am: Makefile $(PROGRAMS)

installdirs:
      $(mkinstalldirs) $(DESTDIR)$(bindir)
install: install-am
install-exec: install-exec-am
install-data: install-data-am
uninstall: uninstall-am

install-am: all-am
      @$(MAKE) $(AM_MAKEFLAGS) install-exec-am install-data-am

installcheck: installcheck-am
install-strip:
      $(MAKE) $(AM_MAKEFLAGS)
INSTALL_PROGRAM="$(INSTALL_STRIP_PROGRAM)" \
        INSTALL_STRIP_FLAG=-s \
        `test -z '$(STRIP)' || \
          echo "INSTALL_PROGRAM_ENV=STRIPPROG='$(STRIP)'"` install
mostlyclean-generic:

clean-generic:
```

```
distclean-generic:
	-rm -f Makefile $(CONFIG_CLEAN_FILES)

maintainer-clean-generic:
	@echo "This command is intended for maintainers to use"
	@echo "it deletes files that may require special tools to
rebuild."
clean: clean-am

clean-am: clean-binPROGRAMS clean-generic clean-libtool mostlyclean-am

distclean: distclean-am

distclean-am: clean-am distclean-compile distclean-depend \
	distclean-generic distclean-libtool distclean-tags

dvi: dvi-am

dvi-am:

info: info-am

info-am:

install-data-am:

install-exec-am: install-binPROGRAMS

install-info: install-info-am

install-man:

installcheck-am:

maintainer-clean: maintainer-clean-am

maintainer-clean-am: distclean-am maintainer-clean-generic

mostlyclean: mostlyclean-am

mostlyclean-am: mostlyclean-compile mostlyclean-generic \
	mostlyclean-libtool

pdf: pdf-am

pdf-am:

ps: ps-am

ps-am:

uninstall-am: uninstall-binPROGRAMS uninstall-info-am

.PHONY: CTAGS GTAGS all all-am check check-am clean clean-binPROGRAMS \
	clean-generic clean-libtool ctags distclean distclean-compile \
	distclean-depend distclean-generic distclean-libtool \
	distclean-tags distdir dvi dvi-am info info-am install \
```

```
	install-am install-binPROGRAMS install-data install-data-am \
	install-exec install-exec-am install-info install-info-am \
	install-man install-strip installcheck installcheck-am \
	installdirs maintainer-clean maintainer-clean-generic \
	mostlyclean mostlyclean-compile mostlyclean-generic \
	mostlyclean-libtool pdf pdf-am ps ps-am tags uninstall \
	uninstall-am uninstall-binPROGRAMS uninstall-info-am

# Tell versions [3.59,3.63) of GNU make to not export all variables.
# Otherwise a system limit (for SysV at least) may be exceeded.
.NOEXPORT:
```

start.sh

```
#!/bin/tcsh

# This script starts the UvA_Trilearn_2003 team. When player numbers are
# supplied before the (optional) host-name and team-name arguments only
# these players are started, otherwise all players are started.
# Usage:   start.sh <player-numbers> <host-name> <team-name>
#
# Example: start.sh               all players default host and name
# Example: start.sh machine       all players on host 'machine'
# Example: start.sh localhost UvA   all players on localhost and name 'UvA'
# Example: start.sh 127.0.0.1 UvA   all players on 127.0.0.1  and name 'UvA'
# Example: start.sh 1 2 3 4       players 1-4 on default host and name
# Example: start.sh 1 2 remote    players 1-2 on host 'remote'
# Example: start.sh 9 10 remote UvA players 9-10 on host remote and name 'UvA'
# Example: start.sh 0             start coach on default host

set wait  = 0
set host  = "localhost"
set team  = "uva_1"
set dir   = "src"
set prog  = "${dir}/trilearn_player"
set coach = "${dir}/trilearn_coach"
set pconf = "${dir}/player.conf"
set fconf = "${dir}/formations.conf"

echo "***************************************************************"
echo "* UvA_Trilearn 2003 - University of Amsterdam, The Netherlands  *"
echo "* Base code version                                           *"
echo "* Created by:          Jelle Kok                               *"
echo "* Research Coordinator: Nikos Vlassis                          *"
echo "* Team Coordinator:     Frans Groen                            *"
echo "* Copyright 2000-2001.  Jelle Kok and Remco de Boer            *"
echo "* Copyright 2001-2002.  Jelle Kok                              *"
echo "* Copyright 2002-2003.  Jelle Kok                              *"
echo "* All rights reserved.                                        *"
echo "***************************************************************"

#first check if the last two supplied arguments are no numbers and represent
#<host-name> or <host-name> <team-name>
if( $#argv > 0 && ($argv[$#argv] !~ [0123456789]* || $argv[$#argv] =~ *.* ) ) then
  @ second_last = $#argv - 1
  if( $#argv > 1 && ($argv[$second_last] !~ [0123456789]* || $argv[$second_last] =~ *.*
) ) then
     set host = $argv[$second_last]
     set team = $argv[$#argv]
  else
```

```
      set host = $argv[$#argv]
  endif
endif

#then if first argument is a number, start only the players with the numbers
#as supplied on the prompt, otherwise start all players.
if( $1 =~ [0123456789]* && $1 !~ *.* ) then
  echo "$argv[$#argv]"
    echo "$1"
  foreach arg ($argv)
    if( $arg =~ [123456789]* && $arg !~ *.*) then
      ${prog} -num ${arg} -host ${host} -team ${team} -f ${fconf} -c ${pconf} &
      sleep $wait
    else if( $arg =~ [0]* ) then
      sleep 2
      ${coach} -host ${host} -team ${team} -f ${fconf} &
    endif
  end
else
  set i = 1
  while ( ${i} <12 )
    ${prog} -number ${i} -host ${host} -team ${team}  -f ${fconf} -c ${pconf} &
    sleep $wait
    @ i++
  end
  sleep 2
  ${coach} -host ${host} -team ${team} -f ${fconf}  &
endif
```