RoboSoccerLib 2D

Manual do Usuário

Bruno Marques, André Lucas
Pablo Sampaio

Índice

1. Introdução

A RoboSoccerLib tem por objetivo a ser uma biblioteca que auxilie na criação de times para ser usado em competições da RoboCup. Implementada por alunos da Universidade Federal Rural de Pernambuco, essa biblioteca irá facilitar o acesso regras que vêm do servidor da RoboCup, mapeando todo o conteúdo da biblioteca em forma de orientação objetos.

A biblioteca foi desenvolvida em Java (versão 1.8) e é baseada nas percepções dos jogadores, transformando essas percepções enviadas pelo servidor em posições no campo. Essas posições ficam padronizadas em forma de vetores bidimensionais, tendo em vista disso, todo conteúdo das percepções serão em forma de vetores.

Mais detalhes sobre as percepções que vêm do servidor se encontram no seu manual, o link está na nossa bibliografia.

2. Começando

Nesse tópico iremos baixar a versão recente da biblioteca, preparar o ambiente e configurá-la para o projeto de um time.

2.1. Instalando a biblioteca

Inicialmente deve-se instalar a biblioteca para obter todas os métodos e comandos, bem como a gerência da comunicação entre os agentes (jogadores) e o servidor. Para este tutorial será usado o Eclipse IDE Luna Service Release 2 (4.4.2).

2.1.1. Baixe ou Atualize a Biblioteca:

A biblioteca se encontra no repositório do Bitbucket acessado pelo link https://bitbucket.org/brunohpmarques/robosoccerlib. Baixe para seguir os próximos passos.

2.1.2. Defina um Workspace:

Execute o Eclipse IDE e selecione um *workspace* onde ficará a biblioteca e o projeto do seu time. Neste exemplo será usado um diretório com endereço “C:\Tutorial”. Confira a imagem abaixo (figura 2.1.A):

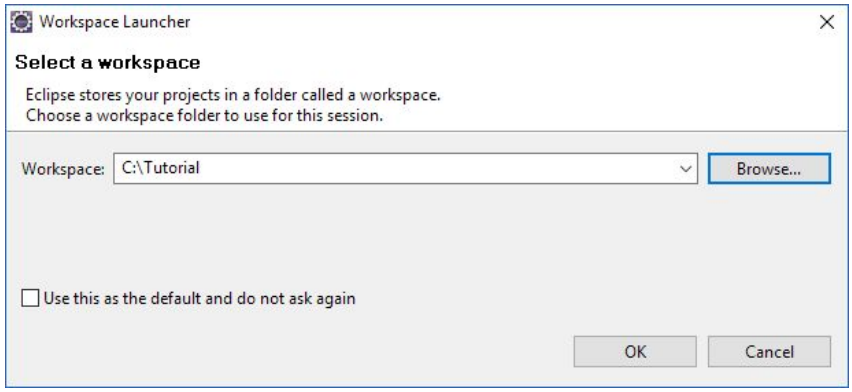

FIGURA 2.1.A

2.1.3. Importe a Biblioteca:

Adicione a biblioteca baixada no diretório do *workspace.* Neste exemplo: “C:\Tutorial\Simple Soccer Lib 1”. Em seguida, no Eclipse IDE, importe-a conforme a figura 2.1.B.

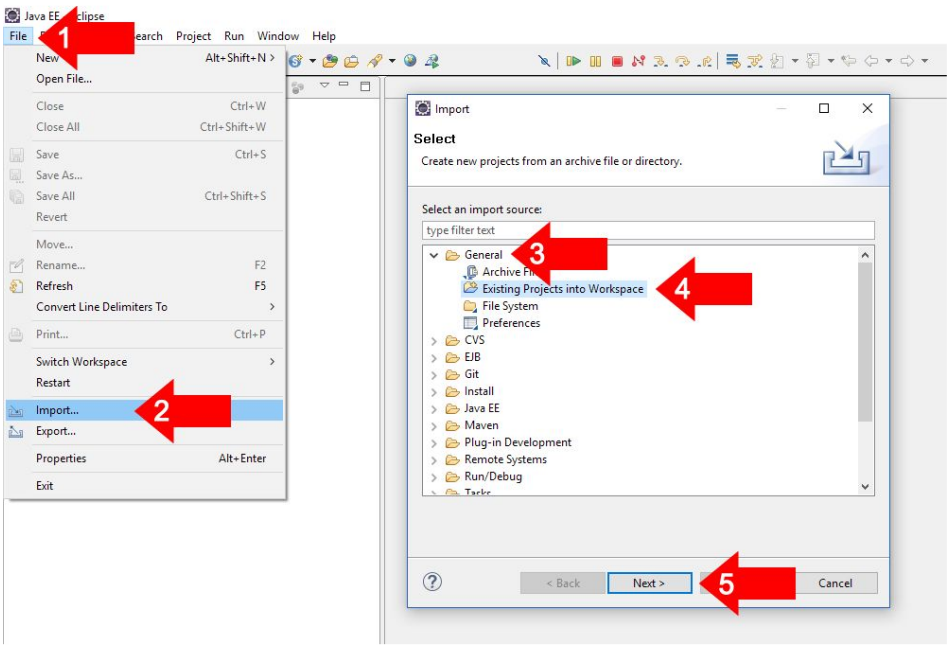

FIGURA 2.1.B

Nos próximos passos, encontre e selecione a pasta da biblioteca clicando no botão “Browse…”. Logo após, clique em “OK”. Acompanhe a figura (2.1.C) abaixo.

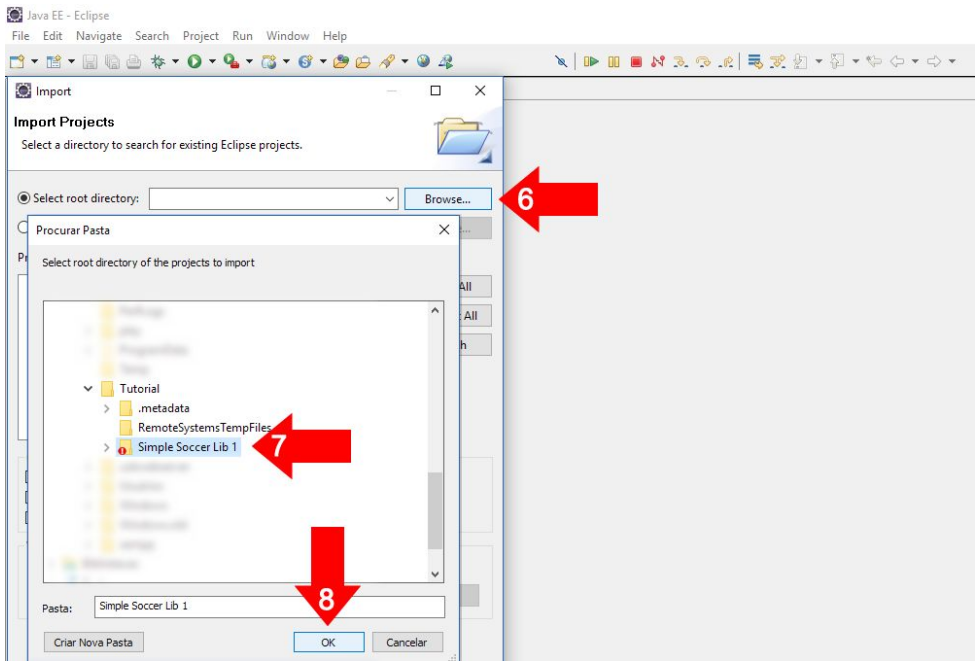

FIGURA 2.1.C

Para terminar, clique em “Finish”. O Eclipse IDE irá importar a biblioteca e mostrará o conteúdo na sessão “Project Explorer” como apresenta a figura 2.1.D.

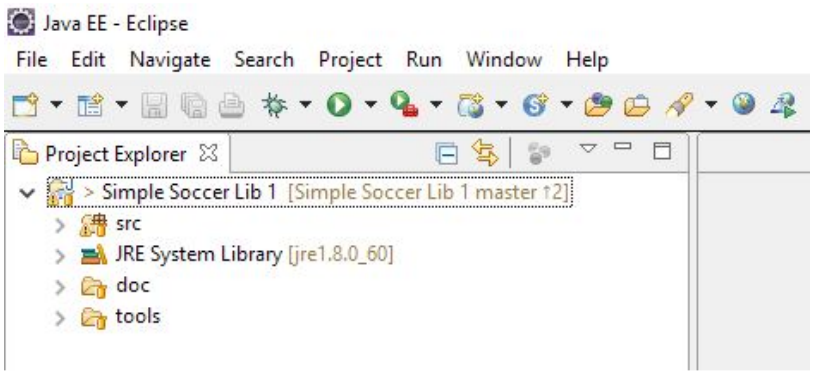

FIGURA 2.1.D

2.1.4. Inicie um Projeto:

Na sessão “Project Explorer” clique com o botão direito e siga os passos da figura 2.1.E.

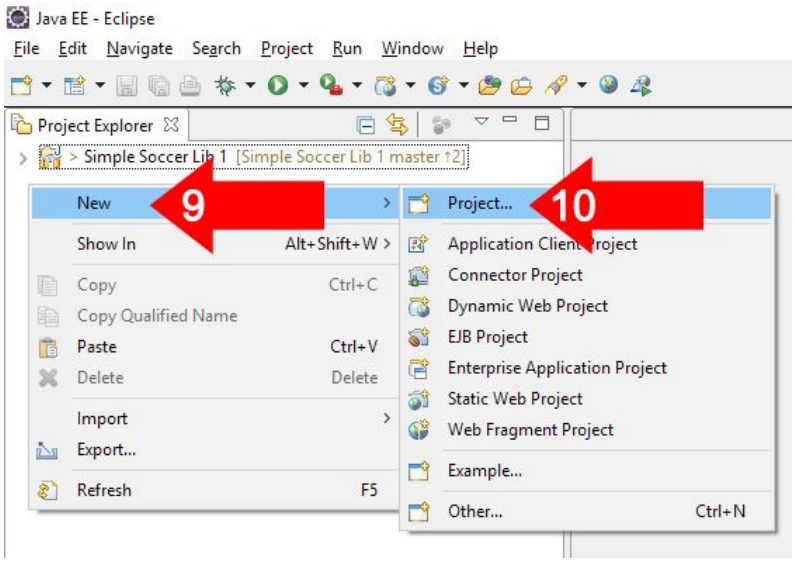

FIGURA 2.1.E

Na próxima janela, defina um nome para seu projeto, certifique que a versão do Java é a 1.8 e clique em “Finish”. Continue seguindo a figura.

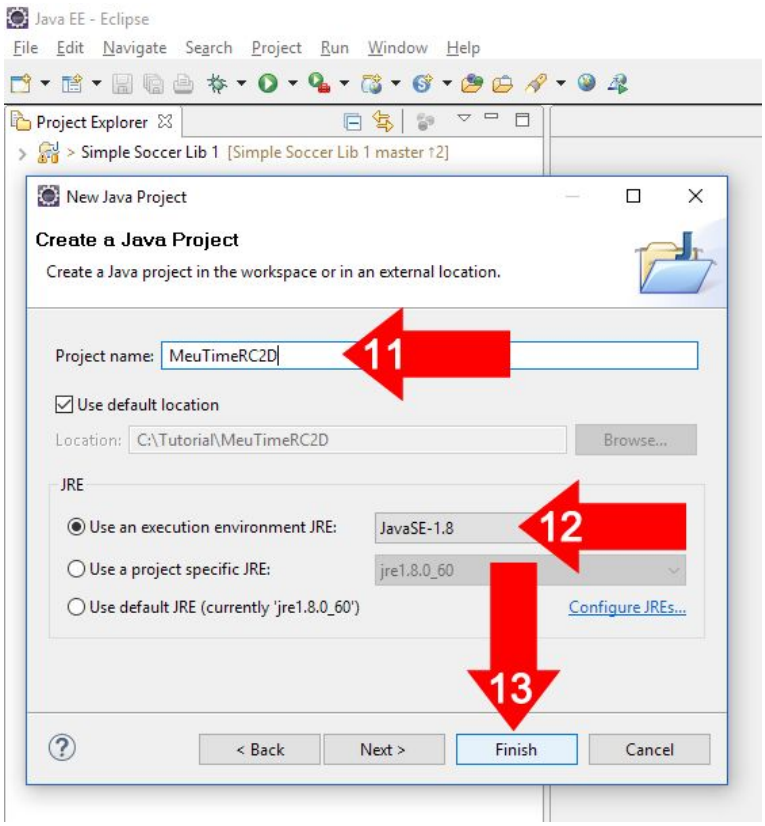

FIGURA 2.1.F

O Eclipse IDE irá iniciará o projeto e mostrará o conteúdo na sessão “Package Explorer” como apresenta a figura 2.1.G.

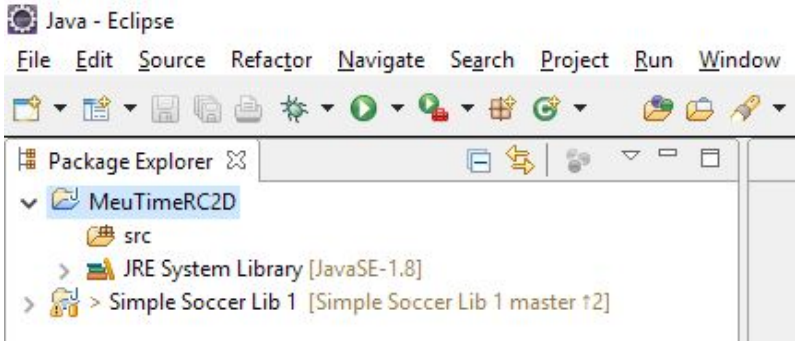

FIGURA 2.1.G

2.1.5. Adicione a Biblioteca no Projeto:

Os próximos passos (figura 2.1.H) irão adicionar a biblioteca no projeto recém criado disponibilizando, assim, os métodos e comandos para a implementação dos agentes.

Na sessão “Package Explorer” clique com o botão direito em “JRE System Library” e siga os passos abaixo.

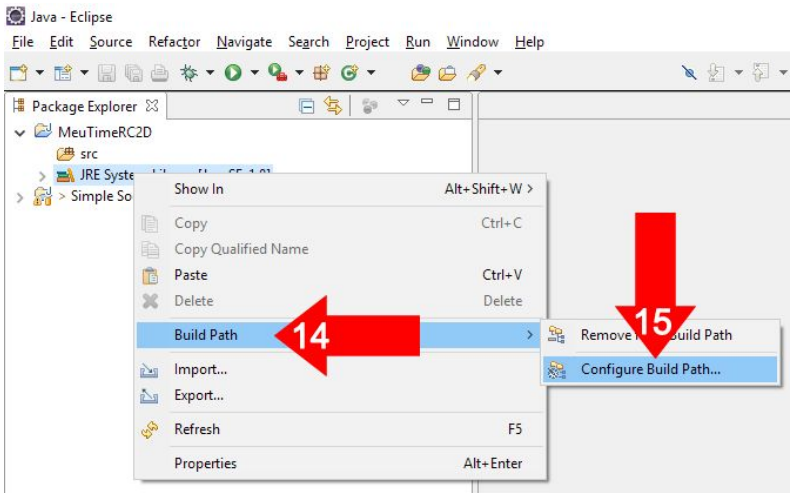

FIGURA 2.1.H

A janela de propriedades do projeto será mostrada. Acompanhe a figura 2.1.I. Clique na aba “Projects”, selecione a biblioteca “Simple Soccer Lib 1” e clique em “OK”. Veja que a biblioteca foi importada na lista da aba “Projects” e, na mesma janela, clique em “OK”.

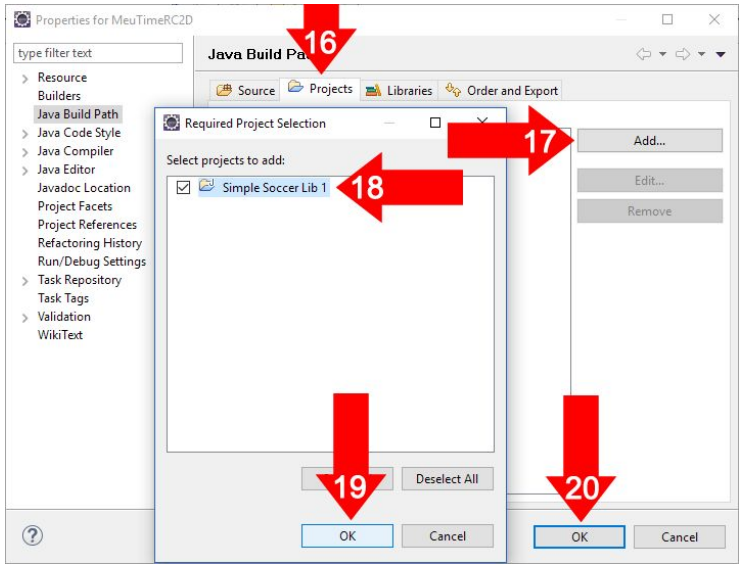

FIGURA 2.1.I

A biblioteca está importada, configurada e pronta para ser utilizada no projeto.

2.2. Iniciando um Time

Nesse tópico, mostraremos como iniciar um time básico com 4 agentes (jogadores), sendo eles: um goleiro, dois armadores e um atacante.

2.2.1. Estrutura do Projeto:

Para começar, é importante definir a estrutura do projeto para separar a responsabilidade de cada um dos módulos em três classes: a) *CommandPlayer*, classe onde as ações dos jogadores serão implementadas; b) *CommandTeam*, objeto que representa o time; e c) *Main*, que iniciará o time e instanciar os jogadores. Crie três classes no projeto do time com os nomes citados acima. Veja a figura 2.2.A.

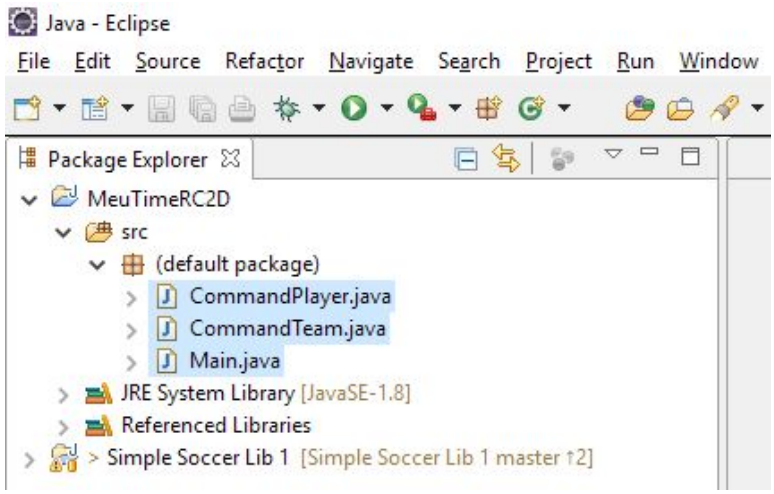

FIGURA 2.2.A

2.2.2. Classe CommandPlayer:

O objetivo desta classe é conter a implementação das ações dos jogadores (agentes) do time. Como cada jogador tem ação independente, esta classe precisa estender da classe **Thread** e para executarmos essas ações, precisamos sobrescrever o método *run()*. Observe a listagem abaixo:

```
public class CommandPlayer extends Thread {
      private int LOOP_INTERVAL = 100;  //0.1s

      private PlayerCommander commander;
      private PlayerPerception selfPerc;
      private FieldPerception  fieldPerc;
      private MatchPerception matchPerc;

      public CommandPlayer(PlayerCommander player) {
            commander = player;
      }

      @Override
      public void run() {
            ...
      }

}
```

O atributo LOOP_INTERVAL deve ser setado com valor 100 para que cada comando seja enviado no intervalo de tempo definido como regra do

jogo. O servidor não executará comandos enviados em intervalos abaixo desse valor.

Os outros atributos serão fundamentais para armazenar os dados da partida, do campo, do próprio agente e o gerenciador de comandos (envio de ações para o servidor).

O construtor receberá a instância do gerenciador de comandos assim que houver a conexão do jogador com o servidor.

Sabendo dos atributos básicos, a listagem abaixo apresenta um código simples para definirmos o que cada jogador fará.

```
@Override
public void run() {
      System.out.println(">> Executando...");
      long nextIteration =
            System.currentTimeMillis() + LOOP_INTERVAL;
      updatePerceptions();
      switch (selfPerc.getUniformNumber()) {
      case 1:
            acaoGoleiro(nextIteration);
            break;
      case 2:
            acaoArmador(nextIteration, -1); // cima
            break;
      case 3:
            acaoArmador(nextIteration, 1); // baixo
            break;
      case 4:
            acaoAtacante(nextIteration);
            break;
      default: break;
      }
}
```

Inicialmente precisamos calcular o *nextIteration* com base no valor da regra do servidor explicado anteriormente. Separamos a posição e suas respectivas ações com um *switch* o qual chamará um método com a ação destinada à posição (número do uniforme 1 é o goleiro, 2 e 3 serão armadores e 4 o atacante). Note que o método *acaoArmador()* recebe um inteiro como parâmetro, isso servirá para posicionar o jogador na área

superior ou inferior do campo, mas não se preocupe com isso agora, iremos detalhar no tópico 2.3.

O método *updatePerceptions()* é fundamental para atualizar as percepções do ambiente (partida, campo e jogador). Este será explicitado na listagem abaixo:

```
private void updatePerceptions() {
      PlayerPerception newSelf =
            commander.perceiveSelfBlocking();
      FieldPerception newField =
            commander.perceiveFieldBlocking();
      MatchPerception newMatch =
            commander.perceiveMatchBlocking();

      if (newSelf != null) this.selfPerc = newSelf;
      if (newField != null) this.fieldPerc = newField;
      if (newMatch != null) this.matchPerc = newMatch;
}
```

2.2.3. Classe CommandTeam:

Neste item mostraremos o código para lançar um time. Confira a listagem a seguir:

```
public class CommandTeam extends AbstractTeam  {
      public CommandTeam() {
            super("TeamA", 4);
      }

      @Override
      protected void launchPlayer(int ag, PlayerCommander comm) {
            System.out.println("Player lançado!");
            CommandPlayer p = new CommandPlayer(comm);
            p.start();
      }
}
```

De início, a classe que representa o time deverá estender de *AbstractTeam*, encontrada na biblioteca. Isso obrigará a implementação de um construtor para instanciar a classe superior, recebendo o nome do time (nomes grandes não serão aceitos pelo servidor) e a quantidade de

jogadores; e do método *launchPlayer()* que recebe o número do jogador e o gerenciador de comandos já instanciado para o agente se comunicar com o servidor. Em seguida, devemos instanciar um jogador do tipo CommandPlayer, implementado no tópico anterior, e inicia-lo em *p.start()*.

2.2.4. Classe Main:

Tendo a implementação da classe que representa cada jogador e da classe que representa o time, falta apenas implementar o código que irá executar o servidor, o monitor e o time com seus jogadores. Analise a listagem abaixo.

```
public class Main {
      public static void main(String[] args) {
            try {
                  CommandTeam teamA = new CommandTeam();
                  teamA.launchTeamAndServer();
            } catch (UnknownHostException e) {
                  System.out.println("Falha ao conectar.");
            }
      }
}
```

Instanciamos o *teamA* do tipo *CommandTeam* que implementamos no tópico anterior, em seguida, executamos o servidor, o monitor e lançamos o time apenas com a chamada do método *launchTeamAndServer()*. Este método lança *UnknownHostException* como exceção caso ocorra algum problema na comunicação com o servidor.

Assim terminamos de desenvolver a base de um time com 4 jogadores que já se conecta com o servidor. As figuras abaixo mostra o console do servidor (figura 2.2.B) e o monitor com o time iniciado (figura 2.2.C).

FIGURA 2.2.B

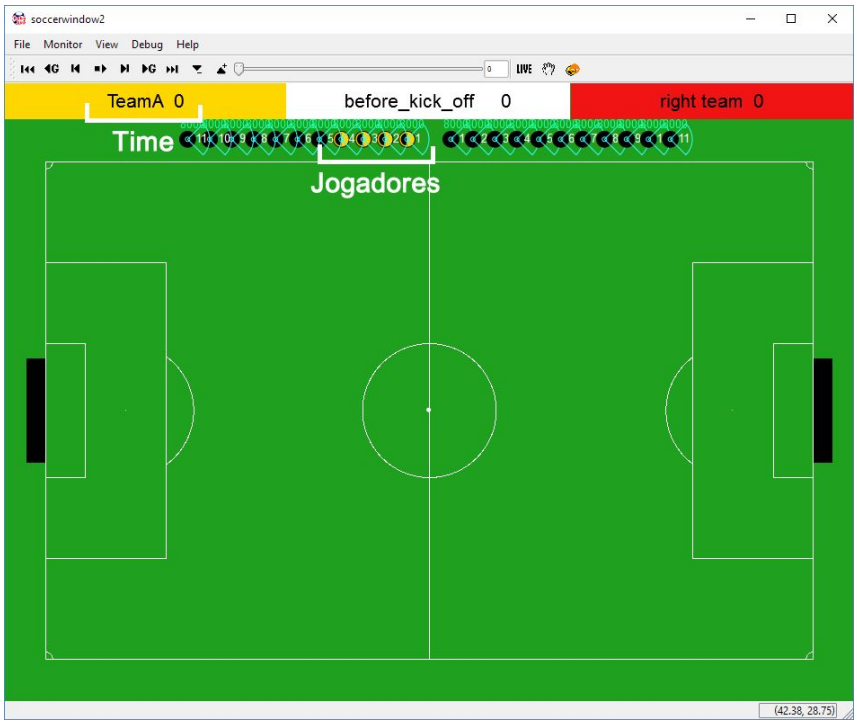

FIGURA 2.2.C

2.3. Executando Ações de Teste no Time

Neste tópico iremos apresentar um exemplo de posicionamento simples e ações básicas dos jogadores. Iremos aproveitar o código base dos 4 jogadores implementados anteriormente e apresentar um comportamento para cada.

2.3.1. Ações Básicas:

Inicialmente é importante implementar ações comuns como as listagens a seguir:

Girar para um ponto:

```
private void turnToPoint(Vector2D point){
     Vector2D newDirection = point.sub(selfPerc.getPosition());
     commander.doTurnToDirectionBlocking(newDirection);
}
```

Correr:

```
private void dash(Vector2D point){
     if(selfPerc.getPosition().distanceTo(point) <= 1) return;
```

```
      if(!isAlignToPoint(point, 15)) turnToPoint(point);
      commander.doDashBlocking(70);
}
```

A segunda verificação presente no código acima evita a execução do comando de *girar para um ponto* sem necessidade. Se o jogador estiver desalinhado (fora da faixa -15 e 15 graus) em relação ao ponto, u comando de *girar para um ponto* é executado. A primeira verificação é a condição de parada, ou seja, se o jogador já estiver próximo do ponto (com margem até 1 de precisão) não é necessário executar o comando de correr. Vale lembrar da inércia contida no jogo, isto é, se o jogador estiver em velocidade alta, não conseguirá executar uma parada brusca e assim, na maioria dos casos, o jogador ficará *correndo em volta do ponto*, nunca chegando em seu destino.

Chutar para um ponto:

```
private void kickToPoint(Vector2D point, double intensity){
      Vector2D newDirection = point.sub(selfPerc.getPosition());
      double angle = newDirection
                        .angleFrom(selfPerc.getDirection());
      if(angle > 90 || angle < -90){
            commander.doTurnToDirectionBlocking(newDirection);
            angle = 0;
      }
      commander.doKickBlocking(intensity, angle);
}
```

A verificação presente no código acima faz o jogador *girar para um ponto* caso esse ponto esteja fora do ângulo de chute (entre -90 e 90 graus). Vale lembrar que o servidor simula o vento e o atrito no ambiente, isso significa que a bola não chegará no ponto exato.

Alinhado com o ponto?

```
private boolean isAlignToPoint(Vector2D point, double margin){
      double angle = point.sub(selfPerc.getPosition())
                        .angleFrom(selfPerc.getDirection());
      return angle < margin && angle > margin*(-1);
}
```

Retorna verdadeiro caso o jogador esteja olhando para um ponto obedecendo uma margem em graus. Caso contrário, retorna falso.

Os pontos estão próximos?

```
private boolean isPointsAreClose(Vector2D reference,
                                 Vector2D point, double margin){
      return reference.distanceTo(point) <= margin;
}
```

Retorna verdadeiro caso o ponto de referência esteja próximo de um ponto obedecendo uma margem de distância. Caso contrário, retorna falso.

Qual o jogador mais próximo do ponto?

```
private PlayerPerception getClosestPlayerPoint(Vector2D point,
                                 EFieldSide side, double margin){
      ArrayList<PlayerPerception> lp = fieldPerc
                                       .getTeamPlayers(side);
      PlayerPerception np = null;
      if(lp != null && !lp.isEmpty()){
            double dist,temp;
            dist = lp.get(0).getPosition().distanceTo(point);
            np = lp.get(0);
            if(isPointsAreClose(np.getPosition(), point, margin))
                  return np;
            for (PlayerPerception p : lp) {
                  if(p.getPosition() == null) break;
                  if(isPointsAreClose(p.getPosition(),
                                       point, margin))
                        return p;
                  temp = p.getPosition().distanceTo(point);
                  if(temp < dist){
                        dist = temp;
                        np = p;
                  }
            }
      }
      return np;
}
```

Retorna a percepção do jogador de um time (*side*) mais próximo do ponto.

Com essas ações já é possível fazer um esquema tático mantendo a formação do time, como por exemplo, é possível coordenar todos os jogadores de modo que não fiquem correndo atrás da bola, mas sim apenas aquele mais próximo dela.

2.3.2. Ações para o Goleiro:

A listagem abaixo apresenta a implementação do método *acaoGoleiro()* citado no tópico 2.2.3. Sugere ações básicas de posicionamento e defesa para o goleiro.

```
private void acaoGoleiro(long nextIteration) {
      double xInit=-48, yInit=0, ballX=0, ballY=0;
      EFieldSide side = selfPerc.getSide();
      Vector2D initPos =
            new Vector2D(xInit*side.value(), yInit*side.value());
      Vector2D ballPos;
      Rectangle area = side == EFieldSide.LEFT?
                  new Rectangle(-52, -20, 16, 40):
                  new Rectangle(36, -20, 16, 40);
      while (true) {
            updatePerceptions();
            ballPos = fieldPerc.getBall().getPosition();

            switch (matchPerc.getState()) {
            case BEFORE_KICK_OFF:
                  // posiciona
                  commander.doMoveBlocking(xInit, yInit);
                  break;
            case PLAY_ON:
                  ballX=fieldPerc.getBall().getPosition().getX();
                  ballY=fieldPerc.getBall().getPosition().getY();
                  if(isPointsAreClose(selfPerc.getPosition(),
                                                ballPos, 1)){
                        // chutar
                        kickToPoint(new Vector2D(0,0), 100);
                  }else if(area.contains(ballX, ballY)){
                        // defender
                        dash(ballPos);
                  }else if(!isPointsAreClose(selfPerc.getPosition(),
                                                initPos, 3)){
```

```
                    // recuar
                    dash(initPos);
                }else {
                    // olhar para a bola
                    turnToPoint(ballPos);
                }
                break;

            /* Todos os estados da partida */

            default: break;
            }
        }
}
```

As variáveis *xInit*, *yInit* guardam as coordenadas da posição inicial do goleiro no campo, mais abaixo o objeto *Vector2D* é instanciado com esses valores; *ballX* e *ballY* são variáveis para armazenar as coordenadas da bola no campo; *side* é o lado do goleiro no campo (esquerdo ou direito). A variável *area*, do tipo *java.awt.Rectangle* possui os pontos mapeados da grande área instanciada de acordo com o lado do goleiro, ela servirá para verificar se a bola está dentro da área do goleiro.

Para que o jogador fique conectado, devemos deixá-lo em um laço infinito, e de acordo com o estado da partida devemos executar alguma ação. O primeiro estado é o BEFORE_KICK_OFF, onde só é possível executar os comandos *doMove()* ou *doMoveBlocking(),* usamos algum destes para posicionar o goleiro em um lugar adequado antes de iniciar a partida. Em seguida, implementamos o estado da partida iniciada e em execução: PLAY_ON (veja todos os estados nos próximos tópicos).

Em ordem de prioridade de execução implementamos a ação de *chutar* para frente quando o goleiro estiver em uma posição adequada (distância entre sua posição e a bola deve ser menor que 1). Seguindo a ordem, verificamos se a bola está na área do goleiro e fazemos com que ele avance até a bola para *defender*, forçando a execução do código de *chutar* quando se aproximar o suficiente dela. Caso a bola não esteja na área do goleiro e ele estiver fora da sua posição inicial, fazemos com que *recue* para seu posto. Se nenhum dos casos anteriores forem satisfeitos, fazemos com que o goleiro apenas *olhe para a bola*.

2.3.3. Ações para o Armador:

A listagem abaixo apresenta a implementação do método *acaoArmador()* citado no tópico 2.2.3. Sugere ações básicas de posicionamento, defesa, e armação de uma jogada.

```
private void acaoArmador(long nextIteration, int pos) {
      double xInit=-30, yInit=10*pos;
      EFieldSide side = selfPerc.getSide();
      Vector2D initPos =
            new Vector2D(xInit*side.value(), yInit*side.value());
      Vector2D ballPos, vTemp;
      PlayerPerception pTemp;
      while (true) {
            updatePerceptions();
            ballPos = fieldPerc.getBall().getPosition();

            switch (matchPerc.getState()) {
            case BEFORE_KICK_OFF:
                  commander.doMoveBlocking(xInit, yInit);
                  break;
            case PLAY_ON:
                  if(isPointsAreClose(selfPerc.getPosition(),
                                                ballPos, 1)){
                        if(selfPerc.getUniformNumber() == 2){
                              // toca para o jogador 3
                              vTemp = fieldPerc.getTeamPlayer(
                                    side, 3).getPosition();
                        }else{
                              // toca para o jogador 4
                              vTemp = fieldPerc.getTeamPlayer(
                                    side, 4).getPosition();
                        }
                        Vector2D vTempF = vTemp.sub(selfPerc
                                          .getPosition());
                        double intensity =
                                    (vTempF.magnitude()*100)/40;
                        kickToPoint(vTemp, intensity);
                  }else{
                        pTemp = getClosestPlayerPoint(ballPos,
                                                side, 3);
                        if(pTemp != null &&
                        pTemp.getUniformNumber() == selfPerc
```

```
                                        .getUniformNumber()){
                        // pega a bola
                        dash(ballPos);
                    }else if(!isPointsAreClose(selfPerc
                              .getPosition(), initPos, 3)){
                        // recua
                        dash(initPos);
                    }else{
                        // olha para a bola
                        turnToPoint(ballPos);
                    }
                }
                break;

            /* Todos os estados da partida */

            default:
                break;
            }
        }
}
```

Como dito anteriormente, este método recebe o parâmetro *pos* afim de deixar o jogador na área superior (uniforme 2) ou inferior (uniforme 3) do campo. Área superior significa a área da metade do campo até a lateral de cima, o mesmo para a área inferior porém, obviamente, até a lateral inferior. As variáveis *xInit* e *yInit* representam as coordenadas da posição inicial e para *yInit* calculamos de acordo com o parâmetro *pos*.

Em ordem de prioridade implementamos o caso quando o jogador estiver com a bola, ou seja, perto o suficiente para chutar (distância 1 da bola). Como temos dois armadores, caso o jogador que esteja com a bola for o armador de uniforme 2, adquirimos a posição do jogador 3 e *tocamos a bola* para ele. Caso o armador tenha uniforme 3, *tocamos a bola* para o jogador 4, fazendo assim, uma jogada da defesa para o ataque. Se os armadores não estiverem perto da bola, verificamos se ele é o mais próximo com objetivo de *pegá-la*. Verificamos em seguida, se o jogador está fora da posição inicial, fazendo com que *recue* até seu posto. Se nenhum dos casos anteriores forem satisfeitos, o jogador apenas *olhará para a bola*.

2.3.4. Ações para o Atacante:

A listagem abaixo apresenta a implementação do método *acaoAtacante()* citado no tópico 2.2.3. Sugere ações básicas de posicionamento e ataque do time.

```
private void acaoAtacante(long nextIteration) {
      double xInit=-15, yInit=0;
      EFieldSide side = selfPerc.getSide();
      Vector2D initPos = new Vector2D(xInit*side.value(), yInit);
      Vector2D goalPos = new Vector2D(50*side.value(), 0);
      Vector2D ballPos;
      PlayerPerception pTemp;
      while (true) {
            updatePerceptions();
            ballPos = fieldPerc.getBall().getPosition();

            switch (matchPerc.getState()) {
            case BEFORE_KICK_OFF:
                  commander.doMoveBlocking(xInit, yInit);
                  break;
            case PLAY_ON:
                  if(isPointsAreClose(selfPerc.getPosition(),
                                                ballPos, 1)){
                        if(isPointsAreClose(ballPos, goalPos, 30)){
                              // chuta para o gol
                              kickToPoint(goalPos, 100);
                        }else{
                              // conduz para o gol
                              kickToPoint(goalPos, 25);
                        }
                  }else{
                        pTemp = getClosestPlayerPoint(ballPos,
                                                      side, 3);
                        if(pTemp != null &&
                        pTemp.getUniformNumber() == selfPerc
                                          .getUniformNumber()){
                              // pega a bola
                              dash(ballPos);
                        }else if(!isPointsAreClose(selfPerc
                                    .getPosition(),initPos, 3)){
                              // recua
                              dash(initPos);
```

```
                    }else{
                        // olha para a bola
                        turnToPoint(ballPos);
                    }
                }
                break;

            /* Todos os estados da partida */

            default:
                break;
            }
        }
}
```

Do mesmo modo implementado nos métodos anteriores, setamos as variáveis *xInit* e *yInit* como coordenadas da posição inicial para nosso atacante. A novidade é a variável *goalPos* que armazena a posição objetivo, ou seja, o gol do adversário - note que usamos mais uma vez a variável *side* para calcular dinamicamente a barra do adversário em relação ao lado do campo.

Seguindo o raciocínio, implementamos em ordem de prioridade, as ações do atacante. Primeiramente o caso de quando estiver com a bola, verificamos se a distância do atacante para o gol do adversário é o suficiente para *chutar para o gol*, caso não seja, o atacante *conduzirá a bola* em direção ao objetivo até que a condição anterior seja satisfeita. Logo após, verificamos se o jogador é o mais próximo da bola para evitar que o mesmo fique seguindo-a sempre, e se for, *pegue a bola*. O próximo caso é saber se ele está fora de sua posição para executar o comando de *recuar* para seu posto. Por fim, se nenhum dos casos anteriores forem satisfeitos, apenas *olhe para a bola*.

Este é um exemplo básico do que se consegue fazer com a biblioteca, setar posicionamento completo do time, construir jogadas, tudo de maneira independente, ou seja, cada jogador (agente) executando comandos próprios.

Para testar a implementação do time de exemplo, execute a classe *Main*, veja a formação do time e interaja iniciando a partida ou soltando a bola perto deles (figuras 2.3.A e 2.3.B).

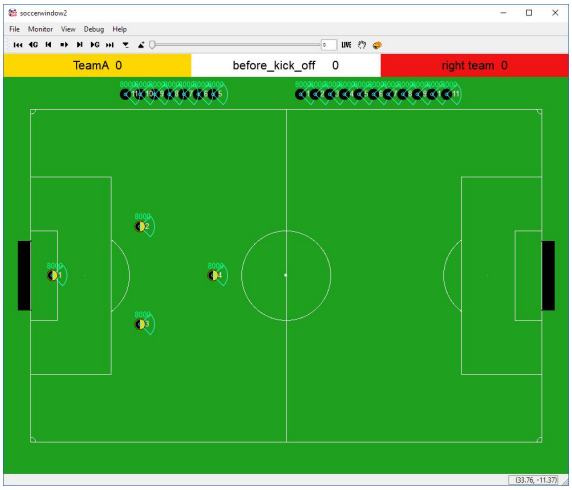

FIGURA 2.3.A

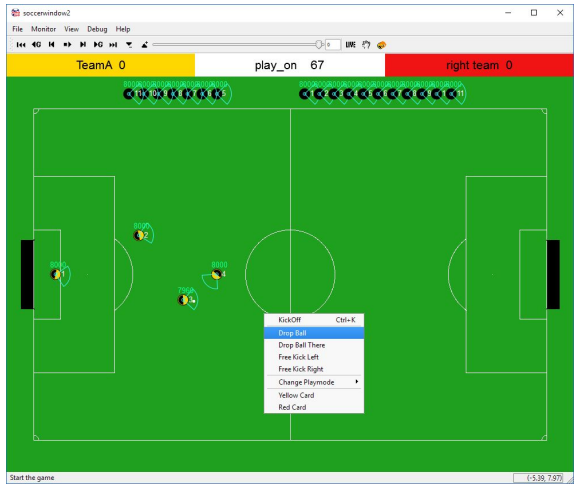

FIGURA 2.3.B

3. Percepções

Nesse tópico iremos expor detalhes sobre as percepções que nossa biblioteca possuem, percepções do jogador em relação ao campo, aos outros agentes, a bola e a si próprio.

3.1. Percepção do Jogador

A classe *PlayerPerception* é responsável pelos detalhes de percepção do jogador sobre o mesmo, essa classe estende a classe *ObjectPerception*.

3.1.1. Posição

A função *getPosition* retorna a posição em Vector2D do jogador atual.

```
@method Vector2D getPosition()
@return Vector2D
@example Vector2D position = playerPerception.getPosition()
```

3.1.2. Direção

Para saber a direção que o usuário está olhando a biblioteca possui a função *getDirection*.

```
@method Vector2D getDirection()
@return Vector2D
@example Vector2D direction = pp.getDirection()
```

3.1.3. Outros Métodos

Detalhes	Retorno	Método
Retorna o nome do time do jogador	String	getTeam()
Retorna o nome do uniforme do jogador	int	getUniformNumber()
Retorna se o jogador é goleiro	boolean	isGoalie()
Retorna o lado do campo do jogador	EFieldSide	getSide()
Retorna o estado do jogador	EPlayerState	getState()

3.2. Percepções de Campo

A classe *FieldPerception* é responsável pelas percepções de campo do jogador.

3.2.1. Posição dos Jogadores

A função *getAllPlayers* retorna todos os jogadores que estão na visão do usuário.

```
@method ArrayList<PlayerPerception> getAllPlayers()
@return ArrayList<PlayerPerception>
@example ArrayList<PlayerPerception> players = fp.getAllPlayers()
```

A função *getTeamPlayers* retorna os jogadores dependendo time que estão na visão do usuário.

```
@params side:EFieldSide
@method ArrayList<PlayerPerception> getTeamPlayers(EFieldSide)
@return ArrayList<PlayerPerception>
@example ArrayList<PlayerPerception> players =
                          fp.getTeamPlayers(EFieldSide.LEFT)
```

3.2.2. Posição da Bola

A função getBall retorna a percepção da bola tendo em vista que ela se encontre na visão do usuário.

```
@method ObjectPerception getBall()
@return ObjectPerception
@example ObjectPerception ball = fp.getBall()
```

3.2.3. Outros Métodos

Detalhes	Retorno	Método
Retorna jogador pelo lado e uniforme	PlayerPerception	getTeamPlayer(EFieldSide, int)

4. Movimentação do Jogador

Neste tópico será tratado tudo que se trata de como movimentar os jogadores utilizando a classe *PlayerCommander*, todos esses comandos são abstrações dos comando mandados diretamente ao servidor.

O servidor possui uma janela de tempo que tem por objetivo dar um tempo do envio de um comando para outro em sequência, comandos que forem enviados nesse meio de tempo não irão acontecer. Para evitar que o usuário da nossa lib tenha dificuldades com isso, foi criado métodos auxiliares em todas as movimentações dos jogadores com o sufixo *Blocking*, estes métodos dão uma maior confiança que o comando do jogador vai ser executado por esperarem o mesmo tempo do intervalo feito pelo servidor. O usuário não é obrigado a usar, possuindo os métodos sem esse contexto, porém é mais um auxílio aos usuários da biblioteca.

4.1. Giro

Giros são fundamentais para se ter uma percepção de todo o campo e mudar a rota de movimentação durante uma partida, abaixo constam algumas formas diferentes de executar um giro num jogador.

<u>Ângulo</u>

Tem como objetivo girar um jogador para um determinado ângulo.

```
@params degreeAngle:double
@method boolean doTurn(double)
@return boolean
@example boolean status = pc.doTurn(90)
```

```
@params degreeAngle:double
@method void doTurnBlocking(double)
@example pc.doTurnBlocking(90)
```

Direção

Tem como objetivo girar um jogador para um determinado vetor da sua direção.

```
@params orientarion:Vector2D
@method boolean doTurnToDirection(Vector2D)
@return boolean
@example boolean status = pc.doTurnToDirection(new Vector2D(5,5))
```

```
@params orientarion:Vector2D
@method void doTurnToDirectionBlocking(Vector2D)
@example pc.doTurnToDirectionBlocking(new Vector2D(5,5))
```

Ponto

Tem como objetivo girar um jogador para um determinado ponto do campo.

```
@params referencePoint:Vector2D
@method boolean doTurnToPoint(Vector2D)
@return boolean
@example boolean status = pc.doTurnToDirection(new Vector2D(5,5))
```

```
@params referencePoint:Vector2D
@method void doTurnToPointBlocking(Vector2D)
@example pc.doTurnToDirectionBlocking(new Vector2D(5,5))
```

4.2. Correr

O correr é um ponto base de uma partida de futebol, para isso é necessário saber a intensidade que o jogador vai correr.

Correr simples

```
@params intensity:double
@method boolean doDash(double)
@return boolean
@example boolean status = pc.doDash(10)
```

```
@params intensity:double
@method void doDashBlocking(double)
@example pc.doDashBlocking(10)
```

Girar e Correr

```
@params orientarion:Vector2D
@params intensity:double
@method void doDashBlocking(Vector2D,double)
@example pc.doDashBlocking(new Vector2D(5,5), 10)
```

4.3. Chutar

Para o jogador chutar é necessário que se tenha a intensidade de chute a direção que o chute deve acontecer, tendo em vista que essa direção será em relação a direção do usuário.

```
@params intensity:double
@params direction:Vector2D
@method boolean doKick(double,Vector2D)
@return boolean
@example boolean status = pc.doKick(10,new Vector2D(5,5))
```

```
@params intensity:double
@params direction:Vector2D
@method void doKickBlocking(double,Vector2D)
@example pc.doKickBlocking(10,new Vector2D(5,5))
```

4.4. Mover

Esse método só é usado quando o jogo está paralisado, por exemplo antes do início do jogo ou em faltas, escanteios e laterais.

```
@params intensity:double
@params direction:Vector2D
@method boolean doKick(double,Vector2D)
@return boolean
@example boolean status = pc.doKick(10,new Vector2D(5,5))
```

```
@params intensity:double
@params direction:Vector2D
@method void doKickBlocking(double,Vector2D)
@example pc.doKickBlocking(10,new Vector2D(5,5))
```

4.5. Defender

Método só utilizado pelo goleiro, responsável por fazer as defesas que estejam num raio dele. O ângulo de defesa deve ser entre -45 e 45.

```
@params angle:double
@method boolean doCatch(double)
@return boolean
@example boolean status = pc.doCatch(-45)
```

```
@params angle:double
@method void doCatchBlocking(double)
@example pc.doCatchBlocking(45)
```

5. Comandando Jogador Manualmente

A RoboSoccerLib oferece a opção de se executar ações em um jogador manualmente, com o auxílio da classe *CommandPlayer* o usuário poderá ver todas as funcionalidades descritas no ponto 4 utilizando comandos que serão interpretados para gerar ações em um jogador. Ao instanciar a classe será aberto uma caixa onde o usuário poderá digitar o comando necessário. Abaixo tem uma tabela apresentando todos os comandos relacionando eles com os métodos já citados e outros comandos para possibilidade de debugar funcionalidades.

Comando	Sigla	Parâmetros	Exemplo
Giro	T	(angle:double)	T 90
Giro para direção	TD	(x:double) (y:double)	T 0 1
Giro para ponto	TTP	(x:double) (y:double)	TTP -10 5
GIro para bola	TTB	N/A	TTB
Mover	M	(x:double) (y:double)	M -25 0
Chutar para ponto	KTP	(x:double) (y:double)	KTP -25 0
Correr para ponto	RTP	(x:double) (y:double) (erro:double)	RTP -10 5 3
Informações do Jogador	SP	N/A	SP
Posição do Jogador	SPPOS	N/A	SPPOS
Direção do Jogador	SPDIR	N/A	SPDIR
Nome do Time	SPTEA	N/A	SPTEA
Uniforme do Jogador	SPUNI	N/A	SPUNI
Informações do Time	STIME	N/A	STIME
Quantidade de Jogadores	SQPL	N/A	SQPL
Informações do Jogador	SPLIN	(lado:string) (numero:int)	SPLIN
Quantidade de Jogadores por Time	SQPSS	(lado:string)	SQPSS
Posição da bola	SBPOS	N/A	SBPOS

6. Times Adversários

Para que o usuário da RobboSoccerLib possa testar o seu time em execução, é oferecido exemplos de agentes que foram disponibilizados pela Universidade de Carleton. Esses agentes se encontram na classe *AbstractTeam* e existe a possibilidade de os mesmos jogarem entre si, abaixo temos como funciona cada um deles.

6.1. Krislet

O Krislet é uma agente simples que persegue a bola e chuta para o gol adversário.

```
@method void launchKrislet()
@example abstractTeam.launchKrislet()
```

6.2. Tracker

O Tracker, que é baseado no Krislet, é uma agente simples que persegue a bola em torno do campo.

```
@method void launchTracker()
@example abstractTeam.launchTracker()
```

6.3. A1

O A1 também é baseado no Krislet, porém esse agente não vai seguir a bola se ver outro agente do seu time mais próximo.

```
@method void launchA1()
@example abstractTeam.launchA1()
```

6.4. NoSwarm

O NoSwarm é um agente que possui mais características estratégicas.

```
@method void noSwarm()
@example abstractTeam.noSwarm()
```