Sumário

Lista de Figuras

Capítulo 1

Introdução

O Robocode [Robocode 2013] é um jogo de simulação programável escrito em Java ou .NET e tem como objetivo é codificar um robô virtual, inicialmente configurado como tanque de guerra, para competir contra outros em um campo de batalha.

O jogador é o programador do robô, moldando-o e atribuindo comportamentos e comandos de interação em uma batalha utilizando técnicas de Inteligência Artificial (IA) e programação. Este trabalho tem como objetivo descrever uma visão mais detalhada da plataforma de simulação Robocode, utilizando a linguagem de programação Java nos códigos implementados. A ideia principal é a demonstração prática do funcionamento do Robocode na forma de tutorial.

Apesar de o Robocode parecer um ambiente simples nos primeiros contatos, o mesmo pode ser usado também para aplicações mais complexas em áreas acadêmicas e científicas. Dentre as possíveis aplicações, pode-se citar o ensino-aprendizagem de IA na elaboração de novas estratégias dos robôs usando Redes Neurais (RNA), Algorítimos Genéticos (AG), Sistemas Multi-Robóticos (SMRs), entre outros.

[Nielsen e Jensen 2010] destacam o uso de AG e RNA na implementação de novas estratégias destinadas às ações e comportamentos dos tanques durante as batalhas. A ideia é provar que esses métodos podem ser implementados com êxito na tarefa de apontar o canhão de um tanque em direção ao adversário. Neste caso, deve-se utilizar o histórico de informações dos oponentes, tais como velocidade,

energia, entre outros.

[Hong e Cho 2004] propõem o uso de AG na criação de comportamentos emergentes, combinando ações primitivas de um robô na batalha. Os comportamentos dos robôs, são separados em cinco subcomportamentos nomeados como movimento, pontaria, seleção de alvos (target selection), busca por oponentes usando o radar (radar search) e gerenciamento de energia. Cada gene do genoma do AG representa um subcomportamento, desenvolvendo um conjunto finito de ações para representar os genes, em que a sua evolução é feita por seleção natural, mutação e criação por cruzamento.

O Robocode pode ser aplicado também para aprendizado de máquina, subárea da IA focada em raciocínio indutivo do qual se extrai regras e padrões de um grande conjunto de dados coletados ao longo de um período para aperfeiçoar algum tipo de habilidade ou tarefa. Como exemplo pode-se citar o projeto de [Gade et al. 2003], que utilizam o Robocode para desenvolver e implementar o Aalbot. Esse robô é capaz de aprender com a experiência, permitindo aprimorar suas habilidades gradualmente ao longo do tempo. Além disso, é proposto a implementação da adaptabilidade no robô, tornando-o capaz de agir e reagir conforme as propriedades do ambiente. Para atacar o oponente é necessário achá-lo usando o radar. Uma vez localizado, deve-se executar ações que expressam seus comportamentos e estratégias de ataque, tal como rotacionar até 360 graus seu canhão de acordo com a posição do inimigo. Entretanto, essa sequência de ações induz a um defasamento de tempo. O problema então é o tempo entre a descoberta do inimigo em campo (procurar), a rotação da torre (rotacionar) e o tiro (atirar).

A estratégia de mover o radar e a arma de maneira sincronizada foi utilizada por [Uchida, Kobayashi e Watanabe 2003], de forma a economizar o tempo de ocorrência dessas ações. Vale ressaltar que os componentes dos robôs se movem de maneira independente como padrão do Robocode. [Uchida, Kobayashi e Watanabe 2003] descrevem ainda a implementação dos agentes SENKEI e SENKEI2, os quais participaram do Robocode Japan Cup 2002. O SENKEI tem como objetivo atirar baseando-se na previsão do movimento do oponente, enquanto que o SENKEI2 prevê o tiro do oponente através da quantidade alterada de energia perdida do adversário e da posição do projétil usando o método de predição linear. Dessa forma, conhecendo esses dois fatores, o SENKEI2 pode escapar, movendo-se de modo a evitar o dano causado por um tiro. Em uma das simulações, configurando o SENKEI2 e SENKEI para uma disputa, observou-se que o vencedor foi o primeiro, pois foi programado para atuar de maneira mais estratégica, baseando o seu movimento para evitar projéteis.

Na área de SMR, a essência está relacionada à interação de dois ou mais robôs de forma a atingir um único objetivo. O Robocode pode ser utilizado na simulação e análise comportamental de robôs. [Jannace 2011] utilizou o Robocode para simular batalhas entre dois times: unificado e não-unificado. O primeiro diz respeito ao time de robôs que trabalharam de maneira cooperada para derrotar o outro time. O segundo refere-se a uma coleção de robôs que agem de forma independente contra os robôs adversários em campo. A partir disto, foi demonstrado que o uso de um time unificado tem melhor desempenho em campo e mais facilidade para vencer um time não-unificado.

O Capítulo 2 deste manual envolverá o processo de instalação do Robocode e sua integração com o NetBeans, demonstrando passo-a-passo até que se possa executar a tela inicial do Robocode. O Capítulo 3 apresenta uma visão geral da plataforma, contando com tópicos como a anatomia de um robô, suas propriedades físicas, regras do campo de batalha e pontuação e os recursos da tela de batalha. O Capítulo 4 promoverá o contato do leitor com o desenvolvimento de robôs no Robocode através da implementação de algoritmos voltados para um propósito específico, desde os comandos básicos de movimentação de um robô até a implementação de um time.

Capítulo 2

Instalação do Robocode e Integração com o NetBeans

O Robocode provém de um editor nativo de programação Java para criação e modelagem de robôs. Por sua simplicidade, não possui a robustez se comparado com outros editores mais avançados, as quais incluem ferramentas para gerenciar uma aplicação Java, tais como versionamento e depuração de código. Sendo assim, é possível integrar o Robocode com as Integrated Development Environments (IDEs) Eclipse [Eclipse 2014] ou NetBeans [Oracle 2014] para compilação e execução do código automaticamente no Robocode.

É importante destacar que este tutorial destina-se à instalação efetuada no sistema operacional Windows. Para se obter êxito na instalação do Robocode, é essencial que o Java SE [Oracle 2014] esteja instalado no computador com versões a partir do Java 5. A Oracle possui uma ferramenta online[1] para verificar se o Java está instalado ou não na máquina. Os procedimentos necessários para correta instalação do Robocode e integração com o NetBeans serão abordados a seguir:

Passo 1: O primeiro passo consiste em efetuar o download[2] do Robocode, baixando a última versão da plataforma. O arquivo a ser baixado será o de extensão .jar, explicitamente nomeado como robocode-versão-setup.jar, no qual versão será a mais recente;

Passo 2: Instalar o arquivo na raiz “C:” do Windows;

Passo 3: Ao abrir o NetBeans, deve-se criar um novo projeto para ser vinculado ao Robocode. Neste projeto é necessário adicionar todos os pacotes, classes e demais arquivos relacionados ao robô que será modelado. Para isso, a Figura 2.1 deve ser visualizada no NetBeans selecionando o menu Arquivo > Novo Projeto. A categoria Java e o projeto aplicação Java serão selecionados nos painéis (1) e (2), respectivamente, antes de clicar no botão Próximo (3);

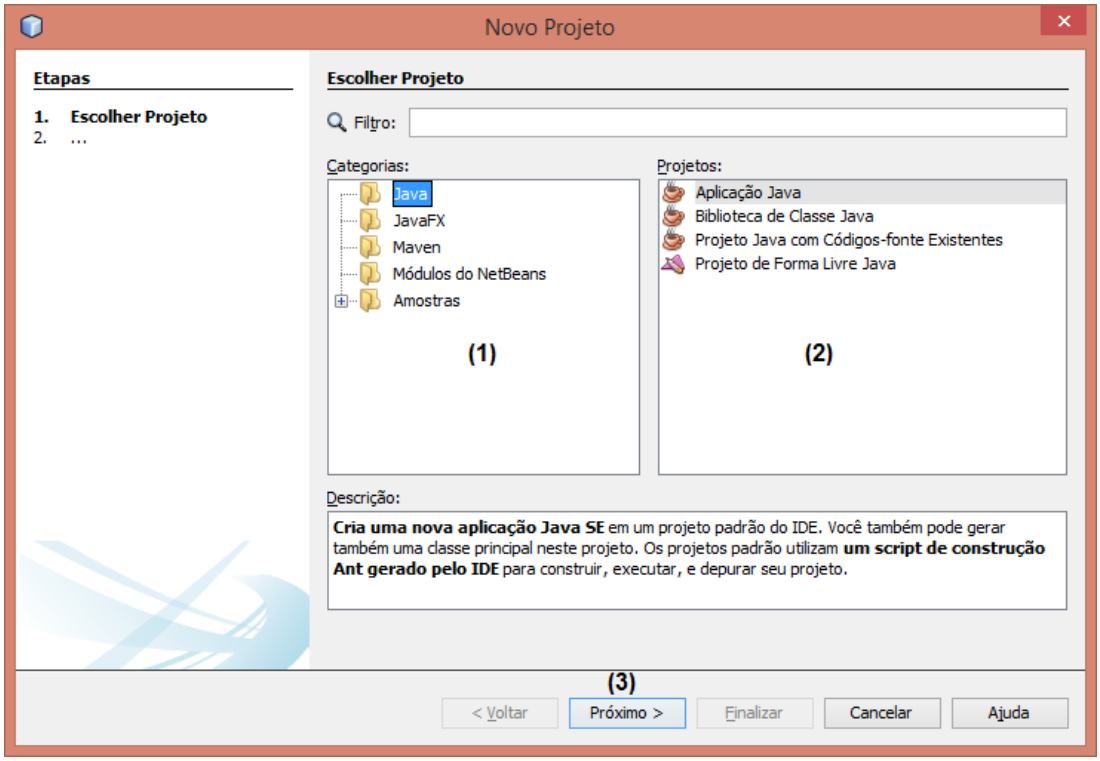

Figura 2.1: Tela de Configuração do Tipo de Projeto a ser Criado no NetBeans.

Passo 4: Na nova janela da Figura 2.2, alguns parâmetros iniciais do novo projeto devem ser configurados. O Nome do Projeto, Local de Armazenamento, Nome para uma nova pasta e o Nome da Classe devem ser definidos em (1), (2), (3) e (4), respectivamente. Em (4), o NetBeans define automaticamente o nomedopacote.NomeDaClasse, no qual o nome do pacote é geralmente o mesmo que o nome dado

ao projeto. Clicando em (5), o usuário confirma as configurações definidas neste passo;

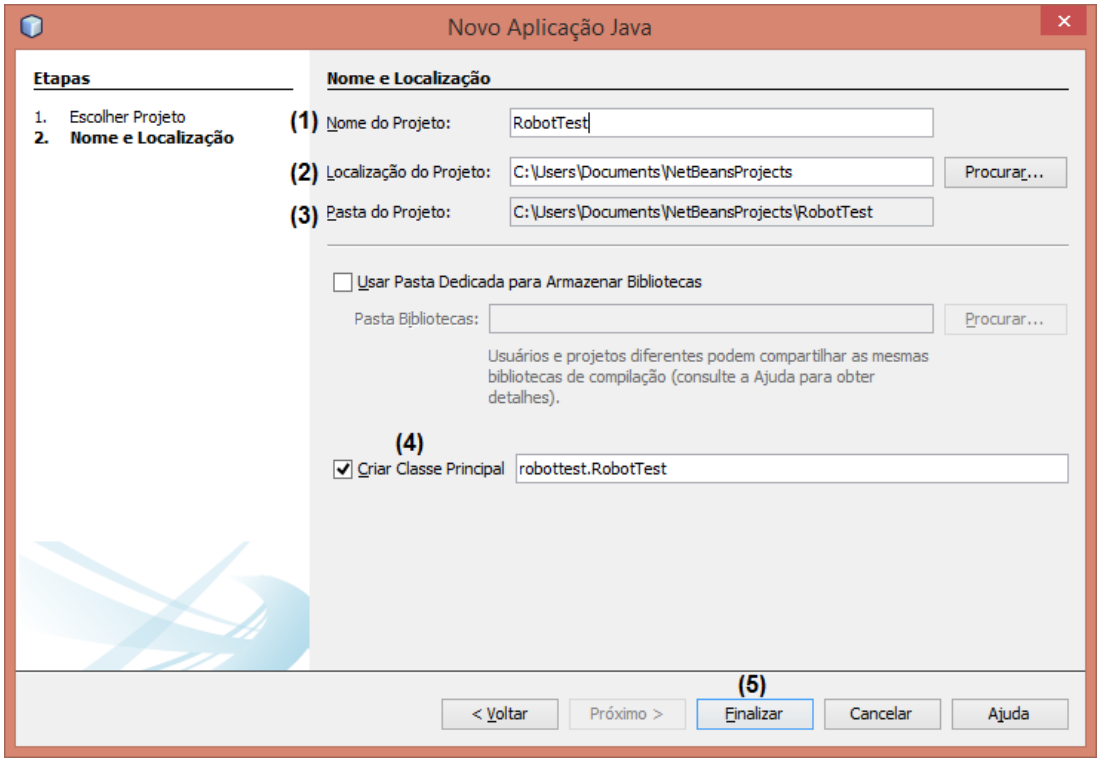

Figura 2.2: Tela para Configuração de Parâmetros Iniciais do Novo Projeto a ser Criado no NetBeans.

Passo 5: Para executar um projeto do Robocode a partir do NetBeans, deve-se adicionar a biblioteca robocode.jar, que se encontra na pasta C:\robocode\libs, após a descompactação do arquivo baixado no Passo 1 e instalado no Passo 2. Para adicionar o arquivo, deve-se selecionar Ferramentas > Bibliotecas > Adicionar JAR/Pasta... no NetBeans, assim como indicado em (1) na Figura 2.3. A seguir, uma janela do Windows será mostrada para que o aquivo .jar seja selecionado. Após selecionar, os diretórios do arquivo da biblioteca estará indicado no campo Classpath da Biblioteca em (2) da Figura 2.3. Logo após, deve-se clicar em (3) para confirmação das configurações;

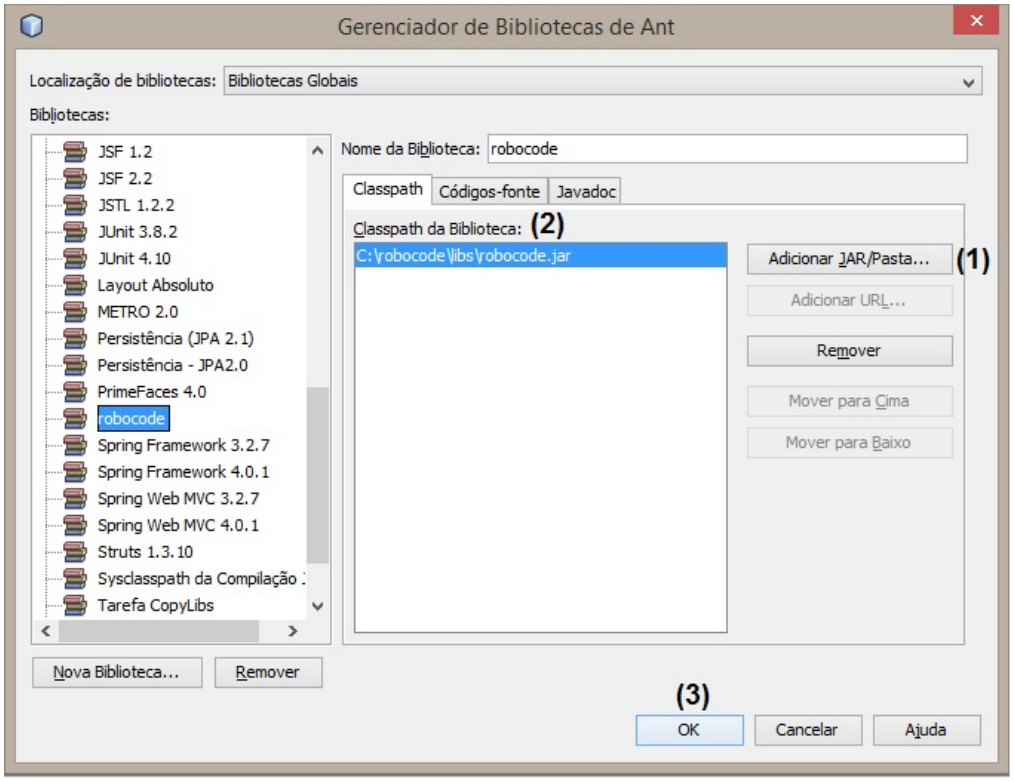

Figura 2.3: Tela para Inclusão da Biblioteca Robocode no NetBeans.

Passo 6: Ao criar um projeto no NetBeans, deve-se adicionar os arquivos que estão na pasta lib do Robocode. Para isso, é necessário clicar na pasta do projeto criado no NetBeans com o botão direito do mouse e selecionar Propriedades > Bibliotecas > Adicionar Biblioteca.... A janela ilustrada na Figura 2.4 será mostrada e os arquivos descritos a seguir, que correspondem aos módulos do Robocode, deverão ser adicionados clicando no botão (1) relacionado à ação Adicionar Biblioteca...:

1. codesize-versao.jar: utilitário Java responsável por retornar o tamanho do código, mensurado em bytes, de arquivos .class. Mais detalhes acerca da propriedade Code Size serão abordados na Seção 3.7;
2. robocode.core-versao.jar: é considerado o coração da plataforma e tem como função carregar todos os módulos que constituem o Robocode. Um módulo é qualquer arquivo .jar em um diretório da biblioteca, cujo nome possui o prefixo “robocode”;
3. picocontainer-versao.jar: introduzida na versão 1.7.0.0 do Robocode, trata-se de uma biblioteca open-source que integra o padrão de desenvolvimento de injeção de depedência. Mais informações acerca do PicoContainer podem ser consultadas no site do projeto[3] ;
4. robocode.battle-versao.jar: módulo responsável por integrar as regras e recursos gerais da batalha, tais como as rodadas, turnos e etapas presentes em cada turno. Detalhes sobre como uma batalha funciona será apresentado no Capítulo 3;
5. robocode.host-versao.jar: módulo que gerencia a segurança dos robôs;
6. robocode.jar: biblioteca principal do Robocode responsável por integrar todas as classes nativas de codificação de robôs e funcionamento do Robocode;

7. robocode.repository-versao.jar: relacionado ao banco de dados de robôs instalado no sistema. Nesse caso, as classes dos robôs são carregadas e registradas quando o jogo as identifica dentro dos seus diretórios. Assim, o metadata gerado é armazenado nessa base de dados, que é serializado em um arquivo. O objetivo é promover maior eficiência, uma vez que a leitura em um banco de dados é mais rápida do que carregar todas as classes e examiná-la;
8. robocode.ui-versao.jar: integra a interface de usuário da plataforma relacionada ao campo de batalha e as janelas para configurações e menus;
9. robocode.ui.editor-versao.jar: integra a interface vinculada ao editor de codificação do Robocode, geralmente destinado a principiantes. Esse editor pode ser usado quando a plataforma não for integrada com o NetBeans ou Eclipse;
10. roborumble-versao.jar: módulo responsável por executar as batalhas e atualizar os resultados de energia e pontuação final a cada rodada do jogo.

É importante destacar que o termo versão dos arquivos .jar está vinculado à versão instalada do Robocode e varia ao longo do tempo. Após esses passos, deve-se clicar no botão (2) da Figura 2.4.

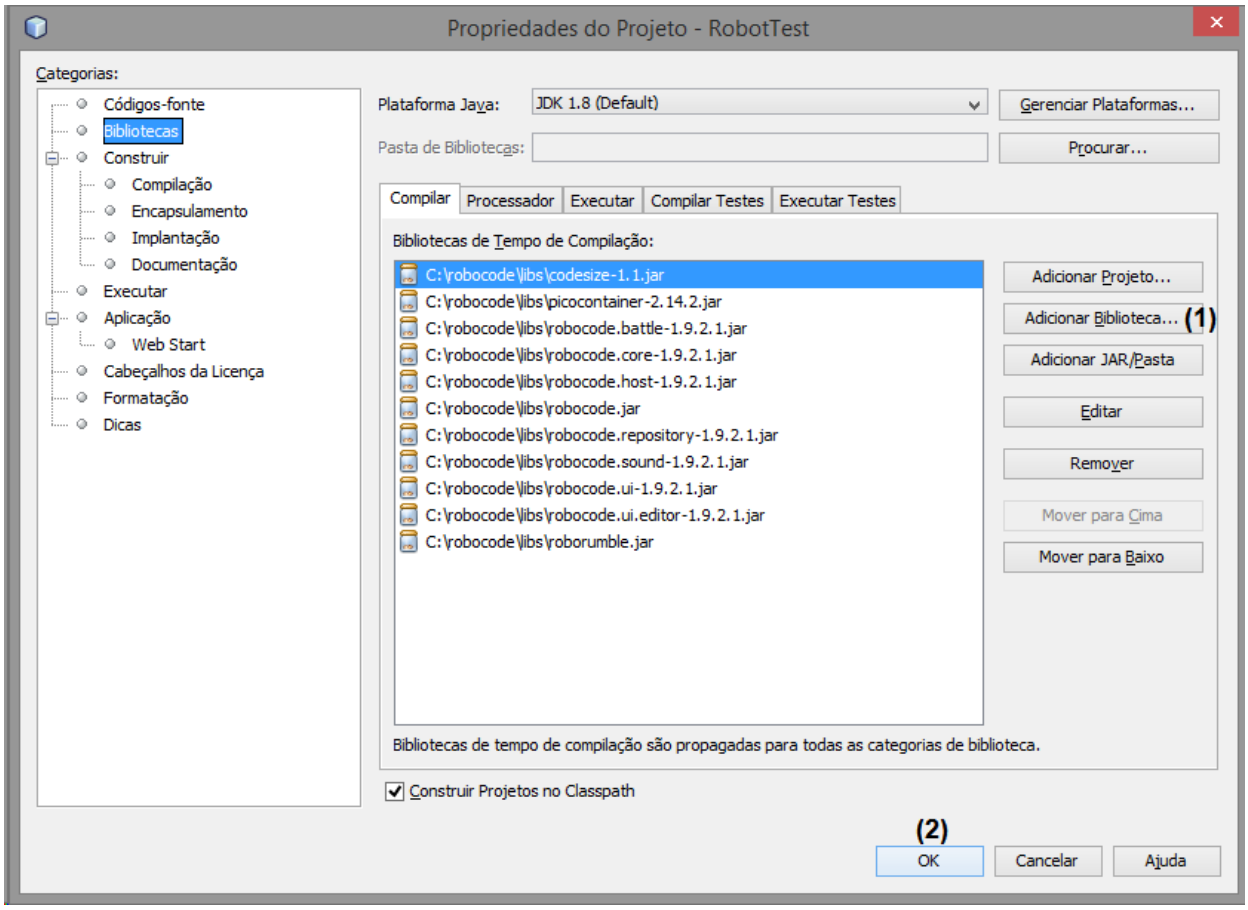

Figura 2.4: Tela para Inclusão da Biblioteca Robocode no Projeto que está sendo Criado.

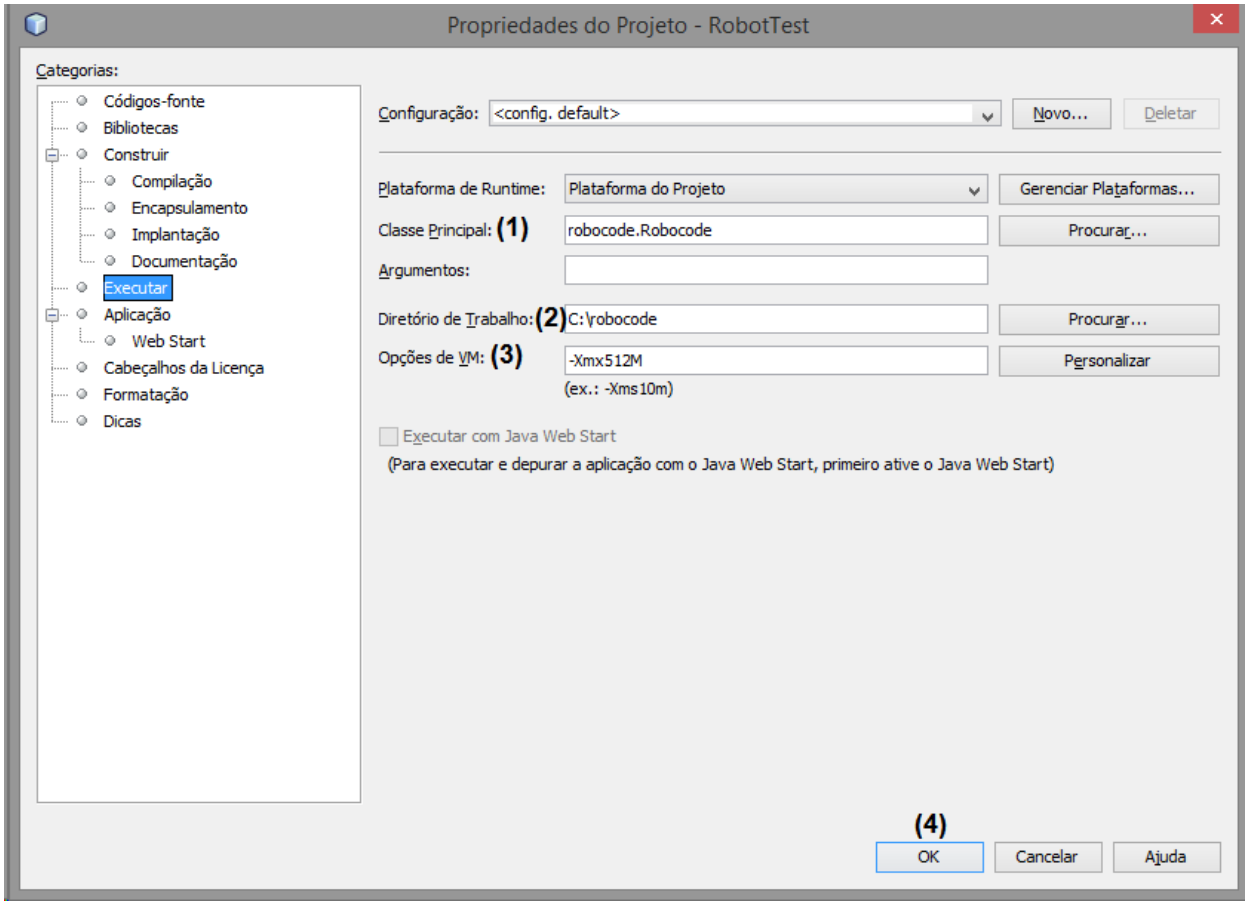

Figura 2.5: Tela para Alteração dos Parâmetros de Execução do Projeto para Chamada Automática do Robocode.

Passo 7: Neste passo, o projeto criado no NetBeans será configurado para ser executado no Robocode. Para isso, basta clicar na pasta do projeto criado no NetBeans com o botão direito do mouse e selecionar Propriedades > Executar. A janela da Figura 2.5 será mostrada e os parâmetros Classe Principal, Diretório de Trabalho (raiz na qual o software foi instalado) e Opções de VM devem ser definidos como em robocode.Robocode (1), C:\robocode (2) e -Xmx512M (3), respectivamente;

Passo 8: Ao clicar em OK (4) na Figura 2.5, o NetBeans estará devidamente configurado e, ao executar o projeto vinculado a essa configuração, através da tecla de atalho “F6” ou em Executar > Projeto, o Robocode deverá ser aberto automaticamente, com a tela inicial da Figura 2.6;

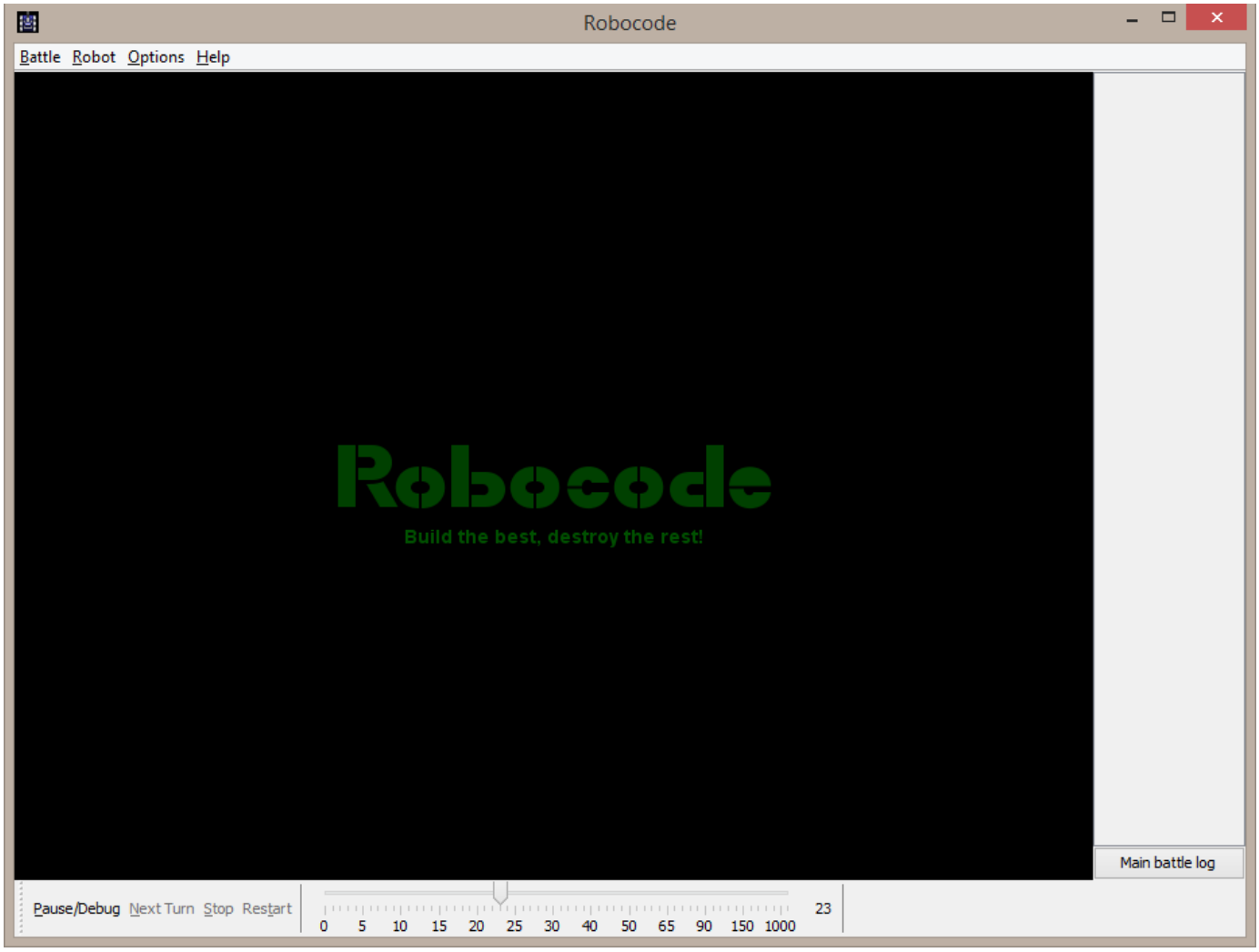

Figura 2.6: Janela da Plataforma Robocode, Iniciado Automaticamente Via NetBeans.

Passo 9: Para que o Robocode reconheça as classes relacionadas aos robôs devidamente codificados no NetBeans, os procedimentos a seguir devem ser executados:

1. Na interface do Robocode da Figura 2.6, deve-se selecionar o menu Options > Preferences > Guia Development Options. A origem das classes relacionadas aos robôs codificados no NetBeans devem ser adicionadas através da janela ilustrada na Figura 2.7, clicando em Add > Seleção da Pasta build > classes do projeto (1) e logo após em Finish (2);
2. Após os procedimentos realizados no passo anterior, ao executar o projeto no NetBeans, o Robocode abrirá automaticamente, podendo-se também acessar os robôs recentemente codificados na IDE e adicionando-os no campo de batalha. Neste caso, basta selecionar as opções Battles > New da Figura 2.6 para que a janela da Figura 2.8 seja aberta. Na guia Robots em (1), deve-se verificar se o pacote do projeto é mostrado no painel Packages (4) e os respectivos robôs em (5). Caso contrário, os Passos 7 e 9 não foram executados devidamente.

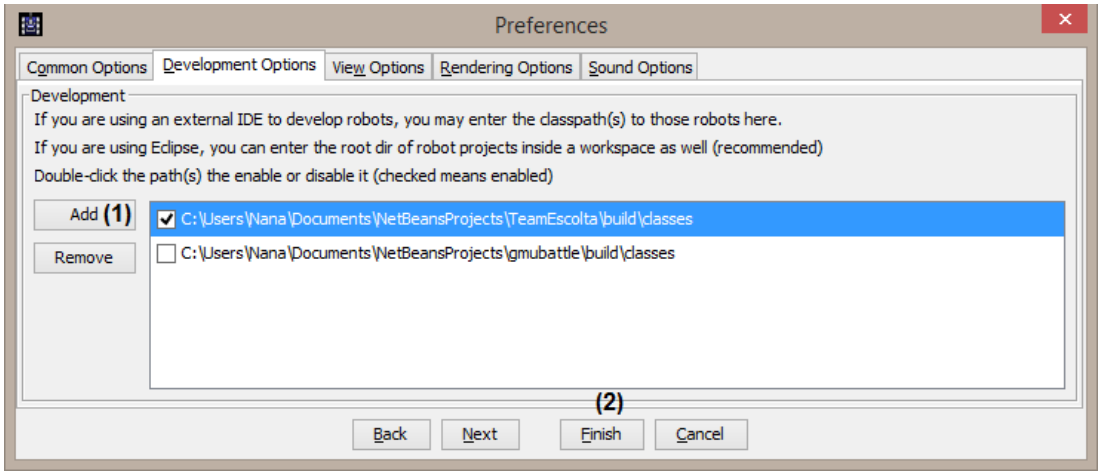

Figura 2.7: Guia Development Options da Janela Preferences do Robocode.

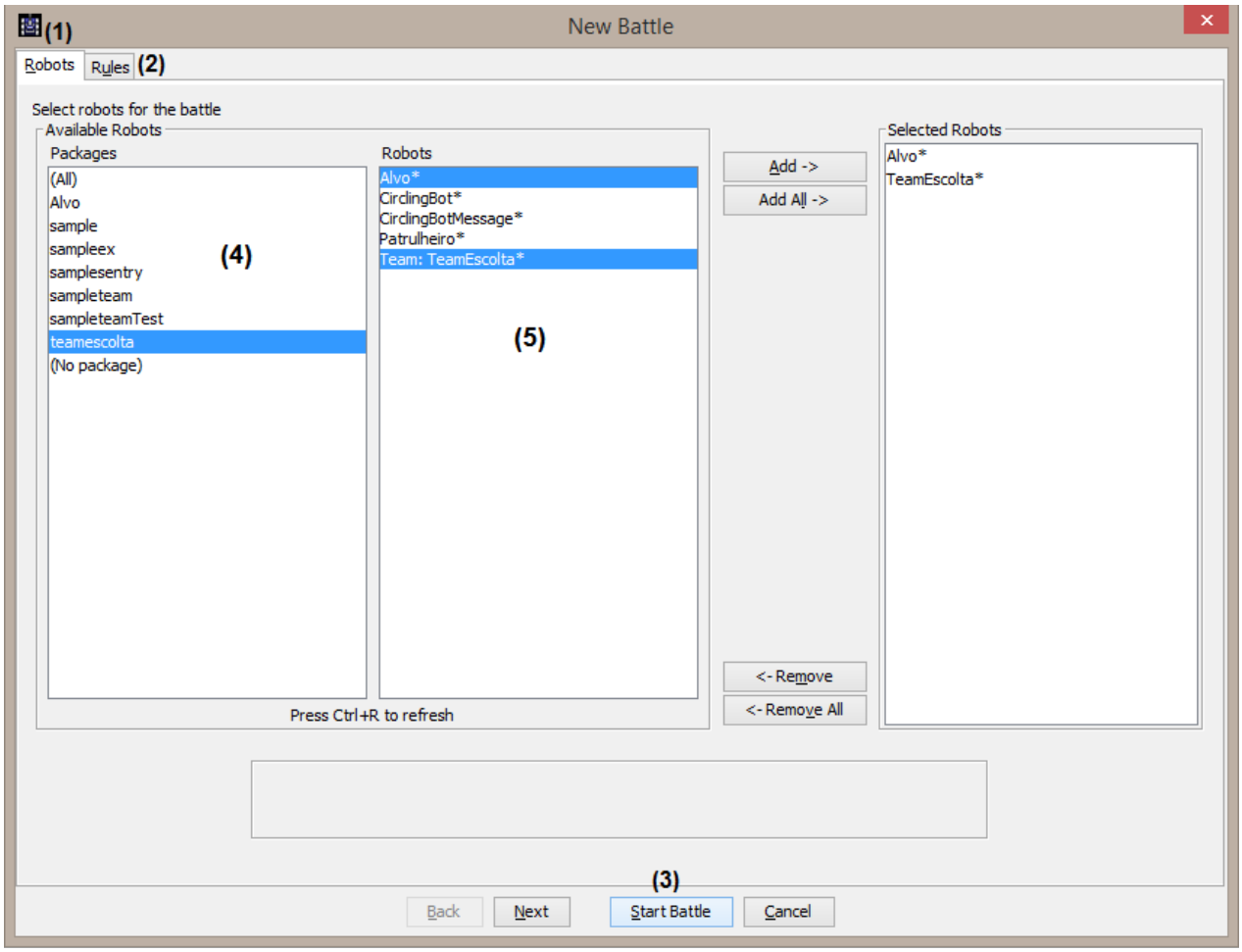

Figura 2.8: Janela de Configuração Inicial de Batalha do Robocode.

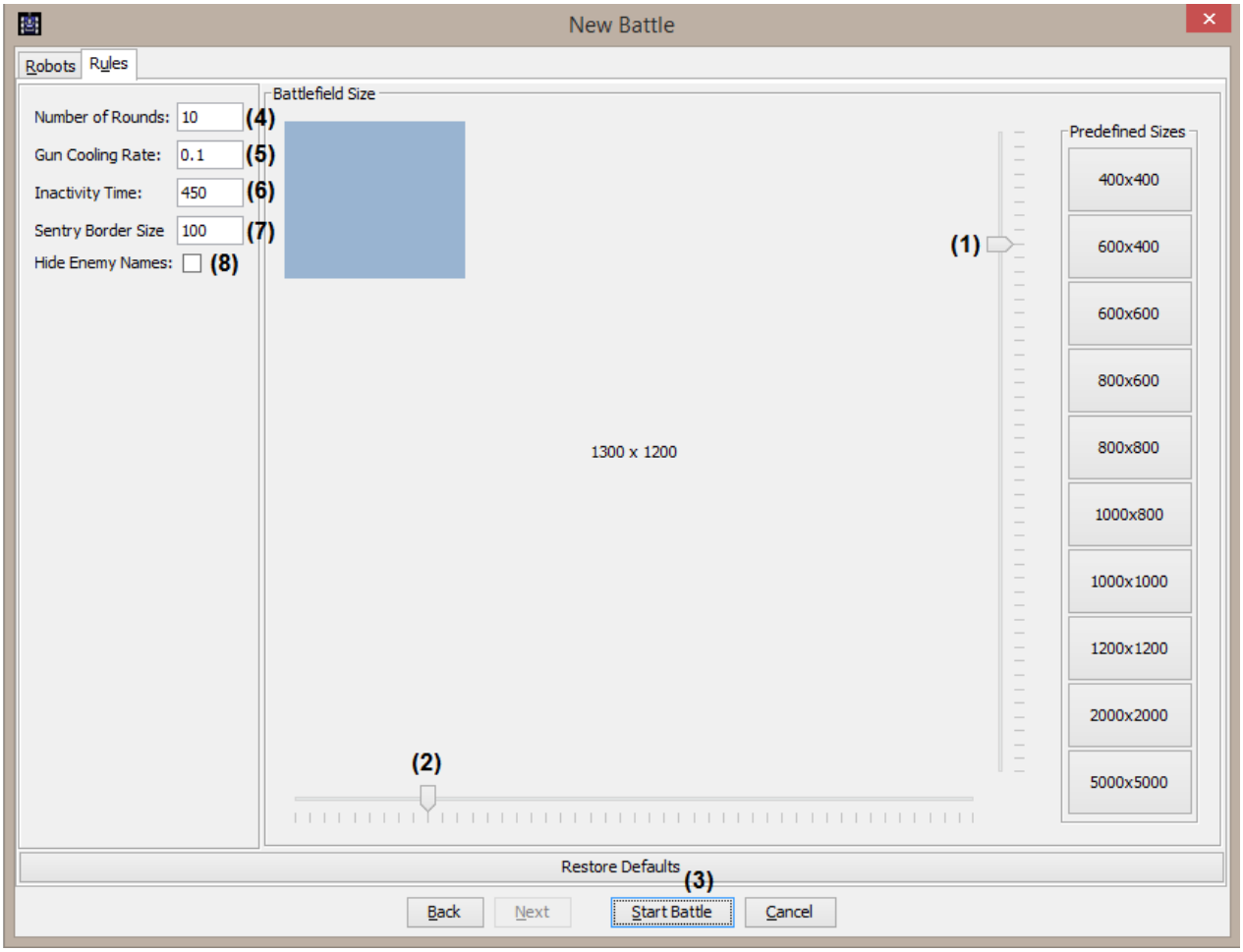

Figura 2.9: Janela de Configuração Inicial de Batalha do Robocode.

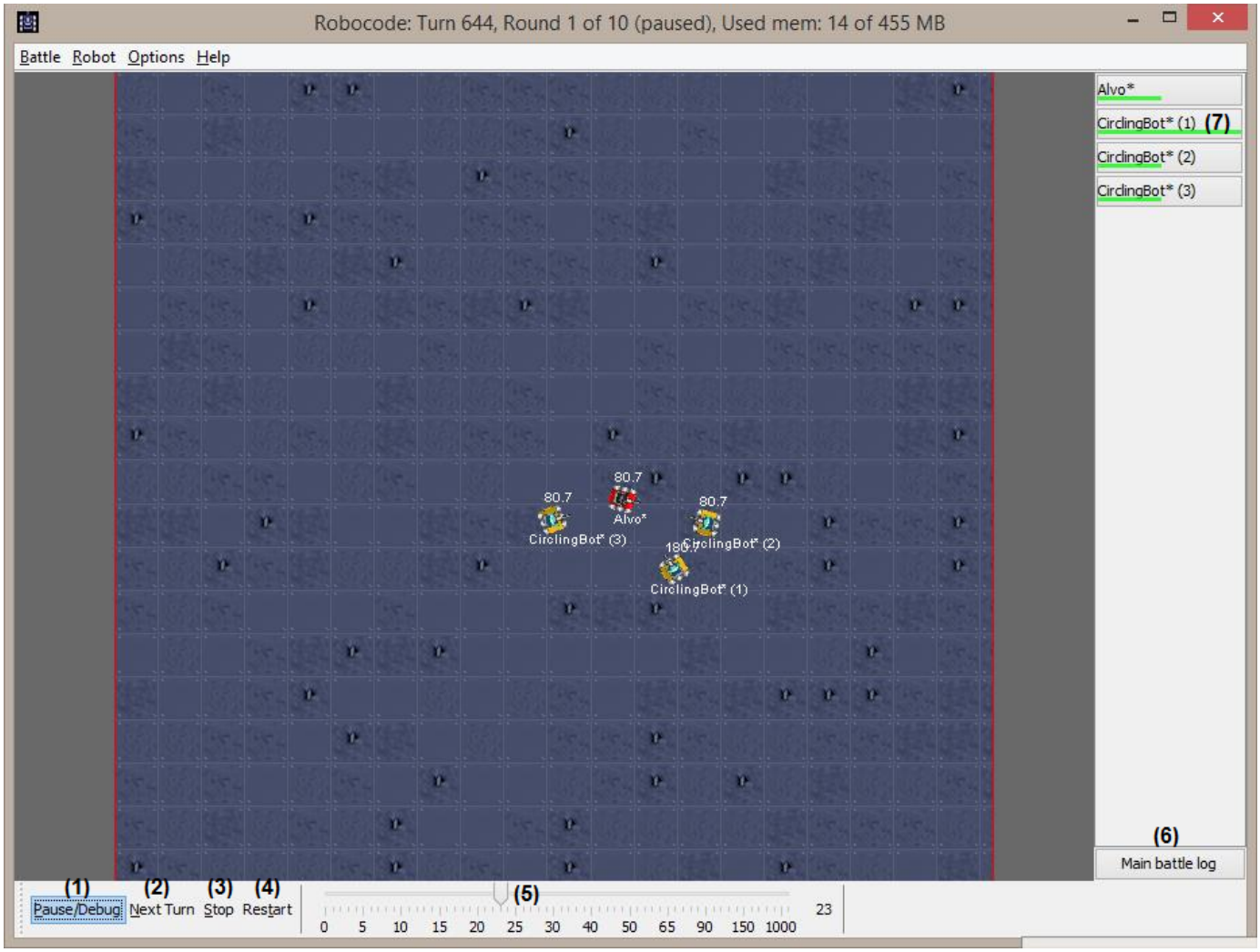

Figura 2.10: Janela com a Execução dos Robôs no Campo de Batalha.

Passo 10: Para uma melhor visualização da execução na aplicação, pode-se alterar o tamanho do campo de batalha, que é mensurado em pixels. A resolução padrão é de 800x600 pixels, mas a mesma pode ser aumentada através da guia (2) em Rules > Battlefield Size da Figura 2.8. Pode-se observar ainda na Figura 2.9 que existe uma barra vertical (1) e horizontal (2) para definir a resolução desejada.

Após as configurações da escolha dos robôs no Passo 9 e o tamanho do campo no Passo 10, deve-se clicar no botão (3) da Figura 2.9 para que a batalha seja inicializada, como mostra a Figura 2.10. É importante enfatizar que cada jogo iniciado tem 10 rodadas por padrão, mas esse valor pode ser alterado através da Guia Rules > Number of Rounds indicado em (4) na Figura 2.9. O recurso Hide Enemy Names (8), ao ser selecionado, tem a função de ocultar os nomes dos oponentes que estarão presentes no campo de batalha. Já os campos relacionados aos recursos (5), (6) e (7) serão discutidos no Capítulo 3.

Capítulo 3

Visão Geral da Plataforma

3.1 Introdução

3.1 Introdução

O Robocode é um simulador de batalhas de tanques de guerra baseado em eventos, que são executados pelos tanques. O evento é um resultado de uma ou mais ações que contempla as operações dos robôs que são codificados pelo desenvolvedor. A plataforma conta com uma série de características que definem as configurações e funcionamentos dos tanques e do campo de batalha.

Neste capítulo será abordado uma visão geral do Robocode, que foi escrita tendo como base a documentação oficial disponível em [Robocode 2014], contando com tópicos relacionados com a estrutura do robô e as regras físicas para o funcionamento do jogo que vão desde informações sobre tempo, distância, velocidade e energia até processos que envolvem a colisão entre robôs. Além disso, outros assuntos serão abordados como, por exemplo, o sistema de pontuação, ciclo de processamento durante uma rodada e os recursos e ferramentas da tela de batalha.

3.2 Anatomia do Tanque

De modo geral, um robô do Robocode é um tanque de guerra com tamanho de 36x45 pixels, ilustrado na Figura 3.1, cuja anatomia consiste por três componentes definidos como corpo, radar e canhão. Além disso, podem rotacionar 360 graus de modo independente e no início da batalha a sua posição inicial no campo é aleatória.

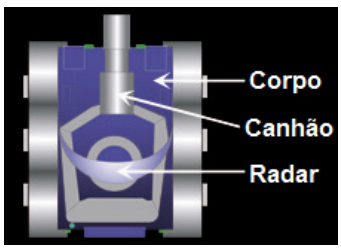

Figura 3.1: Estrutura de um Robô Simulado pelo Robocode.

O corpo é o tanque com o canhão e o radar, além de produzir os movimentos necessários para interagir com os inimigos em campo. Por exemplo, virar à direita (right), esquerda (left), andar para frente (ahead) e para trás (back). Através desses movimentos, pode-se definir uma locomoção mais complexa para melhorar as estratégias de ataque e defesa. Neste caso, é possível elaborar cálculos geométricos analíticos, algorítimos mais elaborados, entre outros.

O radar está relacionado com a capacidade sensorial do robô, o qual é usado para coletar informações precisas a respeito do mundo, de seus inimigos em campo e também de outros robôs companheiros. O radar tem movimento independente do corpo, podendo rotacionar 360 graus. Por exemplo, se alguns robôs em campo forem detectados pelo radar, o evento onScannedRobot, que será abordado no Capitulo ??, permite a implementação dos movimentos do canhão e as estratégias para atacar o oponente é disparado. Quanto mais complexas as estratégias de ataque, comandos mais complexos deverão ser codificados, sendo importante o uso de técnicas de IA e outros algorítimos.

O canhão, que dispara projéteis no oponente, pode girar para a esquerda e direita. Cada impacto por um projétil é capaz de retirar energia do adversário e consequentemente retornar pontos para o robô que o atingiu. A pontuação do jogo, bem como a energia de um robô e outras propriedades físicas serão destacadas a seguir.

3.3 A Física do Jogo

No Robocode, a física do jogo é dotada de propriedades e leis físicas utilizadas para o funcionamento das batalhas e dos robôs. Neste caso, deve-se considerar as características para movimentação, energia, emissão de projéteis e rotação dos componentes do corpo do robô, além das regras acerca do campo de batalha [Larsend 2010].

3.3.1 Campo de Batalha

O campo de batalha, ilustrado na Figura 2.10, é retangular e utiliza-se o sistema de coordenadas cartesianas (x, y), composta por paredes de tamanho variável, sendo que o tamanho padrão é de 800x600 pixels.

O Robocode usa uma convenção relacionada ao sentido horário do campo de batalha. Pode-se observar na Figura 3.2 que 0/360 graus, 90 graus, 180 graus, 270 graus é o sentido Norte, Leste, Sul, e Oeste, respectivamente.

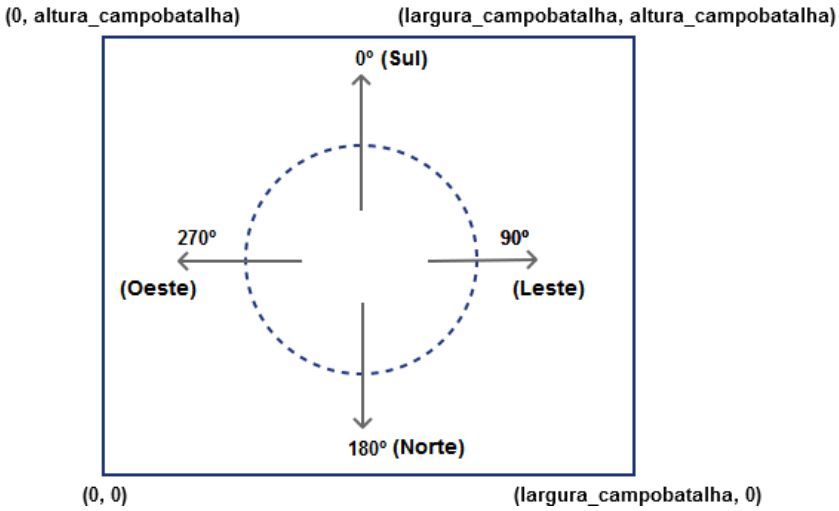

Figura 3.2: Sistema de Coordenadas e Sentido Adotado Como Convenção do Campo de Batalha no Robocode.

3.3.2 Medição de Tempo e Distância

O jogo é baseado em turnos (unidades temporais discretas) bem definidos a cada rodada. Cada turno é destinado para que o robô tenha uma quantidade limitada de tempo para a tomada de decisões e consequente escolha de ações de acordo com a melhor estratégia que lhe é atribuída naquele momento. Uma vez que o turno é completado, as ações selecionadas são executadas de forma sequencial. Entretanto, se um robô exceder o limite de tempo em um turno, o mesmo é perdido. Caso exceda esse limite 30 vezes em uma rodada, o robô é desclassificado.

A distância de um robô a um ponto é medida em pixels, com duas exceções. A primeira está relacionada a todas as distâncias que são medidas com precisão dupla, de modo que o robô possa se mover, na verdade, em uma fração de um pixel. A segunda exceção é que o Robocode dimensiona automaticamente o campo de batalha de acordo com uma escala específica de modo a caber na tela, já que as dimensões do tamanho do campo são ajustáveis. Neste caso, a unidade de distância é menor do que um pixel.

3.3.3 Energia

Inicialmente, os robôs possuem 100.0 unidades de energia a cada rodada. A energia é a propriedade mais importante do jogo, pois a sua diminuição ou aumento implicará no desempenho geral dos robôs na batalha.

A energia é comprometida se um robô for atingido por projéteis dos adversários, se houver colisão com outros robôs ou com a parede do campo de batalha. Com zero unidades de energia o robô sai do campo de batalha até começar uma nova rodada, na qual é iniciada sempre quando restar um ou nenhum robô em campo. A energia também pode ser recuperada ao causar danos no oponente. Por exemplo, se um robô atingir um oponente, o triplo da quantidade de danos causados é recebido pelo robô em forma de energia. Essas quantidades serão mais detalhadas nas Subseções 3.3.5 e 3.3.7.

3.3.4 Física do Movimento dos Robôs

O Robocode trata a velocidade como um vetor, que segue a direção e sentido pela qual o robô se redireciona (robot's heading). A velocidade pode ser de 0.0 a 8.0 pixels por turno, no qual o robô permanecerá parado em 0.0 ou terá velocidade máxima em 8.0 pixels por turno. Os valores que variam de -8.0 a 8.0 representam a velocidade em relação ao sentido do movimento do tanque, sendo que a parte negativa indica movimento para trás (retrógrado), enquanto que a velocidade positiva indica o movimento para a frente.

A aceleração é baseada na distância na qual o robô está se movendo, ou seja, quanto mais perto da posição que deseja alcançar, a sua aceleração diminui em mesma proporção. Os robôs podem acelerar a uma taxa de 1 pixel por turno e desacelerar a 2 pixels por turno. Isso quer dizer que, caso o robô se mova a uma velocidade de 8.0 pixels por turno e reverta sua direção, suas velocidades serão [6.0, 4.0, 2.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0].

Com o valor da velocidade, a equação da distância nativa do robô é dada pela Equação 3.1

$$distancia = velocidade * tempo \quad (3.1)$$

A rotação do corpo de um robô, calculada pela Equação 3.2, é determinada por uma lei relacionada à velocidade, ou seja, quanto mais rápido é o movimento do robô, mais devagar a curva é realizada.

$$rotacao = 10 - 0.75 * abs(velocidade) \quad (3.2)$$

O termo abs(velocidade) indica a velocidade absoluta, ou seja, apenas a parte positiva é considerada. Vale ressaltar que o resultado da Equação 3.2 será dado em graus, com variação de 0 a 360 graus, já que está relacionada ao ângulo de giro do robô.

3.3.5 Regras Físicas do Projétil

No que rege às regras do comportamento físico do projétil, a propriedade mais importante é a potência de disparo, denominado como firepower. Em geral, quanto maior o firepower, mais danos o projetil acarretará ao inimigo e menor será a sua velocidade para atingir o alvo devido ao seu peso. Os valores máximo e mínimo do firepower são 0.1 e 3.0, respectivamente. Ao atingir o oponente, o robô recebe mais energia, sendo o triplo do firepower do projétil. Ao mesmo tempo, ao atirar, o robô perde energia que corresponde ao firepower do projétil. Por exemplo, caso um robô tenha uma quantidade de energia menor que 0.1 e o oponente é atingido, o robô que atirou irá ganhar 0.3 em energia.

A velocidade do projétil pode ser obtido pela Equação 3.3.

$$velocidadedoprojetil = 20 - 3 * firepower \quad (3.3)$$

Um projétil retira energia de um robô, ou seja, o seu dano é calculado pelas Equações 3.4 e 3.5, se o firepower for menor ou igual a 1 ou se for maior que 1, respectivamente.

$$dano = 4 * firepower \quad (3.4)$$

$$dano = 4 * firepower + 2 * (firepower - 1) \quad (3.5)$$

Em relação ao intervalo de tempo entre um tiro e outro, há uma propriedade física definida como Gun Heat, na qual o canhão possui um poder de aquecimento quando um projétil é atirado. Tal aquecimento gerado tem um valor característico dado pela Equação 3.6.

$$aquecimentodoprojetil = 1 + (firepower/5) \quad (3.6)$$

À medida que o canhão “esfria”, o tempo do intervalo para o próximo lançamento do projétil diminui. Isso ocorre através de uma taxa padrão temporal definida e que corresponde a 0.1 por turno. Isso significa que, caso o firepower seja igual à 3.0, a arma atirará a cada 16º turno. Esse resultado pode ser dado se a Equação 3.6 for dividida pela taxa temporal, isto é: [1 + (3.0/5)]/0.1 = 16.

Através dessa propriedade não é possível estabelecer um intervalo para o lançamento do projétil para cada robô em campo, já que a mesma é dependente do firepower, assim como mostra a Equação 3.6. Neste caso, deve-se esperar que a arma “esfrie” novamente para o próximo tiro.

A taxa padrão temporal que define o esfriamento do canhão pode ser configurada antes da batalha ser iniciada. Ao escolher os robôs que atuarão em campo na Figura 2.8, pode-se alterar o valor padrão 0.1 em New Battle > Rules > Gun Cooling Rate em (5) da Figura 2.9. Por exemplo, se o valor for maior que 0.1, o intervalo de tempo do tiro aumenta. É importante destacar que o valor minímo é 0.1 e máximo é 0.7.

3.3.6 Heading e Bearing

As propriedades heading e bearing são importantes e largamente usadas para diferentes cálculos de posicionamento, localização do inimigo e estratégias de movimento no Robocode.

A Figura 3.3 ilustra a relação das propriedades com o robô. O heading (h) é um ângulo vinculado à direção na qual o robô aponta no campo de batalha. Esse ângulo é absoluto, ou seja, é fixo em relação aos pontos cardiais e está compreendido entre 0 e 360 graus, representado pelo ângulo tracejado e a origem de (h) em 0 graus. Vale destacar que esta propriedade é particular de cada componente do robô, isto é, o radar, o corpo e o canhão tem seu próprio heading, semelhante ao processo de movimento independente abordado na Seção 3.2. Por exemplo, na Figura 3.3, o corpo, o radar e o canhão estão posicionados na mesma direção, possuindo o mesmo ângulo (h). É

permitido, portanto, a utilização os diferentes tipos de heading em cada componente, seja para mirar o canhão em uma determinada direção no inimigo, para posicionar o corpo de forma a aprimorar o movimento ou manipular a direção do radar.

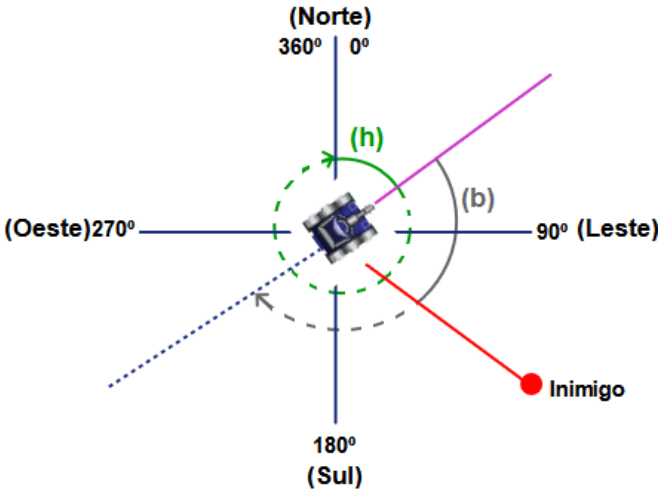

Figura 3.3: Representação das Propriedades Heading e Bearing de um Robô no Campo de Batalha.

A propriedade bearing (b) é um ângulo compreendido entre o heading do robô e algum objeto no campo de batalha, que pode ser a parede, um inimigo ou outro robô de um time, assim como ilustra a Figura 3.3. Trata-se de um ângulo relativo, ou seja, varia em relação a uma referência que, neste caso, é o heading do robô. Essa propriedade está limitada entre 0 a 180 graus, representado pelo ângulo tracejado e a origem de (b) a partir de (h).

A soma das propriedades heading e bearing permite estabelecer, por exemplo, o ângulo do inimigo, definido na Equação 3.7 em relação ao heading do robô.

$$anguloInimigo = (h) + (b) \quad (3.7)$$

3.3.7 Colisões

No Robocode, há colisões entre robôs e a parede do campo de batalha. A colisão entre dois robôs também pode comprometer sua energia e alterar as pontuações. Por exemplo, se dois robôs colidirem, ambos perdem 0.6 unidades de energia. Contudo, caso o robô vá de encontro com o oponente para uma colisão ou golpe proposital, fenômeno esse conhecido como Ramming, a sua recompensa será de 1.2 unidades energéticas. Ao colidir com uma parede, o robô perde 3.0 unidades de energia.

3.4 Processamento

Para executar uma rodada, o ciclo de processamento usado no Robocode executa cada tarefa através da seguinte ordem:

- Todos os robôs executam seu código até partirem para as ações;
- O tempo é atualizado a cada turno;
- Todas os projéteis movem-se, verifica-se as colisões dos mesmos com as paredes e os inimigos, colisões entre robôs e entre um robô e a parede;
- Todos os robôs se movimentam, processando as seguintes propriedades, nesta ordem: corpo, canhão, radar, heading desses três primeiros elementos, aceleração, velocidade e distância;
- Todos os escaneamentos do radar são executados;

- Coleta de mensagens entre times são realizadas, caso haja interação entre robôs;
- Cada robô processa sua fila de eventos para executar uma nova ação, iniciando o ciclo novamente.

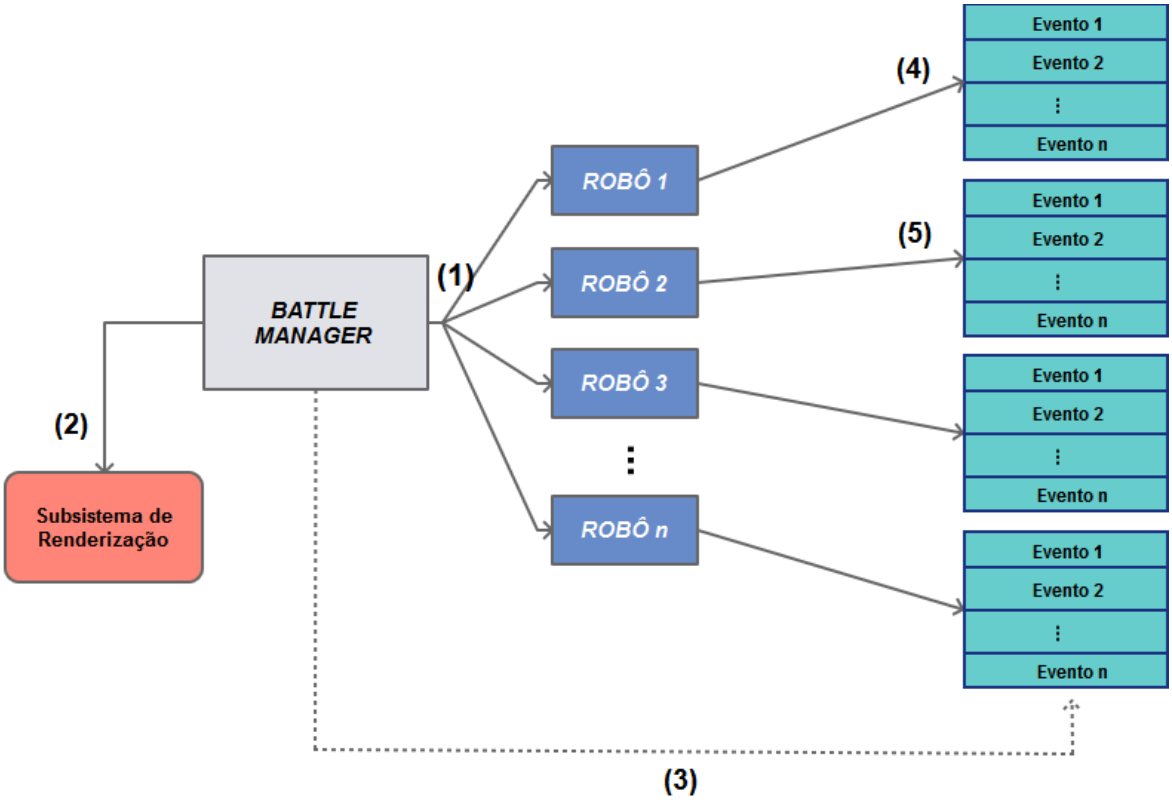

Figura 3.4: Esquema de Execução e Gerenciamento de Robôs e Eventos pelo Battle Manager.

A Figura 3.4 ilustra o processo de execução de cada robô, que é uma thread em Java[1] gerenciada pelo Battle Manager em (1). Esse componente é responsável também por gerenciar o Subsistema de Renderização (2) do campo de batalha e outros gráficos 2D do Robocode. Ao mesmo tempo, o Battle Manager popula as filas da thread com os eventos relacionados como em (3). Os eventos que englobam as ações dos robôs são executadas em sequência, ficando alocados nas filas (4) e (5).

3.5 Pontuação

Segundo [Gade et al. 2003], o Robocode possui um sistema de pontuação para determinar o ganhador da batalha, que é definido pelo robô que possui o melhor desempenho durante cada rodada e consequentemente a maior quantidade de pontos. Portanto, os seguintes critérios devem ser utilizados:

- Pontos Totais (Total Score): soma total dos pontos obtidos durante as n-rodadas;
- Pontos de Sobrevivência (Survival Score): durante uma rodada, cada robô vivo no jogo ganha 50 pontos quando um outro morrer. Esse valor se acumula durante as n-rodadas totais estabelecidas pelo programador;
- Bônus do Último Sobrevivente (Last Survivor Bonus): o último robô que sobreviver em campo ganha um adicional de 10 pontos por cada robô que morreu;
- Dano Causado por Projéteis (Bullet Damage): a cada ponto de danos causados no oponente, o robô que causou o mesmo é recompensado com 1 ponto;
- Bônus por Dano Causado por Projéteis (Bullet Damage Bonus): ao matar o inimigo, o robô ganha um adicional de 20% do total de

danos causados no oponente;

- Ram Damage: um robô que colide ou golpeia o oponente propositalmente, cuja ação é denominada por ramming, é recompensado com 2 pontos;
- Ram Damage Bonus: ao matar o inimigo por ramming, o robô ganha um adicional de 30% do total de danos causados no oponente.

Ao final do jogo é executada todas as suas rodadas, o desempenho de cada robô pode ser visualizado em uma Tabela ilustrada na Figura 3.5. Pode-se salvar as informações geradas em formato CSV (Comma Separated Value) e fechar a janela, clicando em (1) e (2), respectivamente.

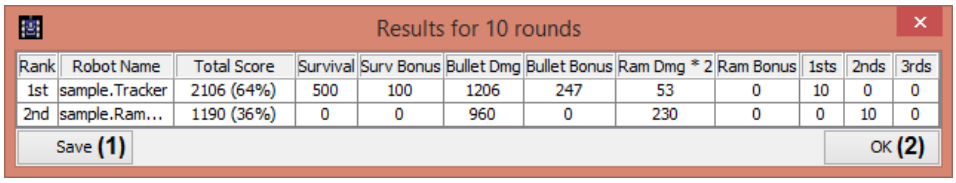

Figura 3.5: Tela com os Resultados Finais e Pontuações.

3.6 Recursos da Tela de Batalha

A tela da Figura 2.10 onde a batalha é executada apresenta diferentes funcionalidades durante as rodadas. Por exemplo, uma vez que o código dos robôs é implementado no Netbeans, pode-se incluí-los em um campo de batalha e verificar algumas de suas propriedades físicas e informações do jogo, que serão descritas nesta seção, enquanto uma rodada é executada.

3.6.1 Controles Básicos

Os controles básicos da tela de batalha incluem pausar, parar, reiniciar e pular para o próximo turno. A seguir serão apresentados os botões disponíveis na Figura 2.10:

(1)
Pause/Debug: pausa o jogo e também pode ser usado no processo de depuração do código;

(2)
Next Turn: uma vez que o jogo é pausado clicando-se no botão (1) da Figura 2.10, é possível observar cada turno no qual os robôs agem de modo isolado;

(3)
Stop: encerra uma batalha;

(4)
Restart: reinicia uma nova rodada, interrompendo a atual. Por exemplo, se a batalha for configurada para 10 rodadas e o botão (4) for clicado na quarta rodada, o jogo reiniciará na primeira rodada, ou seja, 1 de 10;

(5)
Controle de Velocidade: é possível manipular a velocidade com a qual a batalha ocorre. Os números em (5) indicam a velocidade atual escolhida.

3.6.2 Uso do Console

No Robocode, o console ilustrado na Figura 3.6 é uma janela com informações individuais que são dinamicamente atualizadas ao longo das ações efetuadas por cada robô em campo. Para acessar o console de um robô em campo, basta clicar no botão que contém o nome do

robô desejado. Por exemplo, o console do CirclingBot*(1) estará disponível através do botão (1) da Figura 2.10.

As informações da guia Console incluem a rodada atual e anteriores do jogo, além de mensagens enviadas por outros integrantes caso o jogo envolva um time de robôs. É importante destacar que o console não apresenta informações referentes ao inimigo. Trata-se também de uma ferramenta extremamente útil para o programador, pois é possível identificar algum problema na modelagem do robô.

Além do status geral do robô, é possível visualizar suas propriedades físicas, alteradas dinamicamente ao longo de turnos e ações efetuadas em uma rodada, de forma a permitir uma análise sobre como o robô responde à modelagem programada pelo desenvolvedor. Essas propriedades incluem energia, posições x e y, velocidade, gun heat, estado (ativo ou inativo), bodyHeading, gunHeading e radarHeading. As três últimas propriedades serão abordadas posteriormente no Capítulo 4.

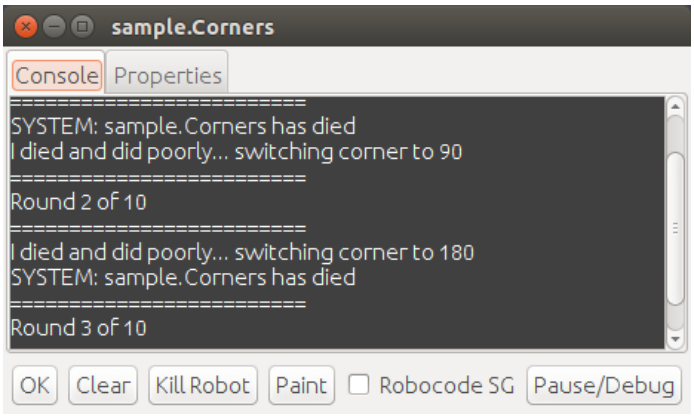

Figura 3.6: Tela do Console com Informações Transmitidas pelo Robô ou pelo Robocode.

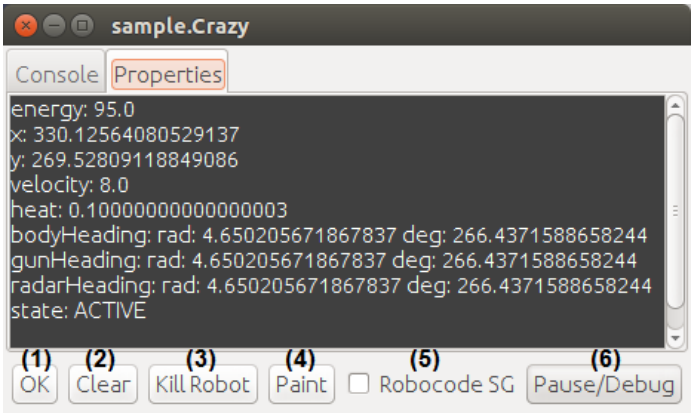

Figura 3.7: Tela do Console com Informações Sobre as Propriedades Físicas do Robô.

Os botões das Figuras 3.6 e 3.7 apresentam as seguintes funções:

(1)

OK: fecha a janela;

(2)

Clear: exclui todo o texto impresso no console;

(3)

Kill Robot: mata o robô em campo, tirando-o da rodada;

(4)

Paint: ativa / desativa o processo de painting do robô ao usar o método onPaint;

(5)

Robocode SG: ativa ou desativa o modo de compatibilidade de um robô codificado para ser utilizado no Robocode SG (Swing Graphics). Este recurso permite a criação de desenhos gráficos na tela de batalha do Robocode[2] ;

(6)

Pause / Debug: o modo de pausa do jogo pode ser usado no processo de depuração do código do robô programado.

3.6.3 Main Battle Log

O recurso Main battle log (6) da Figura 2.10 tem como objetivo verificar o status geral da batalha. Essa ferramenta permite identificar alguma falha ou erro de código de um robô durante a sua execução em campo. Ao clicar em (6), será aberta a janela da Figura 3.8, que indica a rodada atual e anteriores na guia Console (1). Também é uma alternativa ao uso do Console, recurso abordado na Subseção 3.6.2, que permite visualizar informações de todos os robôs envolvidos em campo.

Por se tratar de um recurso que armazena informações e mensagens do sistema ao longo das rodadas, pode-se limpar todo o conteúdo gerado na janela clicando no botão Clear (4) e fechar a janela através de (3).

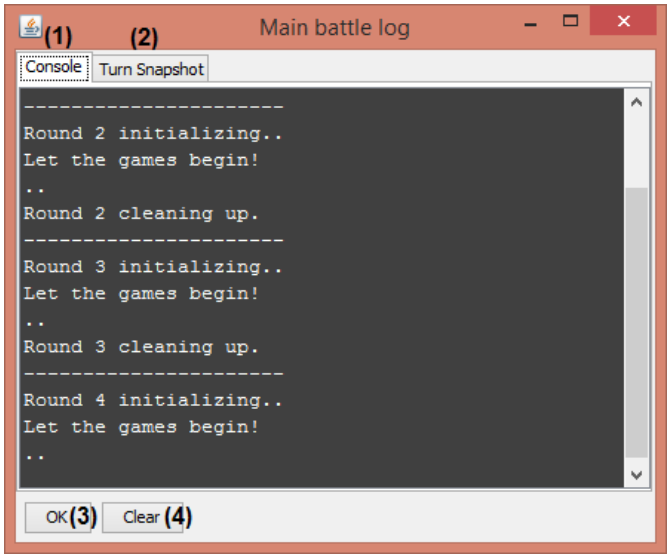

Figura 3.8: Tela do Main battle log, Mostrando a Rodada Atual e Anteriores na Guia Console.

Na guia Turn Snapshot (2) da Figura 3.8, pode-se visualizar dinamicamente as ações e propriedades físicas dos robôs que atuam no campo de batalha, sendo representadas em código XML e estão ilustradas na Figura 3.9. Essas informações podem auxiliar o desenvolvedor a depurar o código dos robôs caso haja algum erro na execução. Além disso, esse código XML apresenta informações sobre os projéteis atirados e que percorrem o campo de batalha.

Figura 3.9: Tela do Main battle log, que Mostra os Status de Cada Robô e os Projéteis Via Representação XML na Guia Turn Snapshot.

3.6.4 Scan Arc

O Scan Arc, ou arco de escaneamento, é uma ferramenta gráfica que auxilia o desenvolvedor na visualização do funcionamento do radar, permitindo também a detecção de erros do componente por meio da depuração do código do robô. O recurso é apresentado Figura 3.11, na qual os arcos dos dois robôs são destacados através da linha tracejada em cores diferentes.

Este recurso está relacionado a um arco que percorre todo o campo de batalha e que se forma entre duas unidades temporais (turnos) cada vez que o radar rotaciona, isto é, o arco formado entre a posição anterior e atual do radar. Dessa forma, tudo que o radar detectar dentro do arco deve ser enviado para o evento onScannedRobot(), responsável pelas ações e decisões a serem tomadas quando um outro robô é detectado e que será abordado no Capítulo 4.

O Scan Arc é um recurso opcional do Robocode, que por padrão está desabilitado. Para habilitá-la, deve-se acessar a tela da Figura 2.6 e selecionar o menu Options > Preferences > Guia View Options em (1) da Figura 3.11 e selecionar a opção View Scan Arcs (2). Para confirmar o recurso, deve-se clicar em (3).

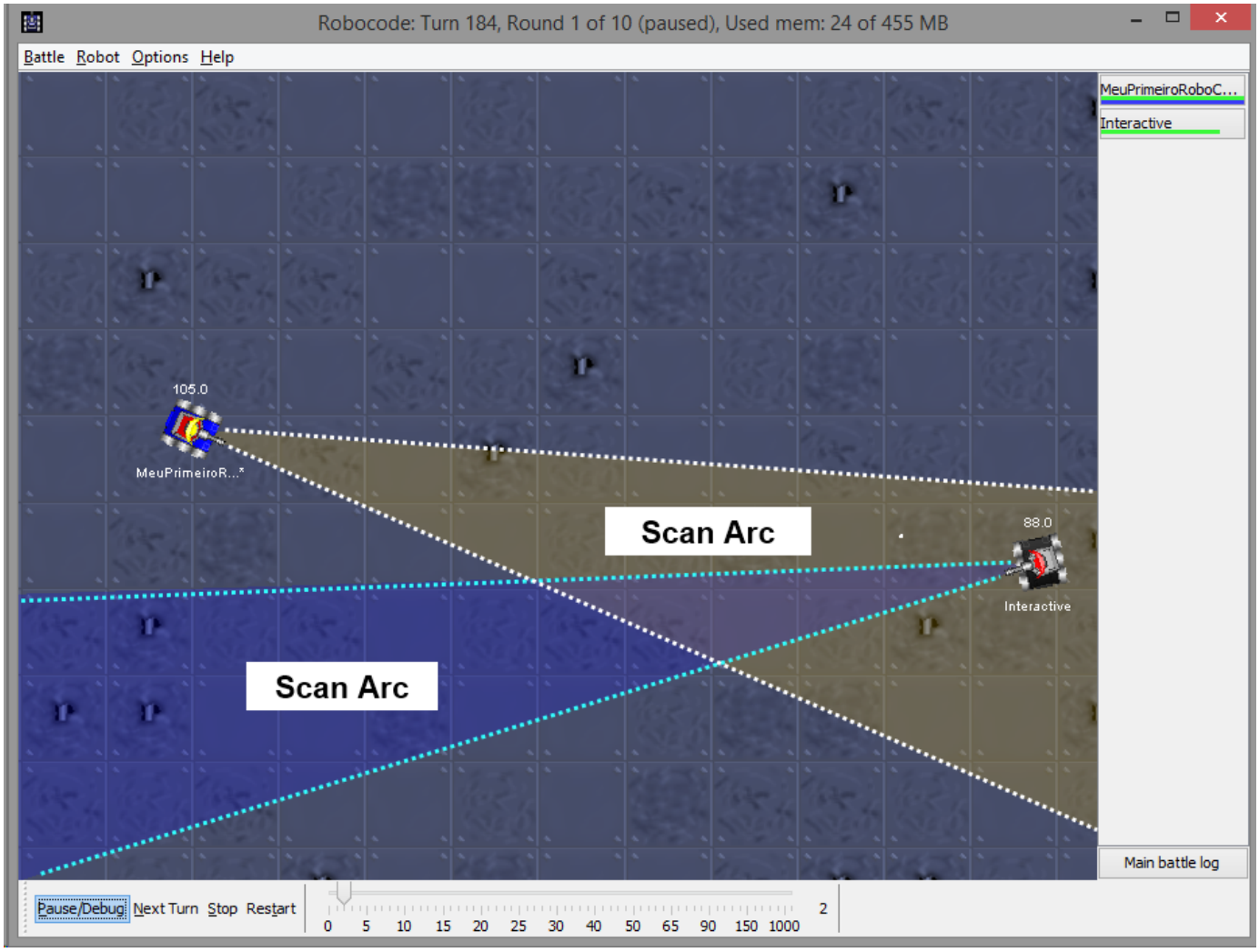

Figura 3.10: Tela do Campo de Batalha com o Recurso Scan Arc Habilitado.

Uma vez ativado esse recurso, o arco de escaneamento será atribuído para todos os robôs que competirão no campo de batalha. Para visualizar o efeito da ferramenta, deve-se adicionar, no mínimo, dois robôs e iniciar a batalha, repetindo o procedimento 2 do Passo 9, abordado no Capítulo 2. Para selecioná-los, acessar o pacote sample do Painel Packages da Guia Robots (1) da Figura 2.8, que possui vários robôs já codificados como exemplos de demonstração e que são incluídos no processo de instalação do Robocode descrito no Capítulo 2. Escolher dois robôs quaisquer e clicar no botão Start New Battle (3) para que a batalha seja inicializada de acordo com a Figura 3.10. Como os robôs atuando em campo, observa-se também que, quando o radar está parado para executar as ações após detectar a presença do oponente, o arco deixa de existir, passando a ser representado por uma linha reta.

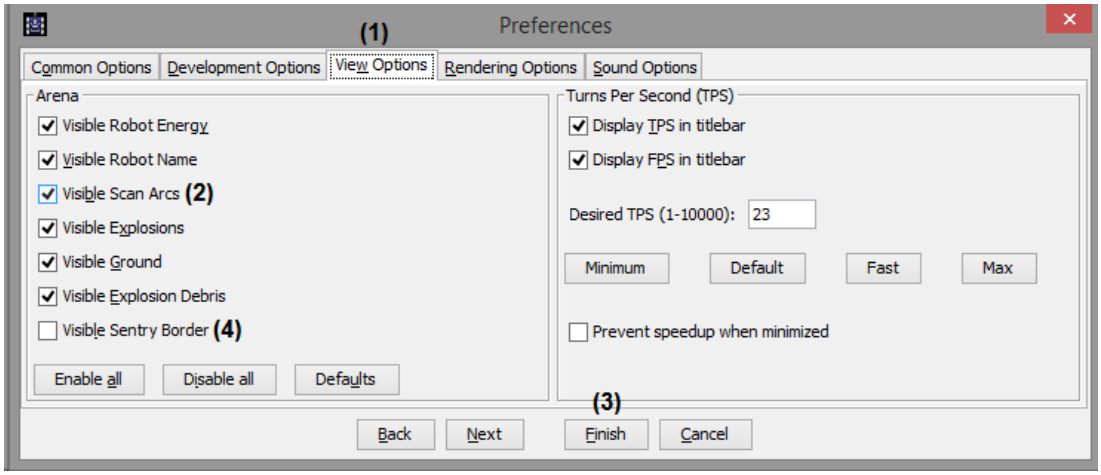

Figura 3.11: Tela de Opções de Visualização do Robocode.

O radar pode rotacionar em um ângulo de, no máximo, 45 graus a cada turno. Além disso, o radar permite detectar outros robôs presentes em um alcance de até 1200 pixels. Usando o arco de escaneamento e aumentando o redimensionamento do campo de batalha é possível observar o limite de alcance do radar.

Para visualizar uma batalha com o Scan Arc, primeiramente deve-se seguir os procedimentos do item 2 do Passo 9 e 10 do Capítulo 2 para se alterar a resolução para 2000x2000 pixels através da barras vertical (1) e horizontal (2) da Figura 2.9. O próximo passo é selecionar dois robôs quaisquer e iniciar a batalha no Robocode. O resultado é semelhante à Figura 3.12. Pode-se observar que o arco de escaneamento atua em até 1200 de 2000 pixels anteriormente configurados.

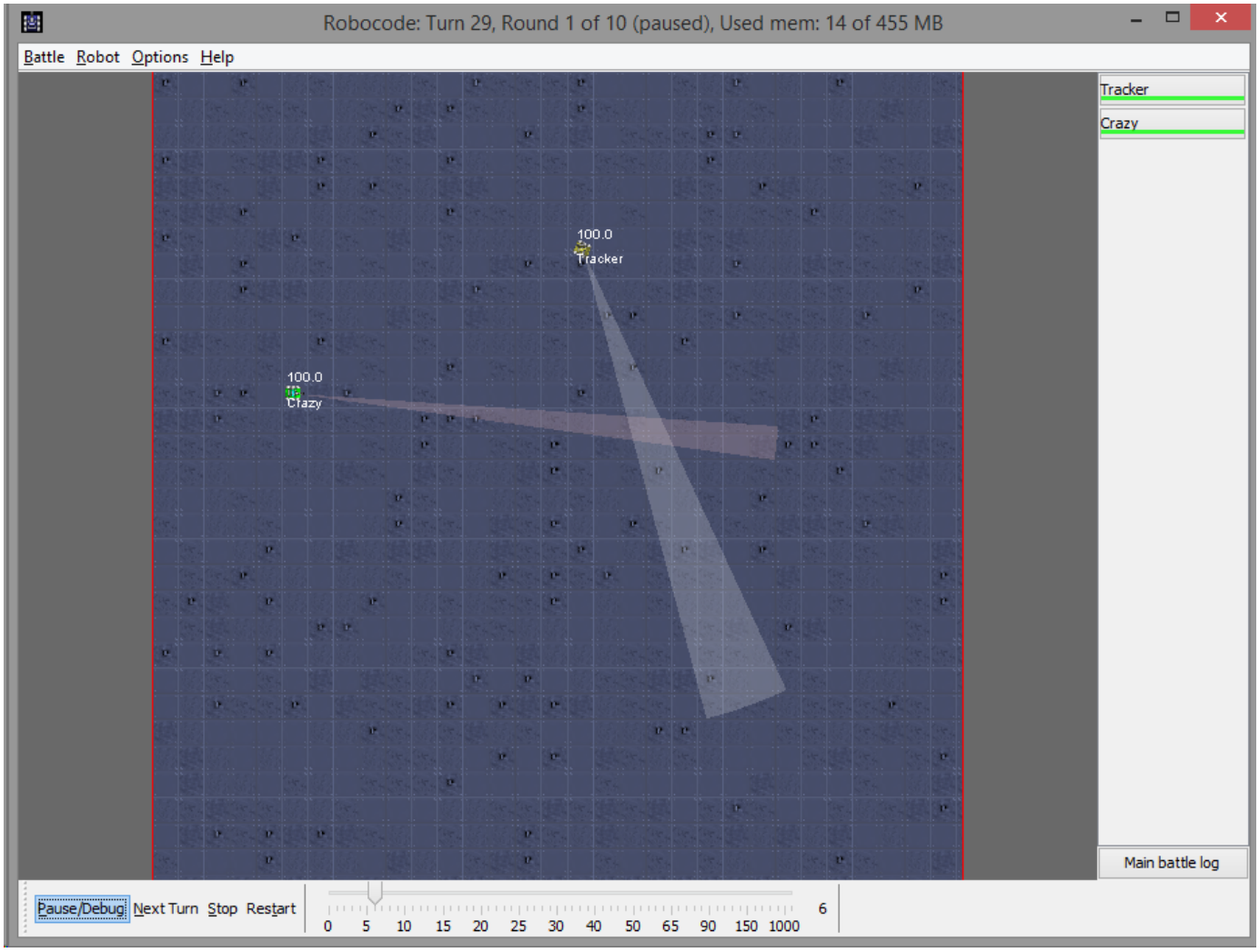

Figura 3.12: Tela do Campo de Batalha Redimensionado para 2000x2000 pixels com o Recurso Scan Arc Ativado.

3.6.5 Inactivity Time

No campo (6) da Figura 2.9 há a opção Inactivity Time, que estabelece o tempo máximo, em turnos, que um robô pode ficar inativo no campo de batalha antes de executar alguma ação. A inatividade de um robô está relacionada a não execução de quaisquer ações, no qual permanece inerte durante um certo tempo no campo de batalha. Trata-se de um recurso do jogo que força a morte do robô caso esse não execute alguma ação, permitindo que o mesmo retorne em uma nova rodada.

No Robocode, o valor padrão do tempo de inatividade, assim como indicado em (6) da Figura 2.9, é de 450 turnos, configurado sempre antes de iniciar uma batalha, além de ser alterado para valores a partir de 1. Após esse intervalo de tempo, caso algum robô ainda esteja inerte na batalha, o mesmo irá perder energia em uma taxa de 0.1 por turno. Entretanto, à medida que o robô passa a agir em campo, esse tempo é reiniciado.

3.6.6 Sentry Border

O Sentry Border, indicado na Figura 3.13, está relacionado a uma cerca inserida no campo de batalha, formada desde a parede até um tamanho inferior do campo e é mensurada em pixels. Este recurso foi adicionado na versão 1.9.0.0 do Robocode e funciona somente com robôs que implementam a interface BorderSentry[3] , que têm a função básica de agirem como guardiões e que limitam seus movimentos somente dentro da borda criada, não sendo possível acessar o campo fora dela.

Com o intuito de manter outros robôs fora da cerca, para aqueles que não forem codificados com o Sentry Border e forem designados a combater com adversários em campo, a borda passa a ser considerada como uma zona de perigo, na qual um robô do tipo BorderSentry atira naqueles que invadirem a cerca. A intenção, portanto, é codificar robôs capazes de permanecerem na área mais segura do campo. É importante destacar que batalhas que possuem esse tipo de robô têm de ter dois ou mais robôs que não usam esse recurso, pois o guardião não luta em campo.

Um exemplo prático é apresentado na Figura 3.13 utilizando o robô nativo BorderGuard, contido no pacote de demonstração samplesentry, visualizado no painel (4) da Figura 2.8. Logo após escolher dois robôs quaisquer do pacote sample e clicar no botão Start New Battle (3).

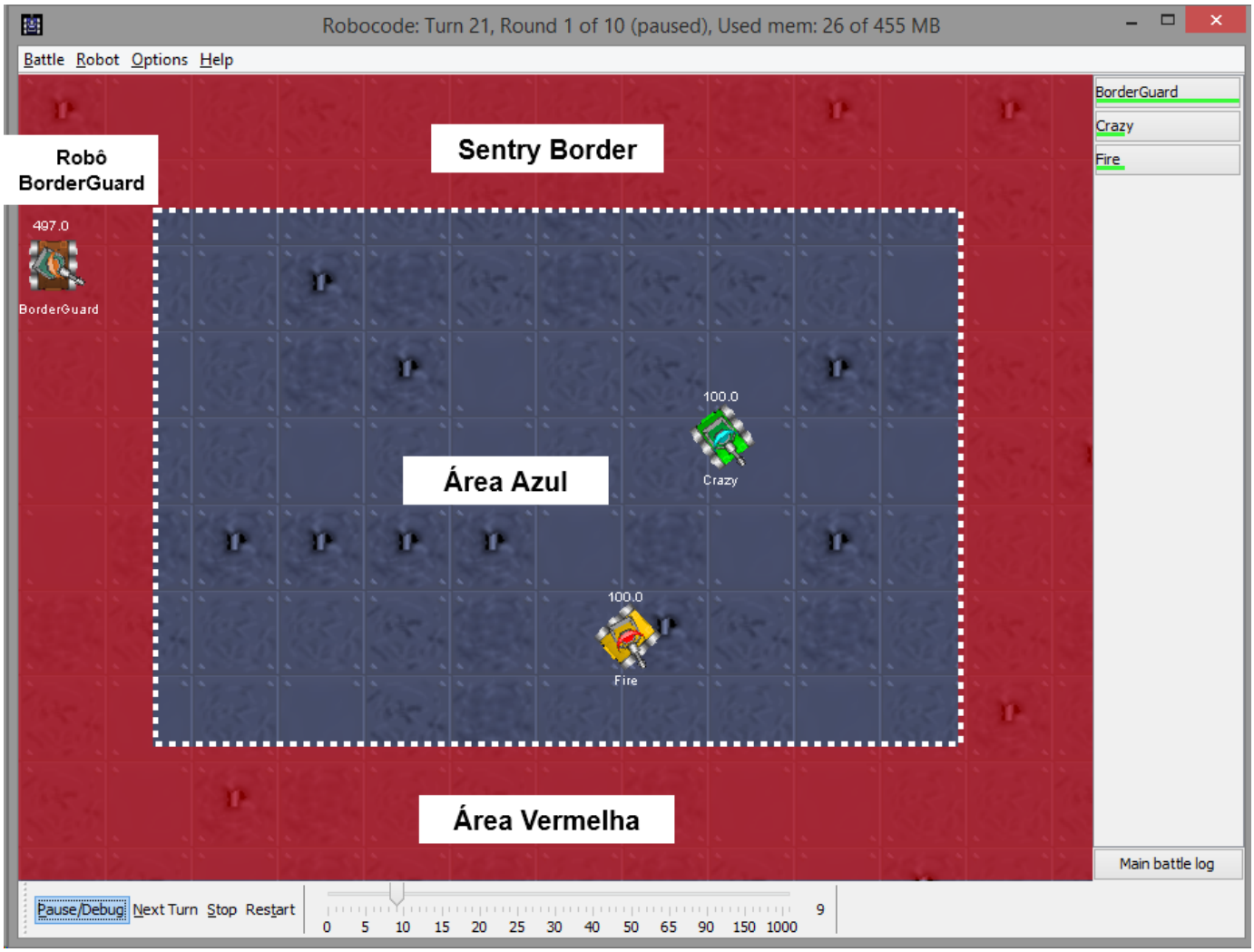

Figura 3.13: Tela do Campo de Batalha que Demonstra o Recurso Sentry Border, Tendo o Robô BorderGuard como Guardião da Borda.

Observa-se na Figura 3.13 que o Sentry Border equivale à área vermelha e que a zona segura para a batalha está relacionado à área azul. Executando esse exemplo, percebe-se que o robô BorderGuard movimenta-se apenas dentro da área vermelha e, caso os robôs Crazy ou Fire acessem essa zona, o guardião passará a seguí-los para golpeá-los através do fenômeno ramming, abordado na Subseção 3.3.7, ou atirará com firepower máximo, ou seja, igual a 3.0. É importante destacar que, via regra, caso o guardião atire e o projétil acesse a zona fora da borda, o mesmo não causa nenhum dano nos demais robôs, valendo somente para aqueles que invadam a borda. Contudo, o guardião perde energia ao ser atingido por projéteis lançados de outros robôs posicionados na área azul. Além disso, robôs do tipo BorderSentry tem 400 unidades de energia extra, totalizando 500 unidades, já que todo robô tem 100 no início de uma rodada. A alta

quantidade de energia está relacionada com a função de vigiar a borda, sobrevivendo o máximo de tempo possível enquanto durar uma rodada.

Não basta apenas adicionar o robô BorderGuard para visualizar a borda, é necessário ativar o recurso. Para habilitá-lo, deve-se acessar o menu Preferences > Guia View Options e selecionar a opção Visible Sentry Border (4) da Figura 3.11. Além disso, é possível configurar o tamanho da borda ao iniciar um novo jogo através da janela da Figura 2.9, alterando o campo Sentry Border Size (7). O tamanho padrão do Robocode é de 100 pixels e, considerando que o tamanho de um robô é 36x45 pixels, o valor do campo Sentry Border Size tem um valor mínimo de 50 pixels.

3.7 Tamanho do Código

O tamanho do código de um robô é representado pela quantidade de código executável contido em um arquivo .class ou .jar, mensurado em bytes. Trata-se de uma restrição importante para o Robocode, pois o tamanho do código pode classificar o tipo de robô, além de incentivar os desenvolvedores com o uso de boas práticas de programação. Neste sentido, quanto mais complexo for o robô, por exemplo, a implementação de diferentes características e a utilização de algoritmos complexos, é de se esperar que seu código tenha um tamanho maior.

Em relação à classificação dos robôs pelo tamanho, o Robocode o restringe em seis categorias, descritas a seguir.

- MiniBots: 1499 bytes ou menos;
- MicroBots: 749 bytes ou menos;
- NanoBots: 249 bytes ou menos;
- Twin Duel: 1999 bytes ou menos;
- Megabots: sem restrições;
- Times: sem restrições.

Mais detalhes acerca dessa característica podem ser consultados no Robowiki[4] .

Capítulo 4

Desenvolvendo Robôs

4.1 Introdução

Neste capítulo será apresentado a estrutura básica utilizada para implementar no NetBeans um robô no Robocode. Através de exemplos práticos deverá dar destaque na forma de implementar os recursos de escaneamento do radar, lançamento de projéteis pelo canhão e movimentação dos robôs.

Uma vez integrado o Robocode no NetBeans e com a aplicação RobotTest criada nos Passos 3 e 4 do Capítulo 2, além de considerar as regras descritas no Capítulo 3, deve-se agora haver um primeiro contato com o código Java para a implementação dos robôs.

4.2 Meu Primeiro Robô

Antes de começar a implementar um algoritmo e observar o que acontece, o primeiro passo consiste em criar a classe MeuPrimeiroRobo. No painel lateral Projetos do NetBeans é necessário clicar com o botão direito do mouse no pacote robottest criado no Passo 4 do Capítulo 2 e selecionar Novo > Classe Java.... Para aparecer a janela da Figura 4.1. No campo (1), deve-se inserir "MeuPrimeiroRobo"e clicar no botão Finalizar (2). A nova classe será criada no NetBeans como MeuPrimeiroRobo.java para implementar o Algoritmo 4.1. Para executar o algoritmo deve-se clicar no menu Executar > Executar Projeto (RobotTest) ou utilizar a tecla de atalho F6 para que a tela inicial do Robocode seja inicializada.

Para adicionar o robô criado no Algoritmo 4.1 no campo de batalha, o procedimento 2 do Passo 9 abordado no Capítulo 2 deve ser executado. Como explícito anteriormente, espera-se que o pacote robottest do projeto apareça e seja selecionado no painel (3) da Figura 2.8. O próximo passo é selecionar o robô Meu Primeiro Robô no painel (5).

Algoritmo 4.1: MeuPrimeiroRobo.java

```
1package robottest;
2import robocode.*;
3
4public class MeuPrimeiroRobo extends Robot{
5
6       public void run() {
7       while (true) {
8           ahead(100);
9           turnGunRight(360);
10           back(100);
11           turnGunRight(360);
12       }
13    }
14
15    public void onScannedRobot(ScannedRobotEvent e) {
16        fire(1);
17    }
18}
```

Para adicionar um oponente, pode-se acessar o pacote sample em (4) da Figura 2.8, que possui vários robôs já codificados. Para esse exemplo o robô Target será selecionado, que foi desenvolvido para servir de alvo e movimentar-se cada vez que perde 20 unidades de

energia. Para iniciar a batalha, o botão Start New Battle (3) deve ser clicado.

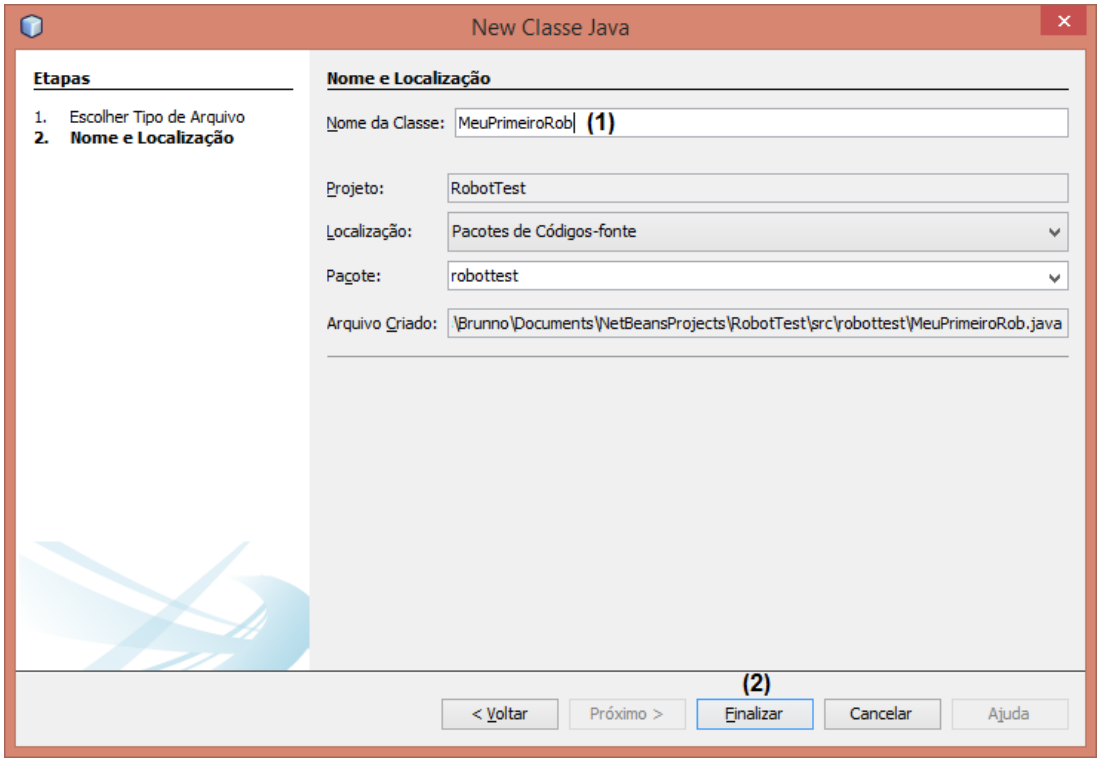

Figura 4.1: Tela para Criação da Nova Classe Java no NetBeans.

Observando os movimentos e ações do robô MeuPrimeiroRobo, pode-se destacar os seguintes passos do Algoritmo 4.1:

Linha 2:

importa todos os objetos do Robocode para serem usados no desenvolvimento do robô;

Linha 4:

declara a classe MeuPrimeiroRobo do tipo public, que herda a classe Robot, isto é, o objeto declarado é do tipo Robot e nomeado como MeuPrimeiroRobo. Mais detalhes sobre os tipos de classes nativas a serem herdadas durante o desenvolvimento de um robô serão abordadas na Seção 4.4;

Linha 6:

o Robocode chama o método run() quando uma batalha é iniciada e deve conter as ações que o robô deverá executar enquanto estiver agindo durante uma rodada;

Linha 7:

o laço while executa os comandos que estiverem dentro do mesmo loop enquanto a condição true for verdadeira, ou seja, para sempre enquanto o robô sobreviver durante uma rodada;

Linha 8:

move o corpo do robô 100 pixels para frente;

Linha 9:

gira o canhão para a direita em 360 graus;

Linha 10:

move o corpo do robô 100 pixels para trás;

Linha 15:

evento de detecção de outro robô pelo radar, que gera a instância do objeto e. O robô detectado pode ser um inimigo ou um membro

de um time;

Linha 16:

dentro do evento onScannedRobot, o robô atira no inimigo através do comando fire(1). Dentro dos parênteses desse método há o parâmetro correspondente ao firepower do projétil atirado pelo canhão, que varia de 1 a 3 assim como abordado na Subseção 3.3.5. É importante destacar que, ao alterar o parâmetro firepower do método fire() de 1 para 2 ou 3, o diâmetro do projétil aumenta, consequentemente.

4.3 Personalizando o Robô

Para diferenciar os robôs em um campo de batalha, deve-se alterar as cores de cada elemento de seu corpo, inclusive a cor do projétil. Usando o Algoritmo 4.1, pode-se personalizar o robô através do método setColors().

Algoritmo 4.2: MeuPrimeiroRoboCoresAlteradas.java

```
1package robottest;
2import robocode.*;
3import java.awt.Color;
4
5public class MeuPrimeiroRoboCoresAlteradas extends Robot{
6
7   public void run() {
8
9       setColors(Color.blue, Color.red, Color.yellow, Color.white, Color.orange);
10      // setColors(Color bodyColor, Color gunColor, Color radarColor, Color bulletColor, Color scanArcColor)
11      while (true) {
12          ahead(100);
13          turnGunRight(360);
14          back(100);
15          turnGunRight(360);
16      }
17  }
18
19  public void onScannedRobot(ScannedRobotEvent e) {
20      fire(1);
21  }
22}
```

Por exemplo, na Linha 3 do Algoritmo 4.2 a biblioteca java.awt.Color é importada. O pacote java.awt contém todas as classes para manipilação das interfaces com o usuário, gráficos e imagens. A classe Colors é usada para encapsular cores em RGB. Para mais detalhes sobre essa classe, além das cores disponíveis pela mesma, pode ser encontrada na documentação oficial do Java[1] .

Na Linha 9 do Algoritmo 4.2, o método que personaliza os elementos do robô é utilizado. A Linha 10 comentada indica qual elemento corresponde, em ordem, às cores atribuídas na Linha 9. Neste caso, definiu-se a cor azul para o corpo, cor vermelha para a parte inferior do canhão, cor amarela para o radar, cor branca para o projétil e cor laranja para o arco de escaneamento. Entretanto, as cores das rodas e

o canhão não são personalizadas, permanecendo com coloração metálica.

Se o método setColors() tiver argumentos do tipo null (nulo), por exemplo, setColors(null, null, null, null, null), ou se o método não for adicionado no código, a cor padrão definida para o corpo, radar, arco de escaneamento e o canhão será azul, enquanto que o projétil será na cor branca.

Existem alguns métodos, descritos a seguir, utilizados para configurar cores distintas para cada elemento do Robô, ou ainda definir uma só cor para a estrutura do mesmo:

- setAllColors(Color.color): define uma só cor para todos os componentes do robô, ou seja, o corpo, o canhão, o radar, o projétil e o arco de escaneamento;
- setBodyColor(Color.color): define a cor para o corpo do robô. Os elementos restantes terão cores definidas por padrão no Robocode, ou seja, branco para o projétil e azul para os demais;
- setGunColor(Color.color): define uma cor para a parte inferior do canhão. Assim como no método setBodyColor(Color.color), os demais elementos do robô terão cor padrão do Robocode;
- setRadarColor(Color.color): define uma cor da parte inferior do radar. Assim como no método setBodyColor(Color.color), os demais elementos do robô terão cor padrão do Robocode;
- setBulletColor(Color.color): define uma cor para o projétil. Assim como no método setBodyColor(Color.color), os demais elementos do robô terão cor padrão do Robocode;
- setScanColor(Color.color): define uma cor para o arco de escaneamento. Assim como no método setBodyColor(Color.color), os demais elementos do robô terão cor padrão do Robocode.

4.4 Classes Nativas dos Robôs

Apesar de o Robocode ser uma plataforma basicamente orientada a eventos, o mesmo tem como paradigma de programação a orientação a objetos. Assim como indica a Linha 4 e a análise do Algoritmo 4.1, foi mencionado termos como classe, objeto, métodos e herança.

Uma classe pode ser definida como um conjunto de códigos de programação que incluem a definição dos atributos e dos métodos necessários para a criação de um ou mais objetos. Um objeto é uma entidade única que reúne atributos e métodos, ou seja, reúne as propriedades do objeto e as reações aos estímulos que sofre. Um método está relacionado à ação ou comportamento dos objetos. Portanto, é uma função ou um serviço fornecido pelo objeto. A herança está relacionada às hierarquias e às relações entre os objetos, relacionado ao mecanismo em que uma classe filha compartilha automaticamente todos os métodos e atributos da sua classe-mãe.

Percebe-se, portanto, que são necessários conhecimentos prévios em Programação Orientada a Objetos (POO), em especial a definição e sintaxe, na linguagem Java, de uma classe, objeto, atributo, método, construtor, interface, herança e polimorfismo. Esses assuntos podem ser encontrados na documentação oficial do Java [Oracle 2014] ou na apostila oferecida gratuitamente do centro de treinamento Java, a Caelum [Caelum 2014].

Antes de prosseguir nos demais exemplos de código implementados no Robocode, um importante tópico está relacionado com as classes nativas da plataforma, que são previamente codificadas. Na realidade, tratam-se como classes-mãe definem os métodos, construtores e atributos dos robôs, limitando seus comandos em classes mais simples ou atribuindo aos mesmos mais complexidade. Por exemplo, as classes JuniorRobot, Robot, AdvancedRobot e TeamRobot são obrigatoriamente herdadas para os robôs que sejam implementados, assim como indicam as Linhas 4 e 5 dos Algoritmos 4.1 e 4.2.

A seguir são apresentados as principais diferenças e alguns métodos e propriedades específicas de cada classe nativa do Robocode.

4.4.1 JuniorRobot

O JuniorRobot é uma classe nativa utilizada para modelar robôs mais simples. Geralmente é utilizada para fins didáticos no aprendizado de princípios básicos de programação, não demandando, portanto, conhecimentos sobre POO.

A classe JuniorRobot está limitada somente em retornar as coordenadas e movimentos simples do robô, além de informações como o último oponente detectado pelo radar, entre outros. Todos os métodos dessa classe não retornam informações antes de as ações contidas nos mesmos não tiverem sido completadas, com exceção à personalização de cores nos componentes do robô, que podem ser executadas imediatamente durante a execução do código. Vale destacar também que os comandos dessa classe não incluem manipular o movimento do radar, mas somente do canhão e do corpo.

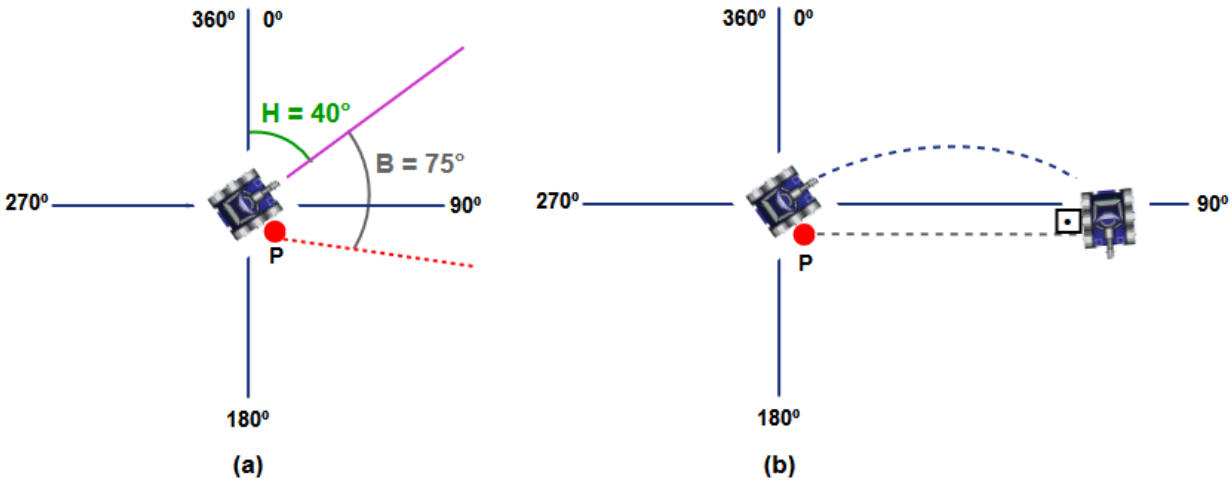

Figura 4.2: Esquema do movimento implementado no Algoritmo 4.3. Em (a) é ilustrado o heading e bearing do corpo do robô e do projétil que o atinge, respectivamente. Já (b) ilustra o movimento realizado após ser atingido.

O Algoritmo 4.3 apresenta um exemplo de codificação simples ao herdar a classe JuniorRobot como indica a Linha 5. Trata-se de um robô que, ao detectar o oponente, o evento onScannedRobot() da Linha 25 é disparado, fazendo girar o canhão em um ângulo na qual o adversário é dectectado, além de atirar nessa mesma direção. Ao ser atingido por um projétil, o evento onHitByBullet() da Linha 36, responsável por executar ações depois que o robô for atingido pelo mesmo, é disparado. Nesse evento, como indica a Linha 38, o robô irá se mover para frente em 100 pixels e será, ao mesmo tempo, rotacionado para a direita, perpendicularmente ao projétil que o atingiu.

A Figura 4.2 (a) ilustra o exemplo do movimento, em que o robô está inicialmente posicionado em uma direção na qual o heading H do corpo vale 40 graus. Um projétil P o atinge em um ângulo B igual a 75 graus, equivalendo ao bearing desse objeto e sendo retornado através da propriedade hitByBulletBearing. Após ser atingido, o robô percorrerá 100 pixels para frente, virando simultaneamente à sua direita com uma angulação de 15 graus, já que 90 – 75 = 15, como indica o cálculo da Linha 38. Forma-se então uma parábola até que o robô permaneça perpendicular à P, assim como mostrado em (b). Esse movimento é uma forma de despistar o canhão do oponente e os futuros projéteis lançados logo após P, por exemplo.

Para implementar o Algoritmo 4.3 e ver o que acontece, deve-se repetir os procedimentos da Seção 4.2, criando uma classe chamada RoboJunior dentro do pacote robottest e adicionando um oponente qualquer através do pacote sample.

Algoritmo 4.3: RoboJunior.java

```
1package robottest;
2import robocode.JuniorRobot;
```

```
3
4
5public class RoboJunior extends JuniorRobot {
6   /**
7    * declaracao do metodo run(), que executa os comandos ate uma nova rodada ser iniciada
8    */
9   public void run() {
10            // configurando cores - verde para o corpo, preto para o canhao e azul para o radar
11            setColors(green, black, blue);
12
13            // laco que executara o movimento do robo enquanto estiver agindo em uma batalha
14            while (true) {
15                  ahead(100); // Move o robo em 100 pixels para frente
16                  turnGunRight(360); // Spin gun around
17                  back(100); // Move o robo em 100 pixels para tras
18                  turnGunRight(360); // Spin gun around
19            }
20      }
21
22      /**
23       * quando um robo detectar seu inimigo, dispara o evento onScannedRobot
24       */
25      public void onScannedRobot() {
26            // gira o canhao no ponto no qual o oponente se encontra
27            turnGunTo(scannedAngle);
28            // Atira no oponente detectado
29            fire();
30      }
31
32      /**
33       * Ao ser atigindo por um projetil, o robo mover-se-a de modo perpendicular ao projetil,
34       * de modo que evite o proximo tiro
35       */
36      public void onHitByBullet() {
37            // Move o robo para frente em 100 pixels e o gira para a direita perpendicularmente ao projetil que o atingiu
38            turnAheadRight(100, 90 - hitByBulletBearing);
39      }
40}
```

Por se tratar de uma classe-mãe que apresenta recursos mais simples e comandos mais intuitivos, pode-se observar que alguns métodos e propriedades são específicos da mesma através do Algoritmo 4.3. Na Linha 9, o método setColors se limita a colorir somente o corpo, o canhão e o radar do robô, diferente da análise no Algoritmo 4.2, no qual herdou-se a classe Robot, que integra métodos mais complexos e menos intuitivos do que a JuniorRobot.

O turnAheadRight, indicado na Linha 38 do Algoritmo 4.2, gira o robô para a direita e o move para frente, ao mesmo tempo. Além disso, a classe JuniorRobot possui propriedades que retornam informações sobre o oponente, como o scannedAngle na Linha 27, que retorna, em graus, o ângulo atual do robô detectado pelo radar.

Propriedades Principais

A seguir são descritos as propriedades principais da classe JuniorRobot. Mais detalhes acerca dessa classe nativa podem ser encontrados na documentação oficial do Robocode[2] .

- energy: propriedade do tipo inteiro que retorna a energia atual do robô;
- robotX: propriedade do tipo inteiro que retorna, em pixels, a localização horizontal atual do robô;
- robotY: propriedade do tipo inteiro que retorna, em pixels, a localização vertical atual do robô;
- heading: propriedade do tipo inteiro que retorna o heading do corpo robô;
- gunHeading: propriedade do tipo inteiro que retorna o heading do canhão do robô;
- heading: propriedade do tipo inteiro que retorna o heading do robô;
- scannedDistance: propriedade do tipo inteiro que retorna a distância do oponente mais próximo. Será retornado um valor válido se o evento onScannedRobot() for disparado. Caso contrário, a propriedade retornará um valor menor que zero;
- scannedBearing: propriedade do tipo inteiro que retorna o ângulo do robô, em graus, mais próximo escaneado em relação ao corpo do robô. Será retornado um valor válido se o evento onScannedRobot() for disparado;
- scannedVelocity: propriedade do tipo inteiro que retorna a velocidade do oponente mais próximo escaneado. Será retornado um valor válido se o evento onScannedRobot() for disparado. Caso contrário, a propriedade retornará -99. Quando for retornado um valor igual a zero, por exemplo, significa que o robô está parado;
- scannedHeading: propriedade do tipo inteiro que retorna o heading do oponente mais próximo escaneado. Será retornado um valor válido se o evento onScannedRobot() for disparado. Caso contrário, a propriedade retornará um valor menor que zero;
- scannedEnergy: propriedade do tipo inteiro que retorna a energia do oponente mais próximo escaneado. Será retornado um valor válido se o evento onScannedRobot() for disparado;
- hitRobotAngle: propriedade do tipo inteiro que retorna o ângulo, em graus, formado pela colisão entre o robô e o oponente. Será retornado um valor válido se o evento onHitRobot() for disparado;
- hitRobotBearing: propriedade do tipo inteiro que retorna o bearing do oponente com relação ao corpo do robô codificado. Será retornado um valor válido se o evento onHitRobot() for disparado;
- hitWallAngle: propriedade do tipo inteiro que retorna o ângulo, em graus, formado pela colisão entre o robô e a parede. Será retornado um valor válido se o evento onHitWall() for disparado;
- hitWallBearing: propriedade do tipo inteiro que retorna o bearing da parede com relação ao corpo do robô codificado. Será retornado um valor válido se o evento onHitWall() for disparado.

Métodos Principais

A seguir são descritos os métodos principais da classe JuniorRobot.

- fire(): atira um projétil com firepower igual a 1;
- fire(n): atira um projétil com firepower igual a n, em que n varia de 0.1 a 3;
- onHitRobot(): método chamado pelo jogo quando um robô colide com outro;
- onHitWall(): método chamado pelo jogo quando um robô colide com a parede;
- turnAheadLeft(int distancia, int angulo): semelhante ao método turnAheadRight(int distancia, int angulo) utilizado no Algoritmo 4.3, com a diferença de, ao invés de virar para a direita, o robô rotaciona para a esquerda. Os parâmetros distancia e angulo são mensurados em pixels e graus, respectivamente;

- turnBackLeft(int distancia, int angulo): faz o robô andar para trás em uma distancia medida em pixels e virar para a esquerda em um angulo mensurado em graus, simultaneamente;
- turnBackRight: faz o robô andar para trás em uma distancia medida em pixels e virar para a direita em um angulo mensurado em graus, simultaneamente;
- turnLeft(int angulo): faz o robô rotacionar para a esquerda em uma angulação medida em graus;
- turnRight(int angulo): faz o robô rotacionar para a direita em uma angulação medida em graus;
- turnTo (int angulo): faz o robô rotacionar em um ângulo especificado pelo parâmetro angulo;
- turnGunLeft(int angulo): faz com que o canhão do robô rotacione para a direita em uma angulação medida em graus;
- turnGunRight(int angulo): faz com que o canhão do robô rotacione para a direita em uma angulação medida em graus;
- turnGunTo(int angulo): faz com que o canhão do robô rotacione em um ângulo especificado pelo parâmetro angulo;

4.4.2 Robot

A classe-mãe Robot permite modelar robôs mais complexos, na qual envolvem métodos que retornam informações mais precisas a respeito do mundo durante o jogo do que um simples número inteiro, assim como abordado na Subseção 4.4.1. Por exemplo, direfente da classe JuniorRobot, propriedades básicas que retornam a energia, a velocidade e o heading do oponente deixam de existir em Robot. Portanto, passa-se a considerar o desenvolvimento de robôs que preveem diferentes propriedades físicas do inimigo através de algoritmos complexos, sendo inicialmente codificadas pelo programador. Para isso, conhecimentos prévios em POO são necessários.

Algoritmo 4.4: PaintedRobot.java

```
1package robottest;
2
3import java.awt.Color;
4import java.awt.Graphics2D;
5import robocode.*;
6
7public class PaintedRobot extends Robot{
8
9   /**
10    * declaracao do metodo run(), que executa os comandos ate uma nova
11    * rodada ser iniciada
12    */
13   public void run() {
14       while (true) {
15       // rotaciona o radar de 0 a 360 graus de modo a detectar os robos em campo de batalha enquanto o loop while retornar true
16        turnRadarRight(360);
17       }
18   }
19
20   // declara as variaveis do tipo inteiro relacionadas as coordenadas X e Y do oponente
21   int scannedX = Integer.MIN_VALUE;
22   int scannedY = Integer.MIN_VALUE;
```

```
23
24  /**
25   * quando um robo detectar seu inimigo, dispara-se evento onScannedRobot
26   */
27  public void onScannedRobot(ScannedRobotEvent e) {
28   // Calcula o angulo do robo escaneado
29   double angle = Math.toRadians((getHeading() + e.getBearing()) % 360);
30
31   // Calcula as coordenadas do robo escaneado
32   scannedX = (int)(getX() + Math.sin(angle) * e.getDistance());
33   scannedY = (int)(getY() + Math.cos(angle) * e.getDistance());
34
35  }
36
37  /**
38   * Desenha um retangulo que sera projetado sobre o ultimo robo escaneado
39   */
40  public void onPaint(Graphics2D g) {
41      // configura a cor do objeto g em vermelho
42      g.setColor(new Color(0xff, 0x00, 0x00, 0x80));
43
44      // Desenha uma linha do robo ao oponente escaneado entre as componentes (scannedX, scannedY) e (getX(), getY())
45      g.drawLine(scannedX, scannedY, (int)getX(), (int)getY());
46
47      // Desenha um quadrado sobre o oponente escaneado na posicao (scannedX - 20, scannedY - 20), com largura e altura de 40 pixels
48      g.fillRect(scannedX - 20, scannedY - 20, 40, 40);
49  }
50}
```

O Algoritmo 4.4 apresenta um exemplo que herda a classe Robot, na qual pode-se prever a localização do oponente através de suas coordenadas, permitindo também conhecer o funcionamento do evento onPaint(), que permite desenhar no campo de batalha diferentes recursos através do objeto g. Esse evento permitirá desenhar um quadrado sobre o oponente a partir do momento que as coordenadas do mesmo são calculadas pelo robô codificado, identificando que o radar o localizou.

Uma vez que o robô PaintedRobot é implementado, faz-se necessário compreender como o inimigo é localizado a partir das suas coordenadas calculadas, além de demonstrar como se pode desenhar diferentes gráficos no campo de batalha para destacar algum recurso ou ação do robô codificado conforme as necessidades definidas pelo programador.

Enquanto a batalha estiver sendo executada, o robô sempre irá rotacionar o radar em 360 graus para a direita de modo a encontrar o inimigo, assim como indica a Linha 16. Portanto, a manipulação do radar passa a ser possível através do método turnRadarRight(double angulo), já que na classe JuniorRobot permitia-se apenas manipular o movimento do canhão e do corpo. Além disso, o parâmetro angulo, mensurado em graus, é do tipo double, permitindo que o método seja executado com maior precisão.

As Linhas 21 e 22 declaram as variáveis scannedX e scannedX, relacionadas à coordenada X e Y do inimigo, respectivamente. Ambas são do tipo inteiro e inicializadas com um valor mínimo através da constante Integer.MIN_VALUE , equivalendo ao valor mínimo da

faixa de números do tipo inteiro, equivalente a -2.147.483.648.

Deseja-se calcular o ângulo do inimigo, sendo possível fazê-lo através da Equação 3.7. Na Linha 29, pode-se obter o heading do robô e o bearing do inimigo através dos métodos getHeading() e getBearing(), respectivamente. Como essas propriedades retornam valores em graus, essas são convertidas para radianos através do método Math.toRadians(), pois as funções de seno e cosseno da biblioteca Math do Java requerem parâmetros em radianos.

O termo que corresponde ao resto de 360 na Linha 29 representa a normalização de ângulos do Java, permitindo que seja atribuído na variável angle somente um valor menor ou igual a 360. Isto significa que, caso a soma de heading e bearing seja maior que 360, isso representará mais que uma volta no círculo trigonométrico. Deseja-se então obter apenas o equivalente a essa soma, representado pelo ângulo excedente localizado entre 0 a 360 graus.

As Linhas 32 e 33 calculam a coordenada X e Y do inimigo escaneado pelo robô, atribuindo o valor obtido à variável scannedX e scannedX, respectivamente. O cálculo pode ser compreendido através da Figura 4.3, na qual os eixos x e y representam o campo de batalha, tendo como origem a coordenada (0,0) como especificado na Subseção 3.3.1. A ilustração da Figura 4.3 mostra a relação de diferentes valores envolvidos na obtenção das coordenadas do inimigo, relacionados com as Linhas 29, 32 e 33 do Algoritmo 4.4 e serão descritas a seguir:

- angle = (h) + (b): referente à soma dos ângulos heading e bearing a qual resulta no ângulo que o oponente se encontra;
- D = e.getDistance(): equivalendo à distância entre o centro do corpo do robô e do inimigo e representa a hipotenusa do triângulo-retângulo DDyDx;
- Dx = D * sin(angle): equivalendo ao valor da componente x da distância D;
- Dy = D * cos(angle): equivalendo ao valor da componente y da distância D;
- Xr = getX() = coordenada x do robô codificado;
- Yr = getY() = coordenada y do robô codificado;
- Xi = scannedX = coordenada x do inimigo;
- Yi = scannedY = coordenada y do inimigo.

Deseja-se determinar a localização (Xi,Y i) do oponente, dado pelo cálculo das Linhas 32 e 33 do Algoritmo 4.4. A partir do angle, deve-se somar as componentes Dx e Dy com as coordenadas Xr e Y r do robô, respectivamente. Uma vez que as componentes Dx e Dy têm como origem o centro do robô PaintedRobot, a soma com suas coordenadas é necessária para considerar a origem (0,0) dos eixos x e y do campo de batalha.

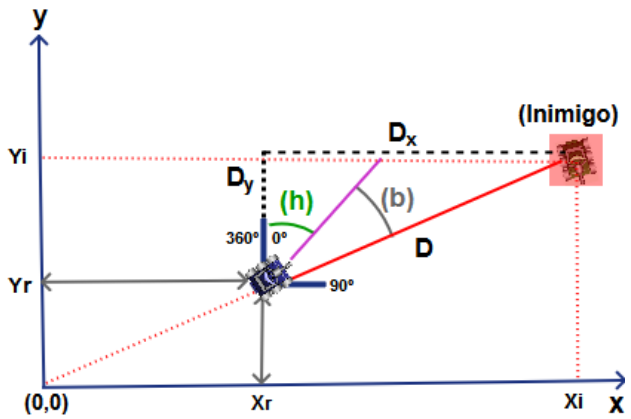

Figura 4.3: Esquema de Localização do Inimigo Através das Suas Coordenadas.

A Linha 40 do Algoritmo 4.4 representa o evento onPaint(), na qual é instanciado o objeto g do tipo Graphics2D. Esse evento permite a manipulação de gráficos bidimencionais no campo de batalha. Nas Linhas 45 e 48 são implementados, respectivamente, um quadrado sobre o inimigo e uma linha que descreve a distância entre o mesmo e o robô PaintedRobot, ambas em vermelho, destacando que as coordenadas do oponente foram localizadas, assim como ilustrado na Figura 4.3. Maiores detalhes sobre a classe Graphics podem ser consultados na documentação oficial do Java[3] .

4.4.2.1 Métodos Principais

A seguir são descritos os métodos principais da classe Robot. Mais detalhes acerca dessa classe nativa podem ser encontrados na documentação oficial do Robocode[4] .

- ahead(double distancia): move o robô para frente em uma distância do tipo double e mensurada em pixels;
- back(double distancia): move o robô para trás em uma distância do tipo double e mensurada em pixels;
- getBattleFieldWidth(): retorna a largura do campo de batalha, mensurado em pixels;
- getBattleFieldHeight(): retorna a altura do campo de batalha, mensurado em pixels;
- getHeight(): retorna a altura do campo de batalha, mensurado em pixels;
- getWidth(): retorna a altura do campo de batalha, mensurado em pixels;
- getName(): método do tipo string que retorna o nome do robô codificado;
- getGunCoolingRate(): método do tipo double que retorna a taxa temporal com o qual o canhão "esfria", relacionado à propriedade Gun Heat descrita na Subseção 3.3.5;
- getGunHeat(): método do tipo double que retorna o valor da propriedade física Gun Heat;
- getGunHeading(): método do tipo double que retorna o heading do canhão do robô codificado;
- getRadarHeading(): método do tipo double que retorna o heading do radar do robô codificado;
- getTime(): método do tipo long que retorna o tempo da rodada atual, na qual o tempo equivale ao turno atual relacionado à batalha;
- getEnergy(): método que retorna a energia atual do robô;
- getVelocity(): método do tipo double que retorna a velocidade do robô codificado;
- scan(): método que executa o escaneamento do robô, disparando imediatamente o evento onScannedRobot(). O seu uso é favorável também quando se deseja interromper esse evento, de modo a executá-lo novamente desde o início;
- setAdjustGunForRobotTurn(boolean independent): método que permite que o canhão se torne independente do movimento do corpo do robô em um turno dado em tempo de execução deste método. O valor padrão para o parâmetro independent é false. Por exemplo, se o método em questão ser configurado como true e o método turnRight(90) for implementado para virar à direita em 90 graus, e logo em seguida, apenas o corpo rotacionará, mantendo o canhão na posição anterior. Vale ressaltar que em cada turno o robô pode executar uma ação diferente;
- setAdjustRadarForRobotTurn(boolean independent): método que permite que o radar se torne independente do movimento do corpo do robô em um turno dado em tempo de execução deste método. O valor padrão para o parâmetro independent é false;
- setAdjustRadarForGunTurn(): método que permite que o radar passe a ser independente do movimento do canhão em um turno dado em tempo de execução deste método. O valor padrão para o parâmetro independent é false. Por exemplo, se o método em questão for configurado como true e o método turnGunRight(90) for implementado para virar à direita em 90 graus e o logo em seguida, apenas o canhão rotacionará, mantendo o radar na posição anterior;
- stop(): método que interrompe todos os movimentos do robô;
- onBulletHit() e onBulletHitBullet(): evento disparado quando um dos projéteis lançados pelo canhão do robô atingem outro robô e projétil, respectivamente;
- onBulletMissed(): evento disparado quando um dos projéteis lançados pelo canhão do robô são perdidos, atingindo a parede, por exemplo;

- onDeath(): evento disparado quando o robô codificado morre;
- onHitByBullet(): evento disparado quando o robô codificado é atingido por um projétil;
- onHitRobot(): evento disparado quando o robô colide com outro;
- onHitWall(): evento disparado quando o robô colide com a parede;
- onRobotDeath(): evento disparado quando outros robôs morrem;
- onWin(): evento disparado quando o robô codificado ganha uma batalha;
- onRoundEnded()
- onBattleEnded() e
- onBattleEnded(): evento disparado quando uma rodada ou batalha é finalizada, respectivamente.

4.4.3 AdvancedRobot

A classe AdvancedRobot é a mais avançada em questões de complexidade de robôs. Trata-se de uma classe nativa que herda os métodos e eventos da classe Robot, incluindo funcionalidades adicionais que permitem a implementação de algoritmos mais complexos.

Uma das peculiaridades aplicadas nessa classe inclui o uso de métodos não-bloqueados (setters), que podem ser executados a qualquer momento. Isto significa que, uma vez executado um método desse tipo, o Robocode irá interromper a ação atual do robô codificado para priorizar esse metódo. Além disso, enquanto que na classe Robot executa somente um método por turno, a classe AdvancedRobot permite executar vários métodos não-bloqueados em um mesmo instante do turno atual.

É importante destacar que somente a classe AdvancedRobot permite que haja danos causados pela colisão do robô com a parede o campo de batalha, isto é, a propriedade que retira a energia do robô que colide com a parede não são implementados nas classes JuniorRobot e Robot. Uma batalha com esse tipo de dano ativo causa complexidade no movimento do robô, pois o mesmo tem de agir com cautela para não ter a energia comprometida. Sendo assim, torna-se necessário, mas não obrigatório, o desenvolvimento de um algoritmo que evite a colisão do robô com a parede para melhor gerenciar sua energia durante uma batalha.

O funcionamento de métodos não-bloqueados é mostrado através do Algoritmo 4.5. Esse algoritmo é capaz de golpear o oponente através do movimento Ramming abordado no Subseção 3.3.7, atirando ao mesmo tempo com o canhão direcionado para o bearing do inimigo com firepower máximo, ou seja, igual a 3.0. Além disso, o robô AdvancedBearingBot imprime mensagens em seu console indicando informações de heading e bearing a cada turno.

Algoritmo 4.5: AdvancedBearingBot.java

```
1package robottest;
2
3import robocode.*;
4
5public class AdvancedBearingBot extends AdvancedRobot {
6
7       public void run() {
8           //ajusta o radar para ser rotacionar independentemente ao movimento do canhao
9               setAdjustRadarForGunTurn(true);
10                while (true) {
11                      // rotaciona o radar em 360 graus ate encontrar o oponente
12                      setTurnRadarRight(360);
```

```
13                  //metodo que executa acoes pendentes ou continua executando acoes que estao sendo processadas.
14                  execute();
15              }
16      }
17
18      public void onScannedRobot(ScannedRobotEvent e) {
19
20              // imprimi mensagens na sua tela do console sobre o heading e bearing atual
21              System.out.println("Detectei um robo com bearing " + e.getBearing() + "\n");
22              System.out.println("Meu heading e " + getHeading() + "\n");
23
24              // rotaciona seu corpo para a direita em um angulo igual ao bearing do robo
25              setTurnRight(e.getBearing());
26              // atira com seu canhao com firepower maximo
27              setFire(3);
28              // anda na direcao do inimigo para golpea-lo
29              setAhead(e.getDistance());
30      }
31}
```

Na Linha 9, o método não-bloqueado setAdjustRadarForGunTurn(true) é responsável por ajustar o radar para se mover de forma independente do canhão. Por padrão, assim como descrito na Seção 3.2, o radar está acoplado no canhão, sendo que, caso o radar não realize nenhum movimento em um dado turno, esse acompanha o canhão ao efetuar uma rotação de 90 graus, por exemplo. Para evitar que isso ocorra, deve-se declarar o método setAdjustRadarForGunTurn() como true para que o radar se torne independente dos movimentos efetuados pela arma do robô.

Quando a batalha é inicializada, o radar é ajustado para rotacionar em 360 graus para a direita, assim como indicado na Linha 12. Na Linha 14, o método execute() é responsável por executar todos os métodos não-bloqueados de uma só vez durante a execução de uma batalha, ou seja, enquanto o loop retornar verdadeiro dentro do método run().

Os métodos das Linhas 9, 12, 25, 27 e 28 são setters e podem ser executados a qualquer momento. A classe nativa executa múltiplas tarefas em ordem definida conforme as necessidades do robô. Entretanto, a classe Robot, dispõe as ações do robô em uma fila e as executa de maneira sequencial.

Quando o evento onScannedRobot é disparado, as Linhas 21 e 22 do Algoritmo 4.5 imprimem, respetivamente, informações do heading e bearing, sendo atualizadas a cada turno. Na Linha 25, o método setTurnRight(e.getBearing()) rotaciona o corpo do robô para a direita com um ângulo igual ao bearing do inimigo, atirando com firepower máximo assim como está indicado na Linha 27.

Os movimentos básicos de um robô são indicados pela Linha 29, na qual o robô anda para frente em uma distância entre o robô e o oponente para golpeá-lo.

4.4.3.1 Métodos Principais

A seguir são apresentados alguns dos métodos e eventos da classe AdvancedRobot. Mais detalhes acerca dessa classe podem ser consultados na documentação oficial do Robocode[5] .

- getGunHeadingRadians(), getHeadingRadians() e getRadarHeadingRadians(): retorna o heading do canhão, do corpo e do radar em radianos, respectivamente;
- setBack(double distancia): ajusta o robô para andar para trás em uma distância mensurada em pixels;
- setTurnLeft(double angulo): ajusta o robô para rotacionar para esquerda em um ângulo mesurado em graus;
- setMaxVelocity(double VelocMaxima): ajusta a velocidade máxima do robô, mensurado em pixels;
- setStop(): ajusta o robô para interromper seus movimentos;
- setTurnGunLeft(double angulo): ajusta o canhão para rotacionar para a esquerda, mensurada em graus;
- setTurnGunLeftRadians(double angulo): ajusta o canhão para rotacionar para a esquerda, mensurada em radianos;
- setTurnGunRight(double angulo): ajusta o canhão para rotacionar para a direita, mensurada em graus;
- setTurnGunRightRadians(double angulo): ajusta o canhão para rotacionar para a direita, mensurada em radianos;
- setTurnLeft(double angulo): ajusta o corpo do robô para rotacionar para a esquerda, mensurada em graus;
- setTurnLeftRadians(double angulo): ajusta o corpo do robô para rotacionar para a esquerda, mensurada em radianos;
- setTurnRadarLeft(double angulo): ajusta o radar para rotacionar para a esquerda, mensurada em graus;
- setTurnRadarLeftRadians(double angulo): ajusta o radar para rotacionar para a esquerda, mensurada em radianos.

4.4.4 TeamRobot

É possível criar um time de robôs através da classe TeamRobot. Em geral, pode-se implementar tanto um time centralizado quanto descentralizado e realizar ações de forma cooperada através da troca de mensagens.

O envio das mensagens de um robô são efetuadas por duas formas:

- individual, na qual uma mensagem é enviada para um só robô;
- broadcast, em que todos os robôs do time recebem uma mensagem. Além disso, essa classe herda métodos e eventos das classes AdvancedRobot e Robot, permitindo implementar robôs com complexidade avançada. Na verdade, trata-se de uma classe do tipo AdvancedRobot, que por sua vez herda a classe Robot.

A classe TeamRobot permite definir diferentes tipos de times. Em uma arquitetura centralizada, um time pode ser composto por:

- Um líder e quatro droids. O líder é o robô que utiliza o radar para realizar o escaneamento do campo de batalha. O droid não possui radar, estando submisso às ordens do líder, além de possuir 20 unidades adicionais de energia;
- Uma combinação de cinco robôs, como por exemplo um líder, dois droids e dois robôs normais, no qual os quatro últimos ficam submissos às ordens do líder.

Em uma arquitetura descentralizada o time pode ser composto por:

- Cinco robôs de uma mesma classe;
- Cinco robôs de classes diferentes.

Uma equipe de robôs homogêneos são executados em uma mesma classe e por isso possem características iguais. Um time com classes diferentes estão relacionados a robôs heterogêneos e podem ser modelados com algoritmos diferentes de modo a executarem ações distintas em uma batalha.

4.4.4.1 Métodos Principais

A seguir são apresentados os métodos e eventos da TeamRobot. Mais detalhes acerca dessa classe podem ser consultados na documentação oficial do Robocode[6] .

- broadcastMessage(Serializable message): método que envia uma mensagem do tipo broadcast para todos os robôs de um time;
- getMessageEvents(): retorna um vetor com todos os eventos relacionado à troca de mensagens entre um time que estão atualmente alocados em uma fila;
- getTeammates(): retorna o nome de todos os integrantes de um time;
- isTeammate(String nome): método do tipo boolean que retorna true se o robô pertencer a um time ou false caso contrário. Tem como parâmentro o nome do robô que se deseja efetuar a verificação;
- onMessageReceived(MessageEvent event): evento disparado quando as mensagens são recebidas;
- sendMessage(String name, Serializable message): envia uma mensagem para somente um integrante do time.

Referências Bibliográficas

[Caelum 2014] CAELUM. Apostila do curso fj-11: Java e orientacao a objetos. In: . Disponível em: <http://www.caelum.com.br/apostila-_java-_orientacao-_objetos/>. Acesso em 20 de janeiro de 2014: [s.n.], 2014.

[Eclipse 2014] ECLIPSE. Eclipse. In: . Disponível em: <https://eclipse.org/>. Acesso em 31 de Dezembro de 2014: [s.n.], 2014.

[Gade et al. 2003] GADE, M. et al. Applying machine learning to robocode. In: . Disponível em: <http://www.dinbedstemedarbejder.dk/Dat3.pdf>. Acesso em 30 de Setembro de 2014: Department of Computer Science - Aalborg University, 2003.

[Hong e Cho 2004] HONG, J.; CHO, S. Evolution of emergent behaviors for shooting game characters in robocode. Evolutionary Computation, p. 634–638, 2004.

[Jannace 2011] JANNACE, M. Multi agent systems and robocode. In: . [S.l.: s.n.], 2011.

[Larsend 2010] LARSEND, F. N. Robocode: Game physics. In: . Disponível em: <http://robowiki.net/wiki/Robocode/Game_Physics>. Acesso em 05 de Agosto de 2014: [s.n.], 2010.

[Nielsen e Jensen 2010] NIELSEN, J. L.; JENSEN, B. F. Modern ai for games: Robocode. In: . Disponível em: <http://www.jonnielsen.net/RoboReportOfficial.pdf>. Acesso em 30 de Setembro de 2014: IT University of Copenhagen, 2010.

[Oracle 2014] ORACLE. Java se. In: . Disponível em: <http://www.oracle.com/technetwork/pt/java/javase/overview/index.html>. Acesso em 31 de Dezembro de 2014: [s.n.], 2014.

[Oracle 2014] ORACLE. Lesson: Object-oriented programming concepts. In: . Disponível em: <http://docs.oracle.com/javase/tutorial/java/concepts/>. Acesso em 20 de janeiro de 2014: [s.n.], 2014.

[Oracle 2014] ORACLE. Netbeans ide. In: . Disponível em: <https://netbeans.org/>. Acesso em 31 de Dezembro de 2014: [s.n.], 2014.

[Robocode 2013] ROBOCODE. Readme for robocode. In: . Disponível em: <http://robocode.sourceforge.net/docs/ReadMe.html>. Acesso em 31 de Dezembro de 2014: [s.n.], 2013.

[Robocode 2014] ROBOCODE. Robocode 1.9.2.0 api. In: . Disponível em: <http://robocode.sourceforge.net/docs/robocode/index.html?overview-_summary.html>. Acesso em 01 de Agosto de 2014: [s.n.], 2014.

[Uchida, Kobayashi e Watanabe 2003] UCHIDA, Y.; KOBAYASHI, K.; WATANABE, K. A study of battle strategy for the robocode. In: . Fukui University, Japan: SICE Annual Conference in Fukui, 2003.

- Ir para o topo
- Home
- Download dos Códigos
- Sobre
- UFABC
- Robocode: Site Oficial

Manual Complementar do Projeto de Pesquisa: Robocode: Um Estudo Acerca do Software

Escrito por Nathalia Paula, aluna do Curso de Engenharia de Instrumentação, Automação e Robótica da UFABC.

Orientação: Prof. Maria das Graças Bruno Marietto e Prof. Wagner Tanaka Botelho

Laboratório de Robótica e Sistemas Inteligentes | UFABC

Baseado na plataforma Bootstrap <3.