Licenciatura em Engenharia Informática e Multimédia
1º Semestre Letivo 2019/2020

Fundamentos de Sistemas Operativos
1º Trabalho Prático

Docente:
Jorge Pais

Autores:
42341, Ana Coelho, 31N
42346, Luís Guimarães, 31N
42356, Érica Pereira, 31N

Lisboa, 28 de outubro de 2019

Índice

1. Objetivos/Introdução 4
2. Desenvolvimento do Trabalho 5
   2.1. Interfaces Gráficas 5
      2.1.1. Classe “GUIDancarino” 5
      2.1.2. Classe “GUICoreografo” 6
      2.1.3. Classes “BD” e “BD2” 7
   2.2. Canal de comunicação 8
      2.2.1. Classe “MyMessage” 8
      2.2.2. Classe “CanalComunicacao” 8
   2.3. Diagrama de classes do Dançarino 10
   2.4. Diagrama de classes do Coreógrafo 11
   2.5. Diagrama de atividades do Dançarino 12
   2.6. Diagrama de atividades do Coreógrafo 15
   2.7. Sincronização no acesso ao Canal de Comunicação 18
   2.8. Resultados práticos da interação entre processos e o robot 19
3. Conclusões 20
4. Bibliografia 21
5. Anexos 22
   5.1. Anexo A – Classe “BD” 22
   5.2. Anexo B – Classe ”GUIDancarino” 24
   5.3. Anexo C – Classe “Dancarino” 29
   5.4. Anexo D – Classe “BD2” 32
   5.5. Anexo E – Classe “GUICoreografo” 34
   5.6. Anexo F – Classe “Coreografo” 37
   5.7. Anexo G – Classe “MyMessage” 40
   5.8. Anexo H – Classe “CanalComunicacao” 41
   5.9. Anexo I – Classe “MyRobotLego” 43

Índice de Figuras

Figura 1 - Interface gráfica "GuiDancarino" _____ *6*
Figura 2 - Interface gráfica "GUICoreografo" _____ *7*
Figura 3 - Diagrama de classes (UML): Dancarino _____ *10*
Figura 4 - Diagrama de classes (UML): Coreógrafo. _____ *11*
Figura 5 - Autómato "automatoDancarino" _____ *12*
Figura 6 - Autómato "switchComandos" _____ *14*
Figura 7 - Autómato "automatoCoreografo" _____ *15*
Figura 8 - Autómato "automatoComandos" _____ *17*

Índice de Tabelas

Tabela 1 - Valores da variável Ordem correspondentes a cada comando do robot 8
Tabela 2 - Classes JAVA auxiliares da comunicação. 8

1. Objetivos/Introdução

Este trabalho pretende demonstrar o funcionamento da programação multiprocesso, ilustrando a gestão e sincronismo de processos em JAVA através de comunicação entre processos recorrendo a memória partilhada.

Foram desenvolvidos dois processos – um processo implementa o comportamento COREÓGRAFO, o segundo processo implementa o comportamento DANÇARINO.

Os dois processos criados seguem o modelo Produtor-Consumidor:

- O comportamento COREÓGRAFO é um processo que produz mensagens. As mensagens são geradas de 500 em 500 milissegundos quando o comportamento se encontra ativo, para serem lidas pelo consumidor;

- O comportamento DANÇARINO é um processo que consome as mensagens que o Coreógrafo produz e guarda-as em memória local, para depois as enviar, como comandos de movimento, para o robot[1].

A comunicação entre os dois processos recorre à utilização de memória partilhada através da classe JAVA *MappedByteBuffer*:

Tanto o Dançarino como o Coreógrafo fornecem GUI's (interfaces gráficas) que permitem aos utilizadores interagir com os processos:

- A GUI do Coreógrafo permite ativar o comportamento Coreógrafo e gerar comandos em quantidades variáveis (1, 16, 32, ilimitados) ou parar a produção de comandos. Também contém um elemento *TextArea* que ilustra os comandos gerados;

- A GUI do Dançarino que permite ativar o comportamento Dançarino ou interagir diretamente com o robot através de cinco botões de controlo de movimento. Tal como com a GUI do Coreógrafo, contém um elemento *TextArea* que ilustra os comandos recebidos.

As GUI's foram criadas em JAVA Swing utilizando o editor gráfico WindowBuilder para o Eclipse.

[1] RobotLegoEV3 - http://www.portugal-didactico.com/legoeducation/2-o-e-3-o-ciclo-2/lego-mindstorms-ev3/

2. Desenvolvimento do Trabalho

2.1. Interfaces Gráficas

Em primeiro lugar, foram desenvolvidas duas interfaces gráficas em Java Swing para controlar o robot RobotLegoEV3, utilizando a biblioteca fornecida pelo docente. Esta biblioteca possuí os métodos responsáveis pelo controlo do robot.
As duas interfaces gráficas criadas denominam-se "GUIDancarino" e "GUICoreografo".

2.1.1. Classe "GUIDancarino"

A classe "GUIDancarino" implementa os métodos public void run(), public static void main(String[] args), public GUIDancarino(), public void myPrint(String text) e private void activateButton(boolean onOff).

Para o desenvolvimento da sua interface gráfica (Figura 1), foram criados e implementados, no método construtor da classe, os seguintes elementos:

- um elemento *JLabel* e um *JTextField* "Robot", onde se pode definir o nome do robot, no caso deste trabalho "EV7";
- um elemento *JRadioButton* "On/Off", que dá acesso à comunicação Bluetooth entre o computador e o robot, podendo começar a comunicação ou terminar a mesma;
- três elementos *JLabel* com os respetivos *JTextField* com os nomes "Raio", "Angulo" e "Distancia", que possibilita a definição dos valores que irão controlar as ações do robot;
- cinco botões com instruções - "Frente", "Esquerda", "Parar", "Direita" e "Retaguarda" do tipo *JButton.* Estes elementos permitem que o utilizador possa controlar o movimento do robot. Os botões podem ser activados ou desactivados[2] conforme o elemento "On/Off" esteja selecionado ou não*;*
- dois elementos *JCheckBox*, "Activar" e "Debug", a primeira permite ativar ou desativar o comportamento e a segunda ativa a o debug[3] na consola;
- um elemento *JButton* "Clear", que apaga tudo o que já foi apresentado na consola.

[2] Ativar/desativar botões – através do método activateButtons(boolean onOff), colocam-se os botões num estado enable ou disable, conforme o valor do booleano passado no parâmetro: caso o booleano onOff esteje a "true", os botões ficam enabled, caso contrário, se estiver a "false", os botões ficam disabled.
[3] Debug - fornece informação relevante, através do método *myPrint()* da classe "GUIDancarino", para a deteção e correção de erros da aplicação.

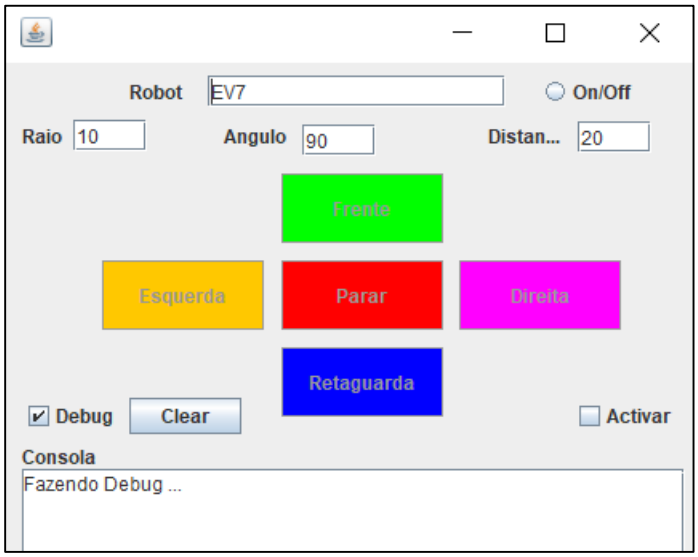

Figura 1 - Interface gráfica "GuiDancarino"

2.1.2. Classe “GUICoreografo”

A classe “GUICoreografo” implementa os métodos public void run(), public static void main(String[] args), public GUICoreografo(), public void myPrint(String text) e private void activateButton(boolean onOff).

Para o desenvolvimento da sua interface gráfica (Figura 2), foram criados e implementados, no método construtor da classe, os seguintes elementos:

- um elemento *Jlabel* e um *JTextArea* “Ultimos 10 comandos” onde são apresentados os últimos comandos enviados;
- um elemento *JButton* “Clear”, que apaga tudo o que já foi apresentado na consola;
- cinco elementos de tipo *JButton* denominados “Gerar 1 comando”, “Gerar 16 comandos”, “Gerar 32 comandos”, “Gerar comandos ilimitados” e “Parar comandos”. Os primeiros quatro botões servem para enviar um dado número de comandos e o último para parar o envio dos comandos;
- Um elemento *JCheckBox* “Activar”, que permite ativar ou desativar o comportamento. Quando ativo, ativa também os cinco botões a baixo e quando desativo, desativa-os.

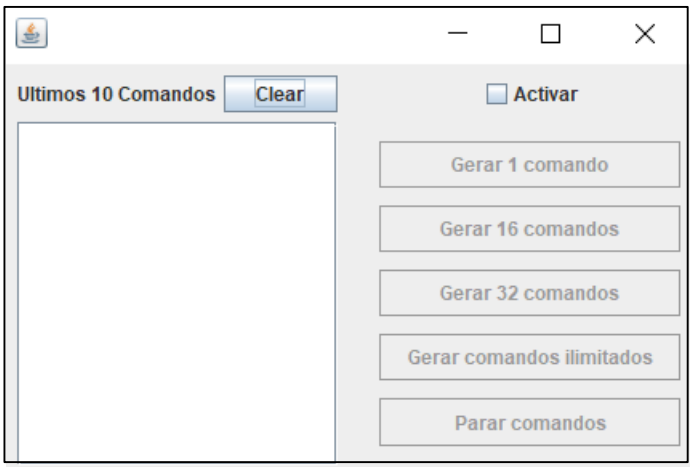

Figura 2 - Interface gráfica "GUICoreografo"

2.1.3. Classes “BD” e “BD2”

As classes “BD” e “BD2” são classes auxiliares de variáveis das classes “Dancarino” e “Coreografo”, passadas como parâmetro nos construtores das classes “GUIDancarino” e “GUICoreografo”, respetivamente.

As classes “BD” e “BD2” possuem todas as variáveis necessárias ao controle do robot (ex. classe “BD”: RobotLegoEV3 robot, int distancia, etc.; ex. classe “BD2”: boolean pararComandos, etc.) podendo, através destas classes, obter ou mudar os seus valores, por métodos de *getter* e *setter*.

É com o auxílio destas classes que a “comunicação” entre as classes “GUIDancarino” e “Dancarino” e as classes “GUICoreografo” e “Coreografo” é possível.

2.2. Canal de comunicação

2.2.1. Classe "MyMessage"

A classe "MyMessage" é uma classe que auxilia na implementação do formato da mensagem a adotar. Nesta classe criaram-se duas variáveis privadas de tipo *int* – a variável *numero,* que identifica a cardinalidade da mensagem enviada, e a variável *ordem,* que irá dar os valores correspondentes aos comandos do robot. Os valores da variável *ordem* são:

Ordem	Comandos do robot
0	Parar(false)
1	Reta(10)
2	CurvarDireita(0, 45)
3	CurvarEsquerda(0, 45)
4	Reta(-10)
5	Parar(true)

Tabela 1 - Valores da variável Ordem correspondentes a cada comando do robot

A classe "MyMessage" também implementa *getter*'s e *setter*'s, para cada uma das variáveis número e ordem.

2.2.2. Classe "CanalComunicacao"

Para a classe "CanalComunicacao" utilizando memória partilhada suportada pela classe JAVA *MappedByteBuffer.* Para isso, o canal de comunicação tem de obedecer ao seguinte modelo fornecido pelo docente:

buffer: *MappedByteBuffer*
fc: *FileChannel*
f: *File* "Comunicacao.txt"

Tabela 2 - Classes JAVA auxiliares da comunicação.

O modelo *MappedByteBuffer* mapeia numa área de memória virtual o conteúdo de um ficheiro. O conteúdo desta memória pode ou não estar disponível na memória principal do processo, portanto o tempo de acesso aos dados pode variar um pouco com a utilização da memória mapeada.

Na classe "CanalComunicacao" vamos criar um ficheiro que irá estar ligado a um canal de comunicação de leitura e escrita, esse canal vai mapear para a memória o conteúdo do ficheiro. Graças a esse canal o processo Coreógrafo pode comunicar com o processo Dançarino.

O buffer criado segue um modelo circular com possibilidade de reter 32 mensagens em simultâneo. Quando o limite de mensagens é atingido, os apontadores *put* e *get* criados que permitem saber a posição atual do buffer para a escrita e leitura de mensagens, respetivamente, voltam a apontar para a posição inicial.

2.3. Diagrama de classes do Dançarino

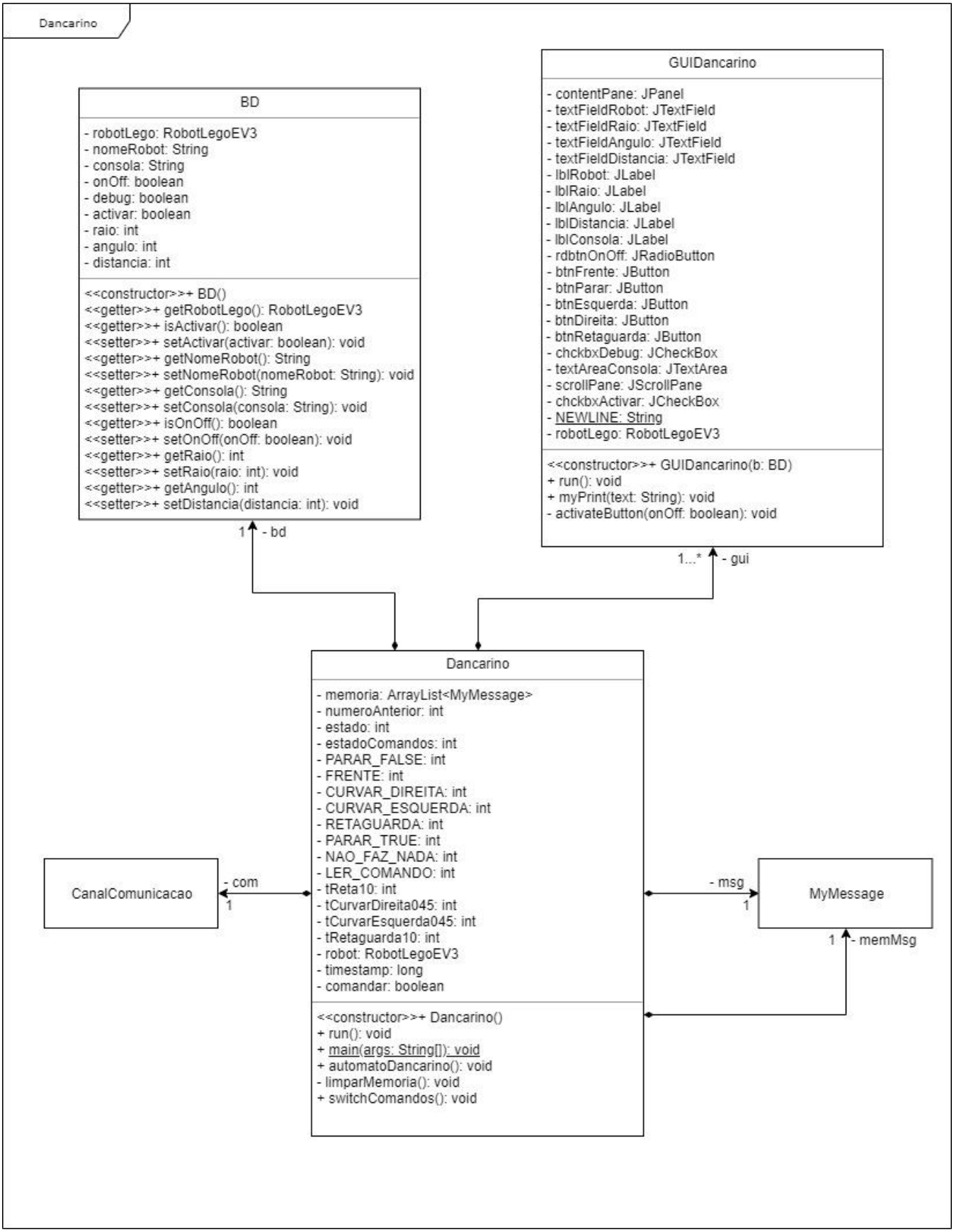

Figura 3 - Diagrama de classes (UML): Dancarino

2.4. Diagrama de classes do Coreógrafo

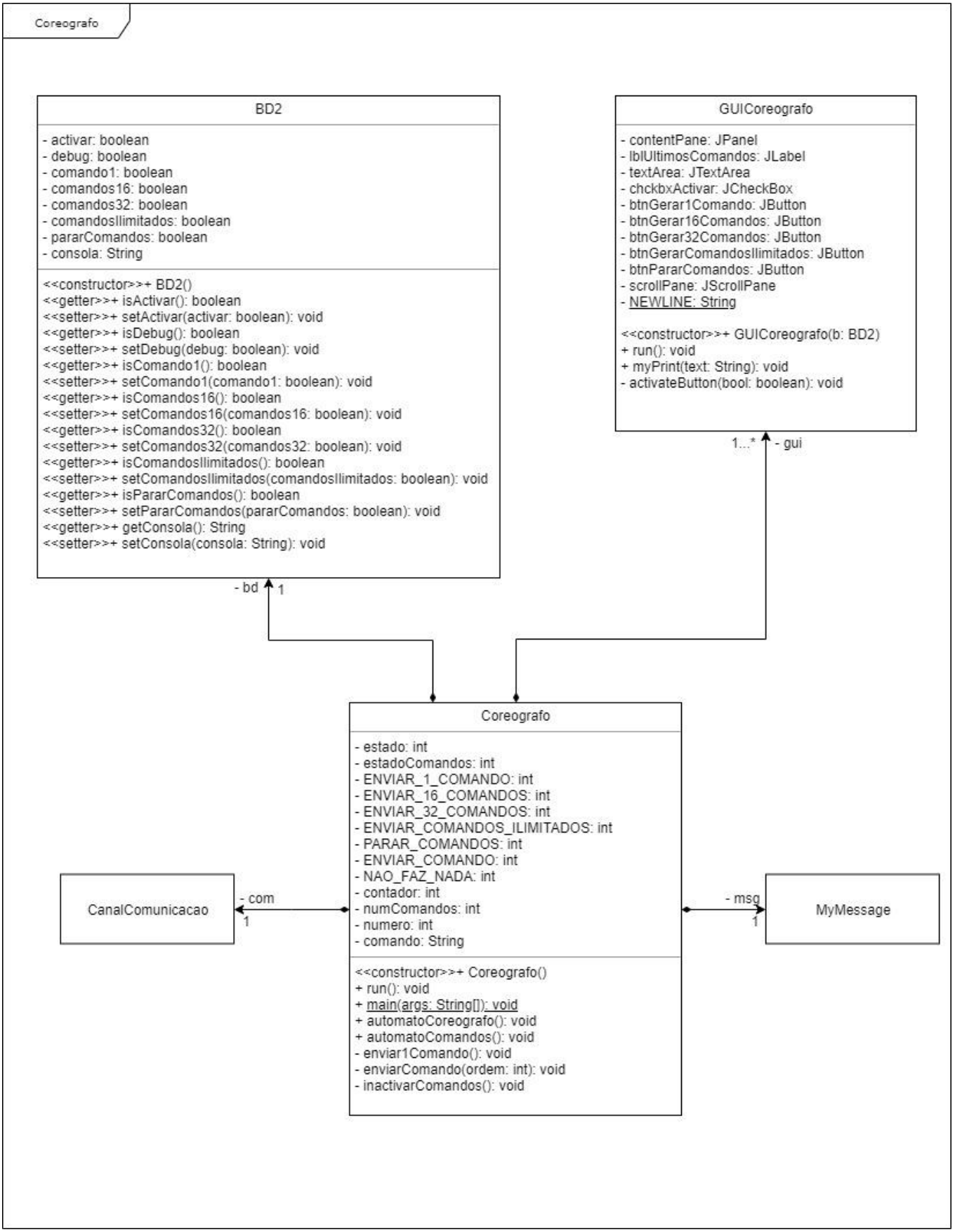

Figura 4 - Diagrama de classes (UML): Coreógrafo.

2.5. Diagrama de atividades do Dançarino

O dançarino é um processo JAVA que lê as mensagens enviadas pelo coreógrafo, guarda-as numa memória local e faz com que o robot execute os comandos das mensagens guardadas pela ordem enviada.
Para a criação do processo dançarino, criaram-se dois autómatos: o autómato "automatoDancarino" (Figura 5) e o autómato "SwitchComandos" (Figura 6).

O autómato "automatoDancarino" é constituído por dois estados:

- NAO_FAZ_NADA – é um estado de espera, onde o comportamento se encontra inativo. O autómato irá permanecer neste estado até a variável "activar" da classe "BD" se encontre a "true". Caso isso aconteça, o autómato transita para o estado LER_COMANDO;

- LER_COMANDO – neste estado é onde o comportamento DANÇARINO irá ler as mensagens que se encontrarem no *buffer* e, caso o número da mensagem seja maior do que o número da mensagem anteriormente lida, vai guardar a nova mensagem na memória local, caso contrário, não guarda, para evitar a repetição de mensagens já lidas. Caso a variável "activar" da classe "BD" fique a "false", o autómato retorna para o estado NAO_FAZ_NADA.

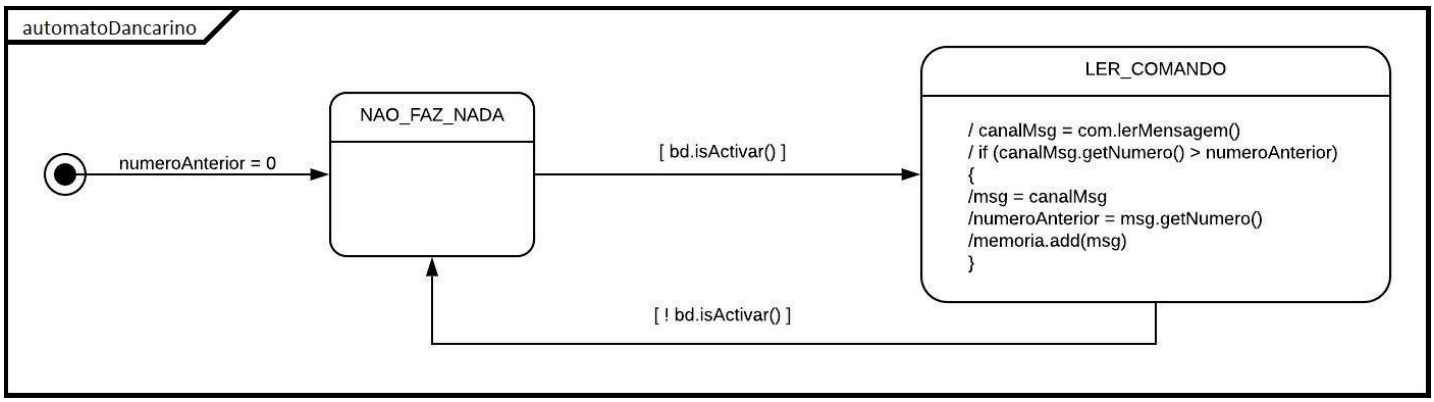

Figura 5 - Autómato "automatoDancarino"

O segundo autómato – "switchComandos" –, é onde irá ter os comandos que controlam os movimentos do robot de acordo com o valor da ordem da mensagem guardada. Os seus estados são:

- NAO_FAZ_NADA – espera que a memoria local não esteja vazia, ou seja, que já tenha sido guardada uma mensagem. Quando existe uma mensagem na memória, o autómato transita para o estado correspondente à ordem da mensagem recebida, simultaneamente colocando o booleano "comandar"[4] a "true".
  Esta transição dá-se através dos valores numéricos dos estados. Todos os estados são *private final int,* e os seus valores, exceto o valor do estado NAO_FAZ_NADA, seguem a correspondência numérica dos valores da variável ordem (Tabela 1) aos comandos do robot, ou seja, a transição de estados neste autómato do estado NÃO_FAZ_NADA para os restantes dá-se por
  "estado = memoria[0].getOrdem()";

- PARAR_FALSE e PARAR_TRUE – tanto o PARAR_FALSE como o PARAR_TRUE possuem a mesma implementação: comanda o robot, imprime na consola da classe "GUIDancarino", remove a mensagem lida da memória local, coloca o booleano "comandar" a "false" e transita novamente para o estado NAO_FAZ_NADA;

- FRENTE, RETAGUARDA, CURVAR_DIREITA e CURVAR_ESQUERDA – estes quatro estados também têm implementações similares. É nestes estados que o tempo de espera pelo término do movimento do robot é mais relevante, por isso, para além de comandar o robot, imprimir na consola, remover a mensagem lida da memória local e colocar o booleano "comandar" a "false", foi necessário um operador *if(),* que condiciona a transição para o estado NAO_FAZ_NADA: enquanto não tiver passado o dado tempo que um dado movimento precisa para acabar, o autómato permanece no estado atual.
  Para que as ações implementadas (ex.: comandar o robot) não se repitam enquanto o autómato não transita de estado, estas estão condicionadas por um outro *if(),* que diz que se o booleano "comandar" estiver a "true", executa as ações. Uma vez que o booleano é passado a "false" dentro deste *if(),* num próximo acesso ao autómato, este não irá volta a "entrar" no operador *if()*.

[4] Booleano "comandar" – impede o bloqueio do autómato num dado estado. É colocado a "true" no estado NAO_FAZ_NADA e colocado a "false" num dado estado, controlador de um movimento do robot, quando este acaba de esperar que o movimento do robot acabe.

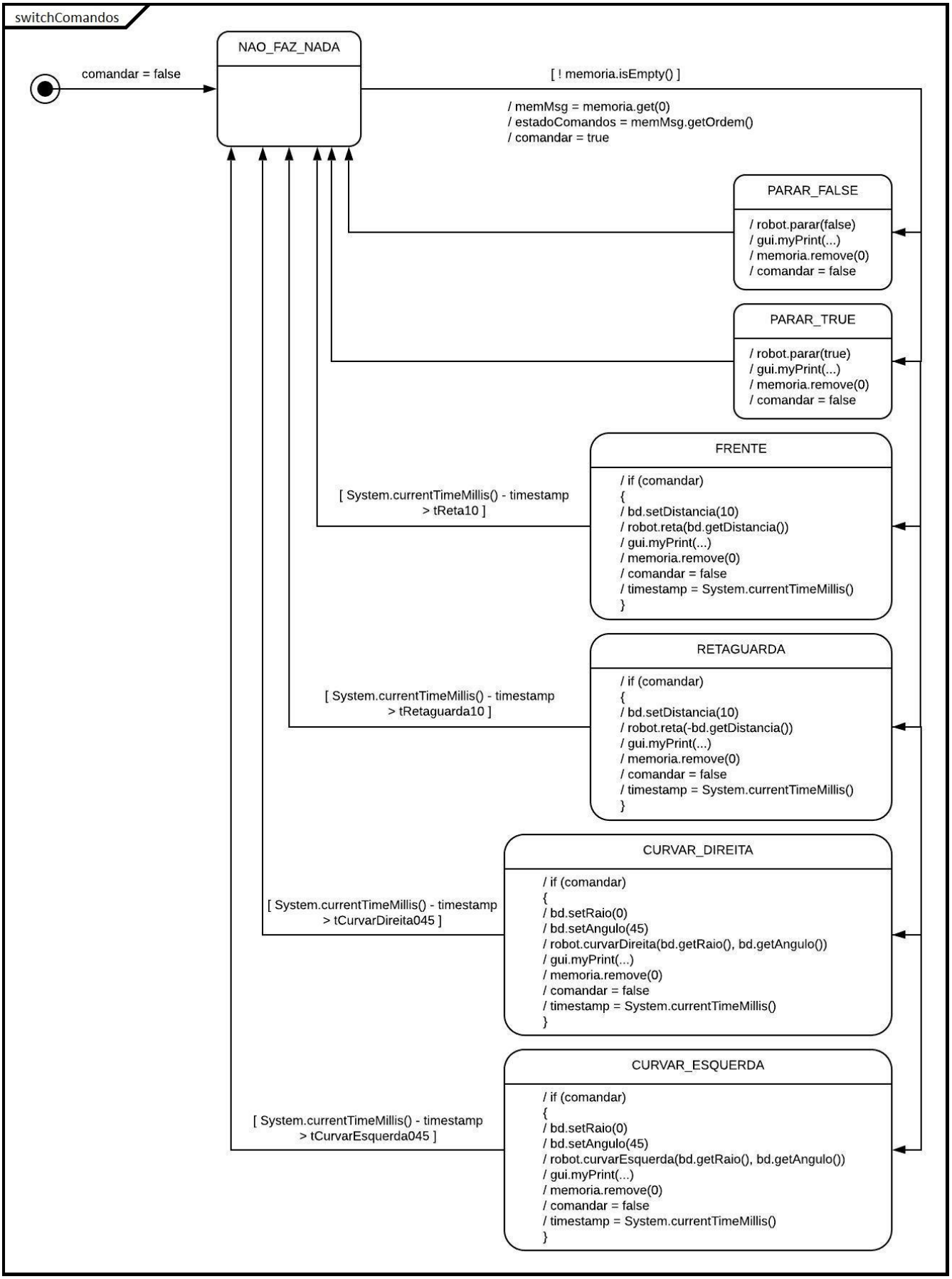

Figura 6 - Autómato "switchComandos"

2.6. Diagrama de atividades do Coreógrafo

O coreógrafo é um processo JAVA que envia mensagens de 500 em 500 milissegundos para o consumidor Dançarino. O número de mensagens a serem enviadas depende do comando selecionado – enviar 1 comando, enviar 16 comandos, enviar 32 comandos, enviar comandos ilimitados, para comandos –, estando este disponível de escolher na interface gráfica "GUICoreografo".

Para a criação do processo coreógrafo, criaram-se dois autómatos: o autómato "automatoCoreografo" (Figura 7) e o autómato "automatoComandos" (Figura 8).

O autómato "automatoCoreografo" é constituído por dois estados:

- NAO_FAZ_NADA – é um estado de espera, onde o comportamento se encontra inativo. Assim como o autómato "automatoDancarino" da classe "Dancarino", o autómato irá permanecer neste estado até a variável "activar", neste caso da classe "BD2", se encontre a "true". Caso isso aconteça, o autómato transita para o estado ENVIAR_COMANDO;

- ENVIAR_COMANDO – é neste estado que o comportamento COREOGRAFO estará ativo. Este estado garante que, quando a variável "activar" da classe "BD2" ficar a "false", o segundo autómato "automatoComandos" irá parar de enviar comandos, retornado este autómato para o seu estado NAO_FAZ_NADA correspondente, inativando os comandos[5] e enviando uma última mensagem "parar(false)"[6]. É também nesta condição que o autómato "automatoCoreografo" retorna para o estado NAO_FAZ_NADA.

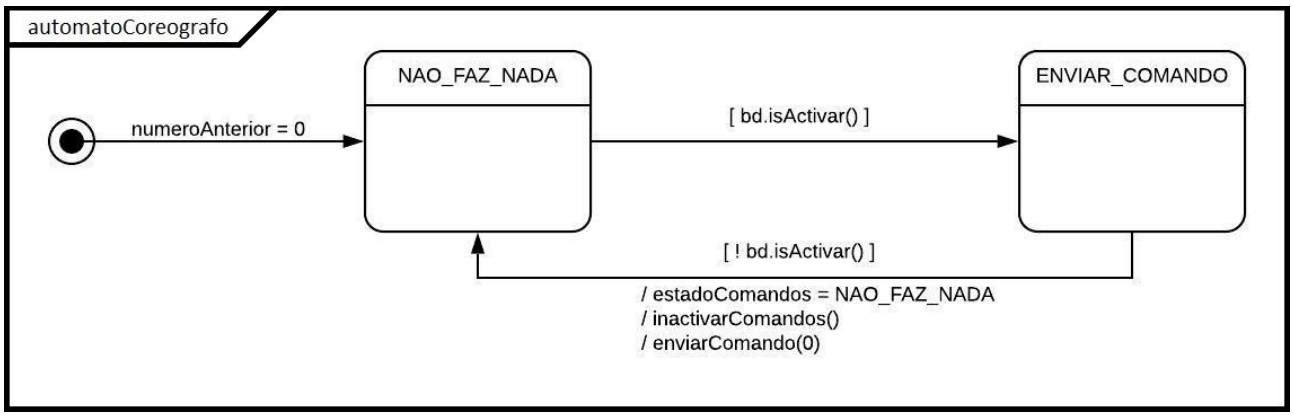

Figura 7 - Autómato "automatoCoreografo"

[5] InactivarComandos() – é uma função que põe todas as variáveis da classe "BD2", relativas aos comandos disponíveis na interface gráfica (ex.: boolean comando1, boolean comandosIlimitados, etc.), a "false", parando o envio de comandos.

[6] mensagem "parar(false)" – no autómato "automatoCoreografo", esta mensagem é enviada no fim de um dado comando, com a função "enviarComando(int ordem)", com o propósito de evitar que o robot execute a última mensagem recebida infinitamente.
A função utilizada para enviar esta mensagem recebe um int com a ordem da mensagem pretendida, neste caso, um zero, que correspondente à mensagem "parar(fase)" (na Tabela 1).

O segundo autómato – "automatoComandos" –, é onde irá ter os comandos que controlam o número de mensagens a serem enviadas de 500 em 500 milissegundos. Os seus estados são:

- NAO_FAZ_NADA – é um estado de espera, que verifica cada uma das variáveis da classe "BD2" relativas aos comandos, usando, em operadores *if(),* os *getter*'s "bd.isComando1()", "bd.isComandos16()", "bd.isComandos32()", "bd.isComandosIlimitados()" e "bd.isParar Comandos()"[7]. Quando um destes retornar "true", o autómato transita para o estado correspondente.
  Juntamente a esta transição, a variável "numComandos" é igualada a '1' para efeitos de debug na consola da interface gráfica "GUICoreografo", auxiliando na contagem das mensagens enviadas em cada comando;

- ENVIAR_1_COMANDO – neste estado é enviada uma única mensagem através da função "enviar1Comando()". Esta função escrever no buffer uma mensagem com uma ordem aleatória e imprime na consola a mensagem enviada. Realiza também um operador "while()" para esperar 500 milissegundos até poder enviar uma outra mensagem.
  Ainda neste estado, a variável "comando1" da classe "BD2" é colocada a "false" e é enviada mais uma mensagem de "parar(false)" com a função "enviarComando(int ordem)". De seguida, o autómato retorna para o estado NAO_FAZ_NADA;

- ENVIAR_16_COMANDOS – neste estado são enviadas 16 mensagens, cada uma com a função "enviar1Comando()". Este estado só irá transitar para o estado NAO_FAZ_NADA, colocar a variável "comando16" e enviar a mensagem final "parar(false)" quando a variável "contador" for igual a 16. Esta variável é inicializada a zero e vai sendo incrementada no operador *if()* situado neste estado. Quando esta condição for verdadeira, o contador é reiniciado a zero novamente.
  Este estado possuí também um operador *if()* para verificar a condição do comando "parar comandos" seja acionado. Caso isso aconteça, o autómato transita para o estado "PARAR_COMANDOS";

- ENVIAR_32_COMANDOS – este estado é igual ao estado ENVIAR_16_COMANDOS, com a exceção de a variável "contador" ser incrementada até esta ser igual a 32 e a variável "comandos32" ser atualizada a "false" ao invés da variável "comandos16";

- ENVIAR_COMANDOS_ILIMITADOS – neste estado, são enviadas mensagens ilimitadas com a função "enviar1Comando()". O autómato só transita deste estado quando o comando "parar comandos" for premido ou quando a variável "activar" ficar a "false";

[7] Variáveis nos *getter*'s dos comandos – estas variáveis são colocadas a "true" quando o botão respetivo ao comando é premido na interface gráfica "GUICoreografo".

- PARAR_COMANDOS – neste estado as variáveis da classe “BD2”, relativas aos comandos, são colocadas a “false” com a função “inactivarComandos()”, é enviada a mensagem “parar(false)” e o autómato retorna ao estado NAO_FAZ_NADA, parando assim os comandos ativos.

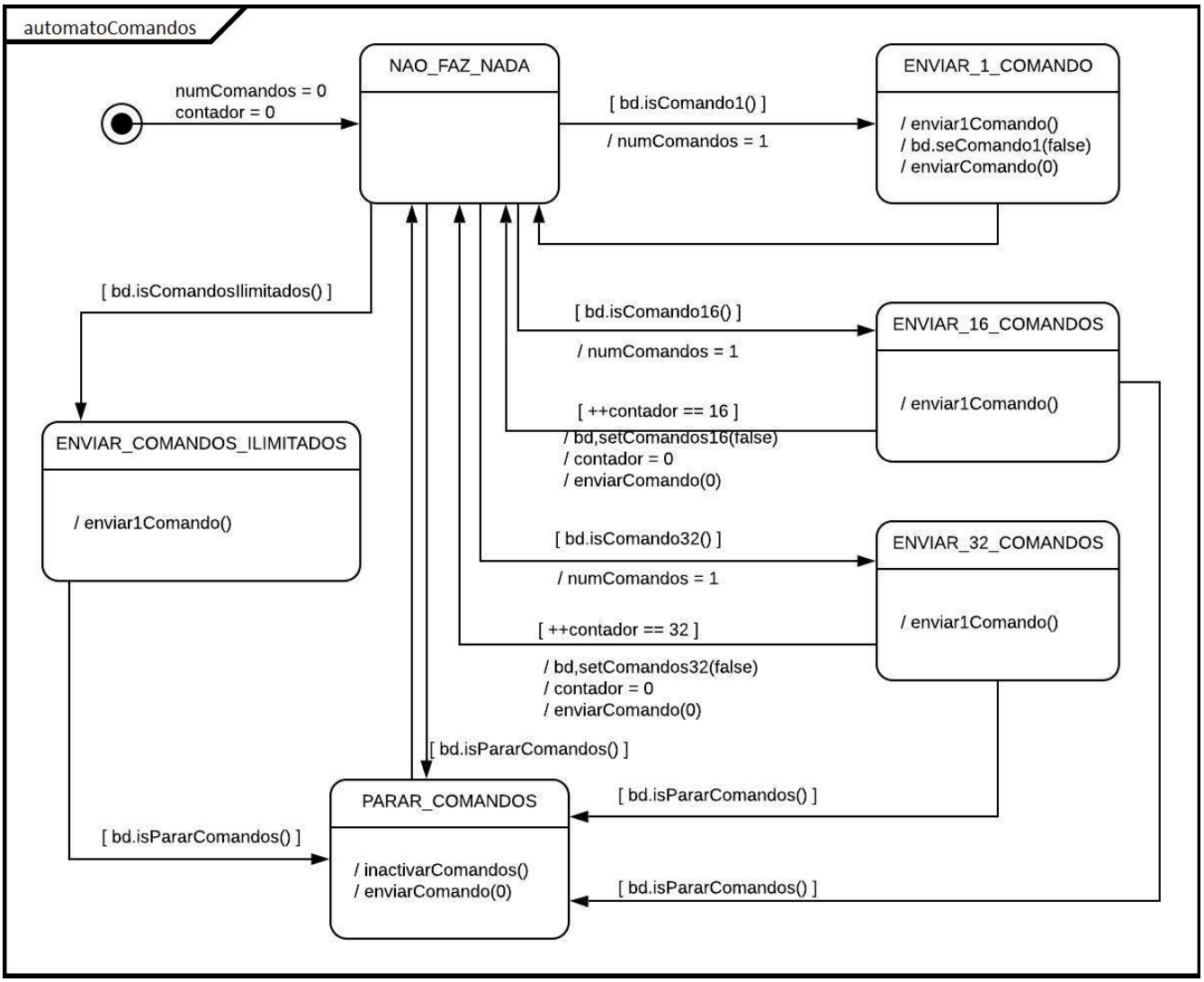

Figura 8 - Autómato "automatoComandos"

2.7. Sincronização no acesso ao Canal de Comunicação

A sincronização no acesso ao Canal de Comunicação é feita através de *FileLock*'s, que torna o acesso ao Buffer exclusivo, isto é, enquanto o processo do Dançarino acede ao canal o processo do Coreógrafo não pode aceder, e vice-versa.

O funcionamento do *FileLock* é o seguinte:

- O processo que irá aceder ao canal utiliza o método *lock()*, bloqueando assim, o acesso ao canal e executa as suas operações nesse canal; quando deixa de ser necessário operar no canal, utiliza o método *release()*, para permitir que outros processos possam aceder ao canal novamente;
- O coreógrafo faz *lock()* ao canal para escrever a mensagem e enquanto estiver a executar a operação o dançarino não poderá aceder ao canal. Da mesma maneira, enquanto o dançarino estiver a ler o canal, o coreógrafo, também não poderá escrever novas mensagens.

Relativamente à integridade dos dados, o canal mantém um registo da última mensagem lida, guardando o número dessa mensagem, e só avança para a posição seguinte no buffer se o número da mensagem que está a ler no momento for maior que o número da mensagem anterior, garantindo assim a leitura da nova informação, quando um processo escrever uma mensagem no buffer.

No entanto, o mesmo não acontece durante a escrita no buffer. Um processo pode escrever mensagens suficientes para preencher o buffer e voltar a escrever uma mensagem na mesma posição onde previamente havia escrito outra, sem garantias de que outro processo que utilize o canal de comunicação para consultar as mensagens escritas tenha lido a mensagem antiga, havendo assim uma possível perda de informação.

2.8. Resultados práticos da interação entre processos e o robot

Como resultados práticos da interação entre processos e o robot, observou-se que:

- Para o Dançarino realizar cada comando enviado, foi necessária uma medição para cada ação (FRENTE, CURVAR_DIREITA, CURVAR_ESQUERDA e RETAGUARDA). Para isso, cronometrou-se cada movimento e inseriram-se os respetivos tempos em cada ação. Como inicialmente o robot realizava nas ações CURVAR_DIREITA e CURVAR_ESQUERDA um ângulo de 90 graus quando inserido um ângulo de 45 graus, teve-se de reduzir o tempo destas ações para metade. Mesmo depois desta redução, o ângulo realizado pelo robot reduziu, mas ainda não era o ângulo necessário, sendo que se fez a validação com esse ângulo. Depois da validação, reduzimos o tempo para mais de metade e já se obteve o ângulo necessário;

- Para a observação das mensagens de forma mais clara nas interfaces gráficas "GUI_DANCARINO" e "GUI_COREOGRAFO", teve-se de alargar os elementos *JTextArea* e fazer: "textAreaConsola.setCaretPosition(textAreaConsola .getDocument().getLength()", para se conseguir visualizar os 10 últimos comandos recebidos;

- Os JAR's executáveis gerados para cada processo estavam funcionais, tendo um funcionamento idêntico ao do lançamento dos processos pelo programa eclipse, tendo assim um código portável para qualquer máquina que tenha o java instalado, sem ser necessário ter o programa eclipse.

3. Conclusões

Os objetivos do trabalho foram atingidos e as conclusões que foram tiradas foram as seguintes:

- A comunicação entre processos com memória partilhada serviu para ganhar sensibilidade relativamente aos acessos de processos a recursos de acesso exclusivo – no âmbito do trabalho, foi preciso controlar o acesso ao canal de comunicação, de modo a que o Dançarino não acedesse ao canal para ler a mensagem no buffer enquanto o Coreógrafo estivesse a utilizar o canal para escrever uma mensagem, mantendo assim a integridade dos dados;

- O Modelo Produtor-Consumidor explicitou o problema de sincronização multiprocesso, cuja solução implementada foi controlar o conteúdo do buffer – em vez do Produtor (Coreógrafo) apenas enviar as ordens com os comandos a serem executados, também enviava a cardinalidade (número) da mensagem. Assim o consumidor (Dançarino) consegue controlar se há informação nova para ser registada;

- Os diagramas de atividades permitiram ter uma visão de mais alto nível do problema e da solução implementada mostrando o fluxo de execução dos programas;

- Os diagramas de classes serviram para ter uma melhor estruturação e visualização do código desenvolvido;

- O desenvolvimento de interfaces gráficas permitiu aumentar a interação com o código desenvolvido e tornar o software mais comerciável;

- O uso do robot possibilitou o desenvolvimento de métodos não-bloqueantes – quando o Dançarino enviava comandos para o robot, em vez de ficar bloqueado à espera de resposta do robot continuava a executar o resto das suas funções.

Através deste trabalho foi possível mudar a forma de pensar de uma programação sequencial para uma programação concorrente, onde não sabemos qual processo vai executar a seguir, e como lidar com isso.

4. Bibliografia

— Jorge Pais, Folhas de Fundamentos de Sistemas Operativos versão 1, 2019-2020

— Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014), *Operating Systems: Three Easy Pieces [Chapter: Condition Variables]* (PDF), Arpaci-Dusseau Books

5. Anexos

5.1. Anexo A – Classe "BD"

```
2  public class BD {
3      private RobotLegoEV3 robotLego;
4  //  private MyRobotLego robotLego;
5      private String nomeRobot, consola;
6      private boolean onOff, debug;
7      private int raio, angulo, distancia;
8      private boolean activar;
9
10     public BD() {
11         robotLego = new RobotLegoEV3();
12 //      robotLego = new MyRobotLego();
13         nomeRobot = new String("EV7"); // Link
14         onOff = false;
15         debug = true;
16         raio = 10;
17         angulo = 90;
18         distancia = 20;
19         consola = new String("Fazendo Debug ... \n");
20
21         activar = false;
22     }
23
24 //  public MyRobotLego getRobotLego() {
25 //      return robotLego;
26 //  }
27     public RobotLegoEV3 getRobotLego() {
28         return robotLego;
29     }
30
31     public boolean isActivar() {
32         return activar;
33     }
34
35     public void setActivar(boolean activar) {
36         this.activar = activar;
37     }
38
39     public String getNomeRobot() {
40         return nomeRobot;
41     }
42     public void setNomeRobot(String nomeRobot) {
43         this.nomeRobot = nomeRobot;
44     }
45     public String getConsola() {
46         return consola;
47     }
48     public void setConsola(String consola) {
49         this.consola = consola;
50     }
51     public boolean isonOff() {
52         return onOff;
53     }
54     public void setOnOff(boolean onOff) {
55         this.onOff = onOff;
56     }
57     public boolean isDebug() {
58         return debug;
59     }
60     public void setDebug(boolean debug) {
61         this.debug = debug;
62     }
```

```
    public int getRaio() {
        return raio;
    }
    public void setRaio(int raio) {
        this.raio = raio;
    }
    public int getAngulo() {
        return angulo;
    }
    public void setAngulo(int angulo) {
        this.angulo = angulo;
    }
    public int getDistancia() {
        return distancia;
    }
    public void setDistancia(int distancia) {
        this.distancia = distancia;
    }
}
```

5.2. Anexo B – Classe ”GUIDancarino”

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.border.EmptyBorder;
4  import javax.swing.JLabel;
5  import javax.swing.JTextField;
6  import javax.swing.JRadioButton;
7  import javax.swing.JScrollPane;
8  import javax.swing.JButton;
9  import javax.swing.JCheckBox;
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;
12 import javax.swing.JTextArea;
13 import java.awt.Color;
14
15 public class GUIDancarino extends JFrame {
16
17     private JPanel contentPane;
18     private JTextField textFieldRobot, textFieldRaio, textFieldAngulo, textFieldDistancia;
19     private JLabel lblRobot, lblRaio, lblAngulo, lblDistancia, lblConsola;
20     private JRadioButton rdbtnOnOff;
21     private JButton btnFrente, btnParar, btnEsquerda, btnDireita, btnRetaguarda;
22     private JCheckBox chckbxDebug;
23     private JTextArea textAreaConsola;
24     private JScrollPane scrollPane;
25     private JCheckBox chckbxActivar;
26
27     private final static String newline = "\n";
28
29     private BD bd;
30 //  private MyRobotLego rb;
31     private RobotLegoEV3 rb;
32
33     public void run() {
34
35     }
36
37     /**
38      * Launch the application.
39      */
40 /*  public static void main(String[] args) {
41         GUIDancarino frame = new GUIDancarino();
42         frame.run();
43     }
44 */
45     /**
46      * Create the frame.
47      */
48     public GUIDancarino(BD b) {
49         bd = b;
50         rb = bd.getRobotLego();
51
52         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
53         setBounds(100, 100, 450, 487);
54         contentPane = new JPanel();
55         contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
56         setContentPane(contentPane);
57         contentPane.setLayout(null);
58
59         lblRobot = new JLabel("Robot");
60         lblRobot.setBounds(77, 11, 39, 14);
61         contentPane.add(lblRobot);
62
```

```
63          textFieldRobot = new JTextField();
64          textFieldRobot.addActionListener(new ActionListener() {
65              public void actionPerformed(ActionEvent arg0) {
66
67                  bd.setNomeRobot(textFieldRobot.getText());
68                  myPrint("Nome robot: " + bd.getNomeRobot());
69              }
70          });
71          textFieldRobot.setText(bd.getNomeRobot());
72          textFieldRobot.setBounds(126, 8, 186, 20);
73          contentPane.add(textFieldRobot);
74          textFieldRobot.setColumns(10);
75
76          rdbtnOnOff = new JRadioButton("On/Off");
77          rdbtnOnOff.addActionListener(new ActionListener() {
78              public void actionPerformed(ActionEvent e) {
79
80                  if (rdbtnOnOff.isSelected()) {
81
82                      if (rb.OpenEV3(bd.getNomeRobot())) {
83                          bd.setOnOff(rdbtnOnOff.isSelected());
84                      }
85                      else {
86                          rdbtnOnOff.setSelected(false);
87                      }
88                  }
89                  else {
90                      bd.setOnOff(rdbtnOnOff.isSelected());
91                      rb.CloseEV3();
92                  }
93                  activateButton(bd.isonOff());
94                  myPrint("OnOff: " + bd.isonOff());
95              }
96          });
97          rdbtnOnOff.setSelected(bd.isonOff());
98          rdbtnOnOff.setBounds(333, 7, 70, 23);
99          contentPane.add(rdbtnOnOff);
100
101         lblRaio = new JLabel("Raio");
102         lblRaio.setBounds(10, 39, 46, 14);
103         contentPane.add(lblRaio);
104
105         textFieldRaio = new JTextField();
106         textFieldRaio.addActionListener(new ActionListener() {
107             public void actionPerformed(ActionEvent e) {
108
109                 try {
110                     bd.setRaio(Integer.parseInt(textFieldRaio.getText()));
111                 } catch (NumberFormatException nfe) {
112                     textFieldRaio.setText("" + bd.getRaio());
113                     myPrint("Raio : Exception Error!");
114                 }
115                 myPrint("Raio: " + bd.getRaio());
116             }
117         });
118         textFieldRaio.setText("" + bd.getRaio());
119         textFieldRaio.setBounds(42, 36, 46, 20);
120         contentPane.add(textFieldRaio);
121         textFieldRaio.setColumns(10);
122
123         lblAngulo = new JLabel("Angulo");
124         lblAngulo.setBounds(136, 39, 39, 14);
125         contentPane.add(lblAngulo);
126
```

```
127         textFieldAngulo = new JTextField();
128         textFieldAngulo.addActionListener(new ActionListener() {
129             public void actionPerformed(ActionEvent e) {
130 
131                 try {
132                     bd.setAngulo(Integer.parseInt(textFieldAngulo.getText()));
133                 } catch (NumberFormatException nfe) {
134                     textFieldAngulo.setText("" + bd.getAngulo());
135                     myPrint("Angulo : Exception Error!");
136                 }
137                 myPrint("Angulo: " + bd.getAngulo());
138             }
139         });
140         textFieldAngulo.setText("" + bd.getAngulo());
141         textFieldAngulo.setBounds(185, 39, 46, 20);
142         contentPane.add(textFieldAngulo);
143         textFieldAngulo.setColumns(10);
144 
145         lblDistancia = new JLabel("Distancia");
146         lblDistancia.setBounds(301, 39, 46, 14);
147         contentPane.add(lblDistancia);
148 
149         textFieldDistancia = new JTextField();
150         textFieldDistancia.addActionListener(new ActionListener() {
151             public void actionPerformed(ActionEvent e) {
152 
153                 try {
154                     bd.setDistancia(Integer.parseInt(textFieldDistancia.getText()));
155                 } catch (NumberFormatException nfe) {
156                     textFieldDistancia.setText("" + bd.getDistancia());
157                     myPrint("Distancia : Exception Error!");
158                 }
159                 myPrint("Distancia: " + bd.getDistancia());
160             }
161         });
162         textFieldDistancia.setText("" + bd.getDistancia());
163         textFieldDistancia.setBounds(357, 37, 46, 20);
164         contentPane.add(textFieldDistancia);
165         textFieldDistancia.setColumns(10);
166 
167         btnFrente = new JButton("Frente");
168         btnFrente.addActionListener(new ActionListener() {
169             public void actionPerformed(ActionEvent e) {
170 
171                 rb.Reta(bd.getDistancia());
172                 rb.Parar(false);
173                 myPrint("Reta: " + bd.getDistancia());
174 
175             }
176         });
177         btnFrente.setBackground(Color.GREEN);
178         btnFrente.setBounds(172, 70, 101, 44);
179         contentPane.add(btnFrente);
180 
181         btnParar = new JButton("Parar");
182         btnParar.addActionListener(new ActionListener() {
183             public void actionPerformed(ActionEvent e) {
184 
185                 rb.Parar(true);
186                 myPrint("Parar");
187             }
188         });
189         btnParar.setBackground(Color.RED);
190         btnParar.setBounds(172, 125, 101, 44);
191         contentPane.add(btnParar);
192 
```

```
193         btnDireita = new JButton("Direita");
194         btnDireita.addActionListener(new ActionListener() {
195             public void actionPerformed(ActionEvent e) {
196 
197                 rb.CurvarDireita(bd.getRaio(), bd.getAngulo());
198                 rb.Parar(false);
199                 myPrint("Direita: raio " + bd.getRaio() + " angulo " + bd.getAngulo());
200             }
201         });
202         btnDireita.setBackground(Color.MAGENTA);
203         btnDireita.setBounds(283, 125, 101, 44);
204         contentPane.add(btnDireita);
205 
206         btnEsquerda = new JButton("Esquerda");
207         btnEsquerda.addActionListener(new ActionListener() {
208             public void actionPerformed(ActionEvent e) {
209 
210                 rb.CurvarEsquerda(bd.getRaio(), bd.getAngulo());
211                 rb.Parar(false);
212                 myPrint("Esquerda: raio " + bd.getRaio() + " angulo " + bd.getAngulo());
213             }
214         });
215         btnEsquerda.setBackground(Color.ORANGE);
216         btnEsquerda.setBounds(60, 125, 101, 44);
217         contentPane.add(btnEsquerda);
218 
219         btnRetaguarda = new JButton("Retaguarda");
220         btnRetaguarda.addActionListener(new ActionListener() {
221             public void actionPerformed(ActionEvent e) {
222 
223                 rb.Reta(-bd.getDistancia());
224                 rb.Parar(false);
225                 myPrint("Reta: " + -bd.getDistancia());
226             }
227         });
228         btnRetaguarda.setBackground(Color.BLUE);
229         btnRetaguarda.setBounds(172, 180, 101, 44);
230         contentPane.add(btnRetaguarda);
231 
232         chckbxDebug = new JCheckBox("Debug");
233         chckbxDebug.addActionListener(new ActionListener() {
234             public void actionPerformed(ActionEvent e) {
235 
236                 bd.setDebug(chckbxDebug.isSelected());
237                 myPrint("Debug: " + bd.isDebug());
238             }
239         });
240         chckbxDebug.setSelected(bd.isDebug());
241         chckbxDebug.setBounds(10, 212, 61, 23);
242         contentPane.add(chckbxDebug);
243 
244         lblConsola = new JLabel("Consola");
245         lblConsola.setBounds(10, 242, 46, 14);
246         contentPane.add(lblConsola);
247 
248         scrollPane = new JScrollPane();
249         scrollPane.setBounds(10, 257, 414, 180);
250         contentPane.add(scrollPane);
251 
252         textAreaConsola = new JTextArea();
253         textAreaConsola.setText(bd.getConsola());
254         scrollPane.setViewportView(textAreaConsola);
255         textAreaConsola.setLineWrap(true);
256         textAreaConsola.setEditable(false);
257 
```

```
258         chckbxActivar = new JCheckBox("Activar");
259         chckbxActivar.addActionListener(new ActionListener() {
260             public void actionPerformed(ActionEvent e) {
261 
262                 if(!bd.isonOff())
263                     chckbxActivar.setSelected(false);
264                 bd.setActivar(chckbxActivar.isSelected());
265                 myPrint("Dançarino Activo - activar: " + bd.isActivar());
266                 activateButton(!bd.isActivar());
267             }
268         });
269         chckbxActivar.setBounds(354, 212, 70, 23);
270         contentPane.add(chckbxActivar);
271 
272         JButton btnClear = new JButton("Clear");
273         btnClear.addActionListener(new ActionListener() {
274             public void actionPerformed(ActionEvent arg0) {
275 
276                 bd.setConsola(" ");
277                 textAreaConsola.setText(bd.getConsola());
278             }
279         });
280         btnClear.setBounds(77, 212, 70, 23);
281         contentPane.add(btnClear);
282 
283         activateButton(false);
284 
285         setVisible(true);
286     }
287 
288     public void myPrint(String text) {
289         if (bd.isDebug()) {
290             bd.setConsola(text);
291             textAreaConsola.append(bd.getConsola() + newline);
292             textAreaConsola.setCaretPosition(textAreaConsola.getDocument().getLength());
293         }
294     }
295 
296     private void activateButton(boolean onOff) {
297         btnFrente.setEnabled(onOff);
298         btnEsquerda.setEnabled(onOff);
299         btnDireita.setEnabled(onOff);
300         btnRetaguarda.setEnabled(onOff);
301         btnParar.setEnabled(onOff);
302     }
303 }
304 
```

5.3. Anexo C – Classe "Dancarino"

```
1  import java.util.ArrayList;
2
3  public class Dancarino {
4
5      private ArrayList<MyMessage> memoria;
6      private int numeroAnterior;
7
8      private int estado, estadoComandos;
9      private final int PARAR_FALSE = 0;
10     private final int FRENTE = 1;
11     private final int CURVAR_DIREITA = 2;
12     private final int CURVAR_ESQUERDA = 3;
13     private final int RETAGUARDA = 4;
14     private final int PARAR_TRUE = 5;
15     private final int NAO_FAZ_NADA = 6;
16     private final int LER_COMANDO = 7;
17
18     private final int tReta10 = 720;
19     private final int tCurvarDireita045 = 180;
20     private final int tCurvarEsquerda045 = 180;
21     private final int tRetaguarda10 = 720;
22
23 //  private MyRobotLego robot;
24     private RobotLegoEV3 robot;
25     private BD bd;
26     private CanalComunicacao com;
27     private GUIDancarino gui;
28     private MyMessage msg;
29     private MyMessage memMsg;
30     private long timestamp;
31     private boolean comandar;
32
33     public Dancarino() {
34         estado = NAO_FAZ_NADA;
35         estadoComandos = NAO_FAZ_NADA;
36
37         memoria = new ArrayList<MyMessage>();
38         limparMemoria();
39         numeroAnterior = 0;
40         msg = new MyMessage(0, 0);
41         memMsg = new MyMessage(0, 0);
42
43         com = new CanalComunicacao();
44         bd = new BD();
45         gui = new GUIDancarino(bd);
46         robot = bd.getRobotLego();
47
48         comandar = false;
49     }
50
51     public void run() {
52         automatoDancarino();
53         switchComandos();
54     }
55
56     public static void main(String[] args) {
57         Dancarino dancer = new Dancarino();
58         while (true) {
59             dancer.run();
60         }
61     }
62
```

```
63      public void automatoDancarino() {
64          switch (estado) {
65          case NAO_FAZ_NADA:
66              if (bd.isActivar())
67                  estado = LER_COMANDO;
68              break;
69
70          case LER_COMANDO:
71              MyMessage canalMsg = com.lerMensagem();
72              if (canalMsg.getNumero() > numeroAnterior) {
73                  msg = canalMsg;
74                  numeroAnterior = msg.getNumero();
75                  memoria.add(msg);
76              }
77              if (!bd.isActivar())
78                  estado = NAO_FAZ_NADA;
79              break;
80
81          default:
82              System.out.println("Erro no Dançarino: Automato Dançarino - Estado: " + estado);
83              estado = NAO_FAZ_NADA;
84              break;
85          }
86      }
87
88      private void limparMemoria() {
89          int size = memoria.size();
90          for (int i = 0; i < size; i++)
91              memoria.remove(0);
92      }
93
94      public void switchComandos() {
95          switch (estadoComandos) {
96          case NAO_FAZ_NADA:
97              if (!memoria.isEmpty()) {
98                  memMsg = memoria.get(0);
99                  estadoComandos = memMsg.getOrdem();
100                 comandar = true;
101             }
102             break;
103
104         case PARAR_FALSE:
105             robot.Parar(false);
106
107             gui.myPrint("[" + memMsg.getNumero() + ", " + memMsg.getOrdem() + "] -> Parar: false");
108
109             memoria.remove(0);
110             comandar = false;
111
112             estadoComandos = NAO_FAZ_NADA;
113             break;
114
115         case FRENTE:
116             if (comandar) {
117                 bd.setDistancia(10);
118                 robot.Reta(bd.getDistancia());
119
120                 gui.myPrint("[" + memMsg.getNumero() + ", " + memMsg.getOrdem() + "] -> Reta: " + bd.getDistancia());
121
122                 memoria.remove(0);
123                 comandar = false;
124
125                 timestamp = System.currentTimeMillis();
126             }
127             if (System.currentTimeMillis() - timestamp > tReta10)
128                 estadoComandos = NAO_FAZ_NADA;
129             break;
130
```

```
131         case CURVAR_DIREITA:
132             if (comandar) {
133 
134                 bd.setRaio(0);
135                 bd.setAngulo(45);
136                 robot.CurvarDireita(bd.getRaio(), bd.getAngulo());
137 
138                 gui.myPrint("[" + memMsg.getNumero() + ", " + memMsg.getOrdem() + "] -> Direita: raio " + bd.getRaio()
139                         + " angulo " + bd.getAngulo());
140 
141                 memoria.remove(0);
142                 comandar = false;
143 
144                 timestamp = System.currentTimeMillis();
145             }
146             if (System.currentTimeMillis() - timestamp > tCurvarDireita045)
147                 estadoComandos = NAO_FAZ_NADA;
148 
149             break;
150 
151         case CURVAR_ESQUERDA:
152             if (comandar) {
153                 bd.setRaio(0);
154                 bd.setAngulo(45);
155                 robot.CurvarEsquerda(bd.getRaio(), bd.getAngulo());
156 
157                 gui.myPrint("[" + memMsg.getNumero() + ", " + memMsg.getOrdem() + "] -> Esquerda: raio " + bd.getRaio()
158                         + " angulo " + bd.getAngulo());
159 
160                 memoria.remove(0);
161                 comandar = false;
162 
163                 timestamp = System.currentTimeMillis();
164             }
165             if (System.currentTimeMillis() - timestamp > tCurvarEsquerda045)
166                 estadoComandos = NAO_FAZ_NADA;
167 
168             break;
170         case RETAGUARDA:
171             if (comandar) {
172                 bd.setDistancia(10);
173                 robot.Reta(-bd.getDistancia());
174 
175                 gui.myPrint("[" + memMsg.getNumero() + ", " + memMsg.getOrdem() + "]  -> Reta: " + -bd.getDistancia());
176 
177                 memoria.remove(0);
178                 comandar = false;
179 
180                 timestamp = System.currentTimeMillis();
181             }
182             if (System.currentTimeMillis() - timestamp > tRetaguarda10)
183                 estadoComandos = NAO_FAZ_NADA;
184 
185             break;
186 
187         case PARAR_TRUE:
188             robot.Parar(true);
189 
190             gui.myPrint("[" + memMsg.getNumero() + ", " + memMsg.getOrdem() + "] -> Parar: true");
191 
192             memoria.remove(0);
193             comandar = false;
194 
195             estadoComandos = NAO_FAZ_NADA;
196             break;
197 
198         default:
199             System.out.println("Erro no Dançarino: AutomatoComandos  - Estado: " + estadoComandos);
200             estado = NAO_FAZ_NADA;
201             break;
202         }
203     }
204 
205 }
```

5.4. Anexo D – Classe "BD2"

```
1
2  public class BD2 {
3      private boolean activar, debug, comando1, comandos16, comandos32,
4      comandosIlimitados, pararComandos;
5      private String consola;
6
7      public BD2() {
8          activar = false;
9          debug = false;
10         comando1 = false;
11         comandos16 = false;
12         comandos32 = false;
13         comandosIlimitados = false;
14         pararComandos = false;
15         consola = new String("");
16     }
17
18     public boolean isActivar() {
19         return activar;
20     }
21
22     public void setActivar(boolean activar) {
23         this.activar = activar;
24     }
25
26     public boolean isDebug() {
27         return debug;
28     }
29
30     public void setDebug(boolean debug) {
31         this.debug = debug;
32     }
33
34     public boolean isComando1() {
35         return comando1;
36     }
37
38     public void setComando1(boolean comando1) {
39         this.comando1 = comando1;
40     }
41
42     public boolean isComandos16() {
43         return comandos16;
44     }
45
46     public void setComandos16(boolean comandos16) {
47         this.comandos16 = comandos16;
48     }
49
50     public boolean isComandos32() {
51         return comandos32;
52     }
53
54     public void setComandos32(boolean comandos32) {
55         this.comandos32 = comandos32;
56     }
57
```

```
58      public boolean isComandosIlimitados() {
59          return comandosIlimitados;
60      }
61
62      public void setComandosIlimitados(boolean comandosIlimitados) {
63          this.comandosIlimitados = comandosIlimitados;
64      }
65
66      public boolean isPararComandos() {
67          return pararComandos;
68      }
69
70      public void setPararComandos(boolean pararComandos) {
71          this.pararComandos = pararComandos;
72      }
73
74      public String getConsola() {
75          return consola;
76      }
77
78      public void setConsola(String consola) {
79          this.consola = consola;
80      }
81  }
82
```

5.5. Anexo E – Classe "GUICoreografo"

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JScrollPane;
4  import javax.swing.border.EmptyBorder;
5  import javax.swing.JLabel;
6  import javax.swing.JTextArea;
7  import javax.swing.JCheckBox;
8  import javax.swing.JButton;
9  import java.awt.event.ActionListener;
10 import java.awt.event.ActionEvent;
11
12 public class GUICoreografo extends JFrame {
13
14     private JPanel contentPane;
15
16     private JLabel lblUltimosComandos;
17     private JTextArea textArea;
18     private JCheckBox chckbxActivar;
19     private JButton btnGerar1Comando, btnGerar16Comandos, btnGerar32Comandos,
20     btnGerarComandosIlimitados, btnPararComandos;
21     private JScrollPane scrollPane;
22
23     private final static String newline = "\n";
24
25     private BD2 bd;
26
27
28     /**
29      * Launch the application.
30      */
31 /*  public static void main(String[] args) {
32         GUICoreografo GUI = new GUICoreografo();
33         GUI.run();
34     }
35 */
36     public void run() {
37
38     }
39
40     /**
41      * Create the frame.
42      */
43     public GUICoreografo(BD2 b) {
44         bd = b;
45
46         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
47         setBounds(100, 100, 523, 300);
48         contentPane = new JPanel();
49         contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
50         setContentPane(contentPane);
51         contentPane.setLayout(null);
52
53         lblUltimosComandos = new JLabel("Últimos 10 Comandos");
54         lblUltimosComandos.setBounds(77, 11, 129, 14);
55         contentPane.add(lblUltimosComandos);
56
57         scrollPane = new JScrollPane();
58         scrollPane.setBounds(10, 36, 277, 214);
59         contentPane.add(scrollPane);
60
```

```
61          textArea = new JTextArea();
62          scrollPane.setViewportView(textArea);
63          textArea.setLineWrap(true);
64          textArea.setEditable(false);
65
66          chckbxActivar = new JCheckBox("Activar");
67          chckbxActivar.addActionListener(new ActionListener() {
68              public void actionPerformed(ActionEvent arg0) {
69
70                  bd.setActivar(chckbxActivar.isSelected());
71                  activateButton(chckbxActivar.isSelected());
72                  myPrint("Coreografo Activo - activar: " + bd.isActivar());
73              }
74          });
75          chckbxActivar.setBounds(368, 7, 101, 23);
76          contentPane.add(chckbxActivar);
77
78          btnGerar1Comando = new JButton("Gerar 1 comando");
79          btnGerar1Comando.addActionListener(new ActionListener() {
80              public void actionPerformed(ActionEvent e) {
81
82                  bd.setComando1(true);
83              }
84          });
85          btnGerar1Comando.setBounds(297, 48, 188, 29);
86          contentPane.add(btnGerar1Comando);
87
88          btnGerar16Comandos = new JButton("Gerar 16 comandos");
89          btnGerar16Comandos.addActionListener(new ActionListener() {
90              public void actionPerformed(ActionEvent e) {
91
92                  bd.setComandos16(true);
93              }
94          });
95          btnGerar16Comandos.setBounds(297, 88, 188, 29);
96          contentPane.add(btnGerar16Comandos);
97
98          btnGerar32Comandos = new JButton("Gerar 32 comandos");
99          btnGerar32Comandos.addActionListener(new ActionListener() {
100             public void actionPerformed(ActionEvent e) {
101
102                 bd.setComandos32(true);
103             }
104         });
105         btnGerar32Comandos.setBounds(297, 128, 188, 29);
106         contentPane.add(btnGerar32Comandos);
107
108         btnGerarComandosIlimitados = new JButton("Gerar comandos ilimitados");
109         btnGerarComandosIlimitados.addActionListener(new ActionListener() {
110             public void actionPerformed(ActionEvent e) {
111
112                 bd.setComandosIlimitados(true);
113             }
114         });
115         btnGerarComandosIlimitados.setBounds(297, 168, 188, 29);
116         contentPane.add(btnGerarComandosIlimitados);
117
```

```
118         btnPararComandos = new JButton("Parar comandos");
119         btnPararComandos.addActionListener(new ActionListener() {
120             public void actionPerformed(ActionEvent e) {
121 
122                 bd.setPararComandos(true);
123             }
124         });
125         btnPararComandos.setBounds(297, 209, 188, 30);
126         contentPane.add(btnPararComandos);
127 
128         JButton btnClear = new JButton("Clear");
129         btnClear.addActionListener(new ActionListener() {
130             public void actionPerformed(ActionEvent arg0) {
131 
132                 bd.setConsola(" ");
133                 textArea.setText(bd.getConsola());
134             }
135         });
136         btnClear.setBounds(216, 7, 71, 23);
137         contentPane.add(btnClear);
138 
139         activateButton(false);
140 
141         setVisible(true);
142     }
143 
144     public void myPrint(String text) {
145         bd.setConsola(text);
146         textArea.append(bd.getConsola() + newline);
147         textArea.setCaretPosition(textArea.getDocument().getLength());
148     }
149 
150     private void activateButton(boolean bool) {
151         btnGerar1Comando.setEnabled(bool);
152         btnGerar16Comandos.setEnabled(bool);
153         btnGerar32Comandos.setEnabled(bool);
154         btnGerarComandosIlimitados.setEnabled(bool);
155         btnPararComandos.setEnabled(bool);
156     }
157 }
158 
```

5.6. Anexo F – Classe "Coreografo"

```
1
2  public class Coreografo {
3
4      private int estado, estadoComandos;
5      private final int NAO_FAZ_NADA = 0;
6      private final int ENVIAR_1_COMANDO = 1;
7      private final int ENVIAR_16_COMANDOS = 2;
8      private final int ENVIAR_32_COMANDOS = 3;
9      private final int ENVIAR_COMANDOS_ILIMITADOS = 4;
10     private final int PARAR_COMANDOS = 5;
11     private final int ENVIAR_COMANDO = 6;
12
13     private int contador, numComandos, numero;
14
15     private CanalComunicacao com;
16     private BD2 bd;
17     private GUICoreografo gui;
18     private String comando;
19
20     public Coreografo() {
21         estado = NAO_FAZ_NADA;
22         estadoComandos = NAO_FAZ_NADA;
23         contador = 0;
24         numComandos = 0;
25         numero = 0;
26
27         com = new CanalComunicacao();
28         bd = new BD2();
29         gui = new GUICoreografo(bd);
30     }
31
32     public void run() {
33         automatoCoreografo();
34         automatoComandos();
35     }
36
37     public static void main(String[] args) {
38         Coreografo coreo = new Coreografo();
39        while (true) {
40            coreo.run();
41        }
42     }
43
44     public void automatoCoreografo() {
45         switch (estado) {
46         case NAO_FAZ_NADA:
47             if (bd.isActivar()) {
48                 estado = ENVIAR_COMANDO;
49             }
50             break;
51
52         case ENVIAR_COMANDO:
53             if (!bd.isActivar()) {
54                 estadoComandos = NAO_FAZ_NADA;
55                 inactivarComandos();
56                 estado = NAO_FAZ_NADA;
57             }
58             break;
59
```

```
60          default:
61              System.out.println("Erro no Coreografo: automatoCoreografo()");
62              estado = NAO_FAZ_NADA;
63              break;
64          }
65      }
66
67      public void automatoComandos() {
68          switch (estadoComandos) {
69          case NAO_FAZ_NADA:
70              if (bd.isComando1()) {
71                  numComandos = 1;
72                  estadoComandos = ENVIAR_1_COMANDO;
73              } else if (bd.isComandos16()) {
74                  numComandos = 1;
75                  estadoComandos = ENVIAR_16_COMANDOS;
76              } else if (bd.isComandos32()) {
77                  numComandos = 1;
78                  estadoComandos = ENVIAR_32_COMANDOS;
79              } else if (bd.isComandosIlimitados()) {
80                  numComandos = 1;
81                  estadoComandos = ENVIAR_COMANDOS_ILIMITADOS;
82              } else if (bd.isPararComandos()) {
83                  estadoComandos = PARAR_COMANDOS;
84              }
85              break;
86
87          case ENVIAR_1_COMANDO:
88              enviar1Comando();
89              bd.setComando1(false);
90              enviarComando(0);
91              estadoComandos = NAO_FAZ_NADA;
92              break;
93
94          case ENVIAR_16_COMANDOS:
95              enviar1Comando();
96              if (++contador == 16) {
97                  bd.setComandos16(false);
98                  contador = 0;
99                  enviarComando(0);
100                 estadoComandos = NAO_FAZ_NADA;
101             }
102             if (bd.isPararComandos())
103                 estadoComandos = PARAR_COMANDOS;
104             break;
105
106         case ENVIAR_32_COMANDOS:
107             enviar1Comando();
108             if (++contador == 32) {
109                 bd.setComandos32(false);
110                 contador = 0;
111                 enviarComando(0);
112                 estadoComandos = NAO_FAZ_NADA;
113             }
114             if (bd.isPararComandos())
115                 estadoComandos = PARAR_COMANDOS;
116             break;
117
```

```
118         case ENVIAR_COMANDOS_ILIMITADOS:
119             enviar1Comando();
120             if (bd.isPararComandos())
121                 estadoComandos = PARAR_COMANDOS;
122             break;
123 
124         case PARAR_COMANDOS:
125             inactivarComandos();
126             enviarComando(0);
127             estadoComandos = NAO_FAZ_NADA;
128             break;
129 
130         default:
131             System.out.println("Erro no Coreografo: automatoComandos");
132             estadoComandos = NAO_FAZ_NADA;
133             break;
134         }
135 
136     }
137 
138     private void enviar1Comando() {
139         int ordem = (int) (Math.random() * ((4 - 0) + 1)) + 0;
140         MyMessage msg = new MyMessage(++numero, ordem);
141         com.escreverMensagem(msg);
142         comando = (ordem == 0) ? "Parar: false"
143                 : (ordem == 1) ? "Reta: 10"
144                         : (ordem == 2) ? "Direita: 0 45"
145                                 : (ordem == 3) ? "Esquerda: 0 45" : (ordem == 4) ? "Reta: -10" : "Parar: true";
146         gui.myPrint("Mensagem " + numComandos + ": [ " + msg.getNumero() + ", " + msg.getOrdem() + "] -> " + comando);
147         numComandos++;
148         long timestamp = System.currentTimeMillis();
149         while (System.currentTimeMillis() - timestamp < 500);
150     }
151 
152     private void enviarComando(int ordem) {
153         MyMessage msg = new MyMessage(++numero, ordem);
154         com.escreverMensagem(msg);
155         comando = (ordem == 0) ? "Parar: false"
156                 : (ordem == 1) ? "Reta: 10"
157                         : (ordem == 2) ? "Direita: 0 45"
158                                 : (ordem == 3) ? "Esquerda: 0 45" : (ordem == 4) ? "Reta: -10" : "Parar: true";
159         gui.myPrint("[ " + msg.getNumero() + ", " + msg.getOrdem() + "] -> " + comando);
160         long timestamp = System.currentTimeMillis();
161         while (System.currentTimeMillis() - timestamp < 500);
162     }
163 
164     private void inactivarComandos() {
165         bd.setComando1(false);
166         bd.setComandos16(false);
167         bd.setComandos32(false);
168         bd.setComandosIlimitados(false);
169         bd.setPararComandos(false);
170     }
171 }
172 
```

5.7. Anexo G – Classe “MyMessage”

```
1
2  public class MyMessage {
3
4      private int numero;
5      private int ordem;
6
7
8
9      public MyMessage(int numero, int ordem) {
10         this.numero = numero;
11         this.ordem = ordem;
12     }
13
14     public int getNumero() {
15         return numero;
16     }
17
18     public int getOrdem() {
19         return ordem;
20     }
21 }
```

5.8. Anexo H – Classe "CanalComunicacao"

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;

public class CanalComunicacao {

    // ficheiro
    File ficheiro;

    // canal que liga o conteúdo do ficheiro ao Buffer
    FileChannel canal;

    // buffer
    MappedByteBuffer buffer;

    //FileLock
    FileLock locker;

    // dimensão máxima em bytes do buffer
    final int BUFFER_MAX = 256; // 32 mensagens de 8 bytes

    private int put, get, number, order, lastNumber;

    public CanalComunicacao() {
        // cria um ficheiro com o nome comunicacao.dat
        ficheiro = new File(".\\comunicacao.dat");

        // cria um canal de comunicação de leitura e escrita
        try {
            canal = new RandomAccessFile(ficheiro, "rw").getChannel();
        } catch (FileNotFoundException e) {
            System.err.println("Canal error - RandomAccessFile(ficheiro, \"rw\")."
                    + "getChannel() not working: " + e.getMessage());
        }

        // mapeia para memória o conteúdo do ficheiro
        try {
            buffer = canal.map(FileChannel.MapMode.READ_WRITE, 0, BUFFER_MAX);
        } catch (IOException e) {
            e.printStackTrace();
        }

        put = 0;
        get = 0;
        number = 0;
        order = 0;
        lastNumber = 0;

        clearBuffer();
    }

```

```
56      public void escreverMensagem(MyMessage mensagem) {
57          try {
58              locker = canal.lock();
59          } catch (IOException e) {
60              System.err.println("FileLock error - canal.lock() not working: " + e.getMessage());
61          }
62
63          buffer.position(put);
64          buffer.putInt(mensagem.getNumero());
65          buffer.putInt(mensagem.getOrdem());
66
67          put = (put += 8) % BUFFER_MAX;
68
69          try {
70              locker.release();
71          } catch (IOException e) {
72              System.err.println("FileLock error - FileLock.release() not working: " + e.getMessage());
73          }
74      }
75
76      public MyMessage lerMensagem() {
77          try {
78              locker = canal.lock();
79          } catch (IOException e) {
80              System.err.println("FileLock error - canal.lock() not working: " + e.getMessage());
81          }
82
83          number = buffer.getInt(get);
84          order = buffer.getInt(get + 4);
85
86          if (number == lastNumber + 1) {
87              get = (get += 8) % BUFFER_MAX;
88              lastNumber = number;
89          }
90
91          try {
92              locker.release();
93          } catch (IOException e) {
94              System.err.println("FileLock error - FileLock.release() not working: " + e.getMessage());
95          }
96
97          return new MyMessage(number, order);
98      }
99
100     public void clearBuffer() {
101         for(int i = 0; i < BUFFER_MAX; i++) {
102             buffer.position(i);
103             buffer.put((byte) 0);
104         }
105     }
106
107     public void fecharCanal() {
108         try {
109             canal.close();
110         } catch (IOException e) {
111             System.err.println("Canal error - FileChannel.close() not working: " + e.getMessage());
112         }
113     }
114
```

5.9. Anexo I – Classe "MyRobotLego"

```
1
2  public class MyRobotLego {
3      public boolean OpenEV3(String s) {
4          System.out.println("OpenEV3 ligado" + s);
5          return true;
6      }
7      public void CloseEV3(){
8          System.out.println("CloseEV3");
9      }
10     public void CurvarDireita(int r, int a){
11         System.out.println("Curva à Direita: raio " + r + " angulo " + a);
12     }
13     public void CurvarEsquerda(int r, int a){
14         System.out.println("Curva à Esquerda: raio " + r + " angulo " + a);
15     }
16     public void Parar(boolean b){
17         System.out.println("Parar: " + b);
18     }
19     public void Reta(int d){
20         System.out.println("Reta: " + d);
21     }
22     public void OffsetEsquerdo(int offset){
23         System.out.println("Offset Esquerdo: " + offset);
24     }
25     public void OffsetDireito(int offset){
26         System.out.println("Offset Direito: " + offset);
27     }
28     public boolean Sensor(int input){
29         int r = (int) Math.round(Math.random() * 100);
30         if( r <= 10)
31             return true;
32         return false;
33     }
34 }
35
```