



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

Fundamentos de Sistemas Operativos
2º Trabalho Prático

Docente:

Jorge Pais

Autores:

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

Lisboa, 21 de dezembro de 2019

Índice

1.	Introdução.....	5
1.1.	Objetivos.....	7
2.	Interfaces Gráficas.....	9
2.1.	Interfaces Gráficas “GUIDancarino” e “GUICoreografo”	9
2.2.	Interface Gráfica “GUITP2”	11
3.	Comportamentos	13
3.1.	Coreógrafo.....	13
3.1.1.	Diagrama de Actividades.....	13
3.2.	Dançarino	15
3.2.1.	Diagrama de Actividades.....	15
3.3.	Canal de Comunicação	17
3.3.1.	Implementação.....	17
3.3.2.	Diagrama de Classes (UML) – CanalComunicacao e Dancarino.....	19
3.3.3.	Diagrama de Classes (UML) – CanalComunicacao e Coreografo	21
3.3.4.	Sincronização e Comunicação entre Coreógrafos e Dançarinos.....	23
3.4.	SpyRobot	25
3.4.1.	Implementação.....	25
3.4.2.	Diagrama de Classes (UML) – SpyRobot e Dancarino	26
3.5.	Eitar	27
3.5.1.	Sincronização e Comunicação entre Dançarino e Eitar	27
3.5.2.	Diagrama de Actividades.....	28
3.5.3.	Diagrama de Classes (UML) – Eitar e Dancarino	28
3.6.	Diagrama de Classes (UML) – Comportamentos e interface “GUITP2”	29
4.	Conclusões	31
5.	Bibliografia	33
6.	Anexos	35
6.1.	Anexo A – Classe “BDDancarino”	35
6.2.	Anexo B – Classe ” GUIDancarino”	37
6.3.	Anexo C – Classe “Dancarino”	43
6.4.	Anexo D – Classe “BDCoreografo”	47
6.5.	Anexo E – Classe “GUICoreografo”	49
6.6.	Anexo F – Classe “Coreografo”	53
6.7.	Anexo G – Classe “Ordem”	57
6.8.	Anexo H – Classe “MyMessage”	59

6.9.	Anexo I – Classe “Descriptor”	61
6.10.	Anexo J – Classe “BDCanal”	63
6.11.	Anexo K – Classe “CanalComunicacao”.....	65
6.12.	Anexo L – Classe “BDGUIT2”	71
6.13.	Anexo M – Classe “GUITP2”	73
6.14.	Anexo N – Classe “BDSpyRobot”	79
6.15.	Anexo O – Classe “GUISpyRobot”	81
6.16.	Anexo P – Classe “SpyRobot”	83
6.17.	Anexo Q – Classe “BDEvitar”	89
6.18.	Anexo R – Classe “Evitar”	91
6.19.	Anexo S – Classe “MyRobotLego”.....	93

Índice de Figuras

<i>Figura 1 - Interface gráfica "GUIDancarino"</i>	9
<i>Figura 2 - Interface gráfica "GUICoreografo"</i>	10
<i>Figura 3 - Interface Gráfica "GUITP2"</i>	11
<i>Figura 4 - Diagrama de Actividades do Coreografo</i>	13
<i>Figura 5 - Diagrama de Actividades do Dancarino</i>	15
<i>Figura 6 - UML: CanalComunicacao e Dancarino</i>	19
<i>Figura 7 - UML: CanalComunicacao e Dancarino (detalhado)</i>	20
<i>Figura 8 - UML: CanalComunicacao e Coreografo</i>	21
<i>Figura 9 - UML: CanalComunicacao e Coreografo (detalhado)</i>	22
<i>Figura 10 - Interface Gráfica "GUISpyRobot": botão "Gravar" e botão "Parar"</i>	26
<i>Figura 11 - UML: SpyRobot e Classes relacionadas</i>	26
<i>Figura 12 - Diagrama de Actividades do Evitar</i>	28
<i>Figura 13 - UML: Dancarino e Evitar</i>	28
<i>Figura 14 - UML: GUITP2 e restantes classes</i>	29

Índice de Tabelas

Tabela 1 - Valores da variável Ordem correspondentes a cada comando do robot.....6

1. Introdução

Este trabalho é uma continuação do primeiro trabalho prático da disciplina de Fundamentos de Sistemas Operativos. É utilizado o trabalho anterior como base neste trabalho com o intuito de aprofundar os conceitos de processos e sincronização entre estes.

Agora, em vez de processos independentes, o pretendido é aprofundar os conceitos de processos-leves/tarefas, com o intuito de estudar multithreading¹, onde a aplicação está a correr vários fios de execução em (aparentemente) simultâneo para fazer tarefas diferentes que não é capaz de fazer em simultâneo.

Um exemplo é o caso do Dançarino no primeiro trabalho prático, ele não é capaz de estar a escutar o canal ao mesmo tempo que envia comandos para o robot, tem de terminar o fio de execução responsável pela escuta para passar para o fio de execução do envio de comandos. Tendo duas tarefas constituintes do Dançarino, uma responsável pela escuta e outra pelo envio, só uma tarefa é que está a trabalhar de cada vez, mas é possível alternar as tarefas que estão a cumprir instruções, a meio dos seus fios de execução, ou seja, aparentemente estão a trabalhar em simultâneo.

Para a criação das tarefas Java, podíamos ter derivado da classe Thread ou implementando a interface Runnable. Optou-se pela implementação da interface Runnable² por terem comportamentos idênticos mas do ponto de vista de boas práticas de programação, visto que não queremos alterar o comportamento da classe Thread, mas sim utilizá-la, é mais correto implementar a interface que estender a classe, para além de assim, ser possível estender de outra classe se necessário porque em Java só se pode estender de uma classe mas pode-se implementar as interfaces que forem necessárias.

O Canal de Comunicação, que permitia comunicação entre processos independentes através do uso de um ficheiro e de um FileLock para garantir que apenas um processo modificava o ficheiro de cada vez, passou a ser um canal de comunicação entre tarefas pertencentes a um mesmo processo. O próprio Canal passou a ser uma tarefa, responsável por gerir o acesso das tarefas ao buffer, sincronizando as leituras de diferentes Dançarinos com as escritas dos Coreógrafos, mantendo a integridade dos dados, garantindo que todas as mensagens são lidas por todos os Dançarinos.

¹ Fonte: <https://www.geeksforgeeks.org/multithreading-in-operating-system/>

² Fonte: <https://stackoverflow.com/questions/541487/implements-runnable-vs-extends-thread-in-java>

As mensagens (Tabela 1) permanecem as mesmas do primeiro trabalho prático, com a única alteração de que agora são guardadas e manipuladas na classe *Enumeration Ordem*, para facilitar a compreensão e utilização destas.

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

1.1. Objetivos

- Desenvolvimento de aplicações multitarefa em Java.

Os Dançarinos, Coreógrafos e Canal de Comunicação passaram a ser tarefas constituintes duma aplicação responsável pela sua instanciação, a GUITP2. Novas classes como o Evitar e o SpyRobot também são tarefas.

- Gestão do ciclo de vida de tarefas.

Terminando o processo-pai, todas as tarefas terminam. Terminando uma tarefa, as outras tarefas continuam a sua execução. Quando uma tarefa fica bloqueada à espera de um recurso, a aplicação prossegue o seu funcionamento, assim como as outras tarefas.

- Comunicação e sincronização entre tarefas Java.

A comunicação entre tarefas é feita através de uma instância do Canal de Comunicação partilhada por todas as tarefas para permitir a troca e manipulação de informação e sincronização entre tarefas é feita com auxílio de semáforos Java.

- Modelo Produtor-Consumidor.

Aumentou-se a complexidade do modelo Produtor-Consumidor implementado, que apenas era utilizado numa relação de um para um sem garantir a integridade dos dados, para ser capaz de operar com vários Produtores e vários Consumidores em simultâneo, sem haver perda de informação.

- Manipulação de ficheiros em JAVA através de InputStreams e OutputStreams numa aplicação multiprocessso.

Foi criada uma tarefa autónoma que simula o funcionamento da biblioteca RobotLegoEV3, tendo métodos idênticos que são enviados para a biblioteca real, de modo a manter o comportamento esperado, mas que através do uso de uma interface gráfica, permita operações de escrita e leitura de ficheiros com base no nome do ficheiro escolhido pelo utilizador, utilizando os comandos intercetados entre o Dançarino e o robot.

- Desafio Adicional – Comportamento Evitar

Foi criada uma tarefa complementar ao Dançarino, o Evitar, que também comunica com o robot, sendo necessário sincronizar o acesso ao robot para o envio de comandos entre tarefas, e que permite terminar a execução do Dançarino quando a condição de choque no robot se realizar, através do uso do sensor de toque integrado.

2. Interfaces Gráficas

2.1. Interfaces Gráficas “GUIDancarino” e “GUICoreografo”

As interfaces gráficas “GUIDancarino” (figura 1) e “GUICoreografo” (figura 2) são as interfaces dos comportamentos “Dancarino” e “Coreografo”, respectivamente. Estas interfaces gráficas não passaram por nenhuma alteração na sua implementação desde o primeiro trabalho prático desta disciplina.

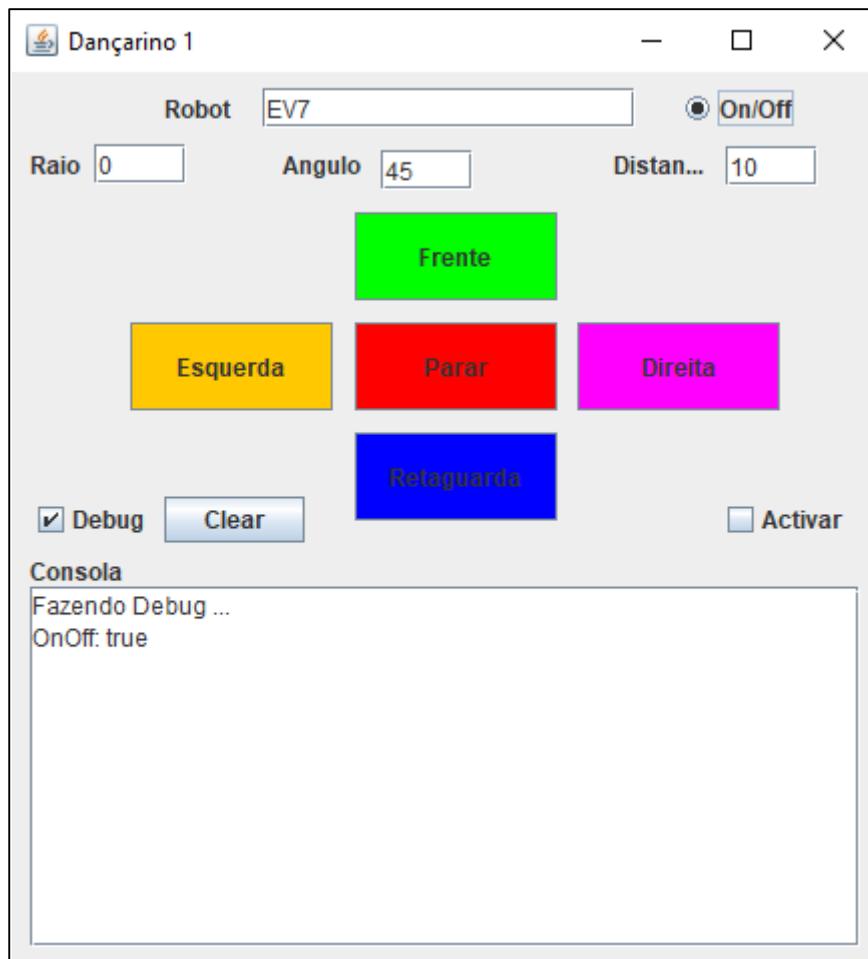


Figura 1 - Interface gráfica "GUIDancarino"

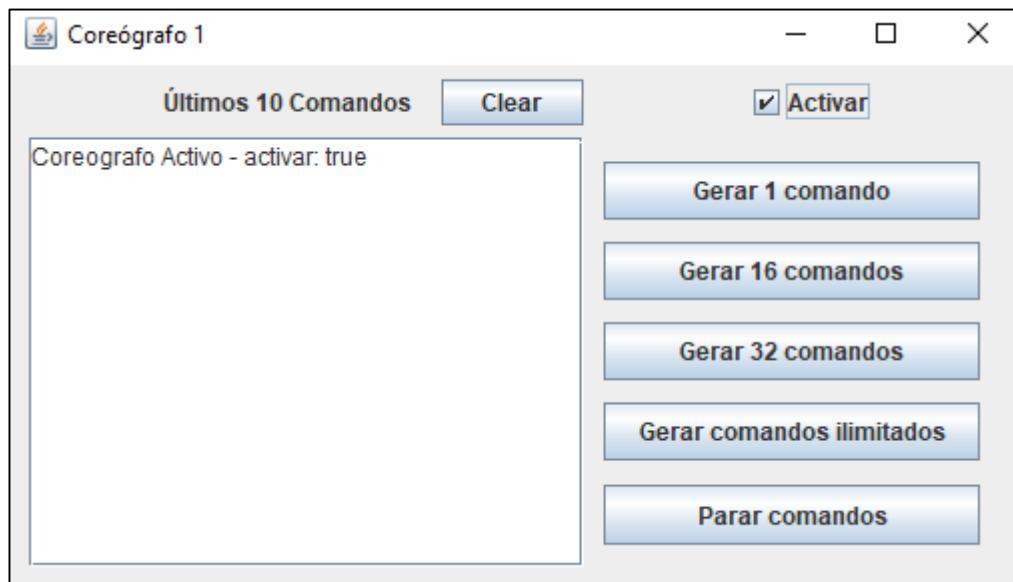


Figura 2 - Interface gráfica "GUICoreografo"

2.2. Interface Gráfica “GUITP2”

A GUITP2 é o processo que engloba todas as tarefas criadas. Através da sua interface gráfica (figura 3), permite ao utilizador adicionar ou remover Coreógrafos a uma lista e Dançarinos a outra, para que o utilizador possa iniciar esses comportamentos através dos seus respetivos botões. As listas criadas também são mostradas na interface para que o utilizador saiba quantos processos serão iniciados quando ele premir o botão de inicialização.

A GUITP2 instancia o Canal de Comunicação e fornece-o às tarefas que instanciar para que seja possível haver comunicação entre tarefas através do uso do canal.

A interface também dispõe de duas checkbox's, que permitem, após a inicialização dos comportamentos, ativar todos os comportamentos: uma para ativar todos os Dançarinos e outra para os Coreógrafos, com o intuito de poupar o trabalho ao utilizador de ir a cada interface individual de cada comportamento ativá-los um a um.

A última funcionalidade da interface é uma zona de logging, que apresenta informações relevantes ao utilizador sobre a instanciação dos comportamentos.



Figura 3 - Interface Gráfica "GUITP2"

3. Comportamentos

3.1. Coreógrafo

O Coreógrafo passou a ser uma tarefa através da implementação da interface Runnable. Através do uso da interface, passou a ser obrigatório a declaração do método run(), responsável por correr o autómato do Coreógrafo em ciclicamente, até ser notificado para parar a sua execução.

Foi adicionado um semáforo que bloqueia o Coreógrafo se na sua interface gráfica não estiver ativado. Assim, se não estiver ativado, o Coreógrafo não consome recursos.

Por fim, deixou de instanciar um Canal de Comunicação, visto que este é fornecido pelo processo-pai.

3.1.1. Diagrama de Actividades

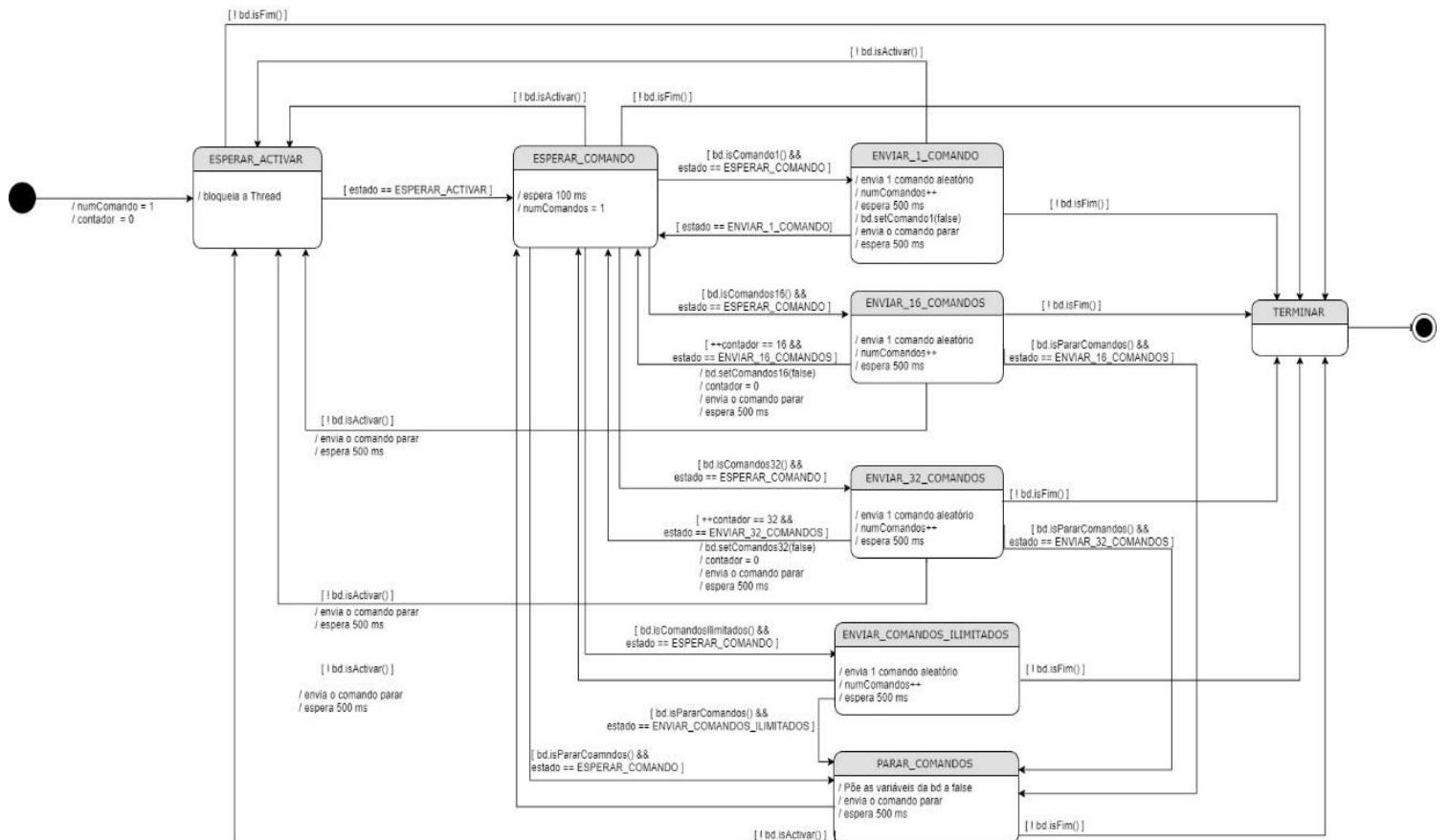


Figura 4 - Diagrama de Actividades do Coreógrafo

3.2. Dançarino

De forma idêntica ao Coreógrafo, foram implementadas as mudanças referidas ao Dançarino, mas o seu código também foi adaptado para ser capaz de trabalhar em conjunto com o comportamento Evitar e utilizar o SpyRobot em vez do RobotLeggo.

3.2.1. Diagrama de Actividades

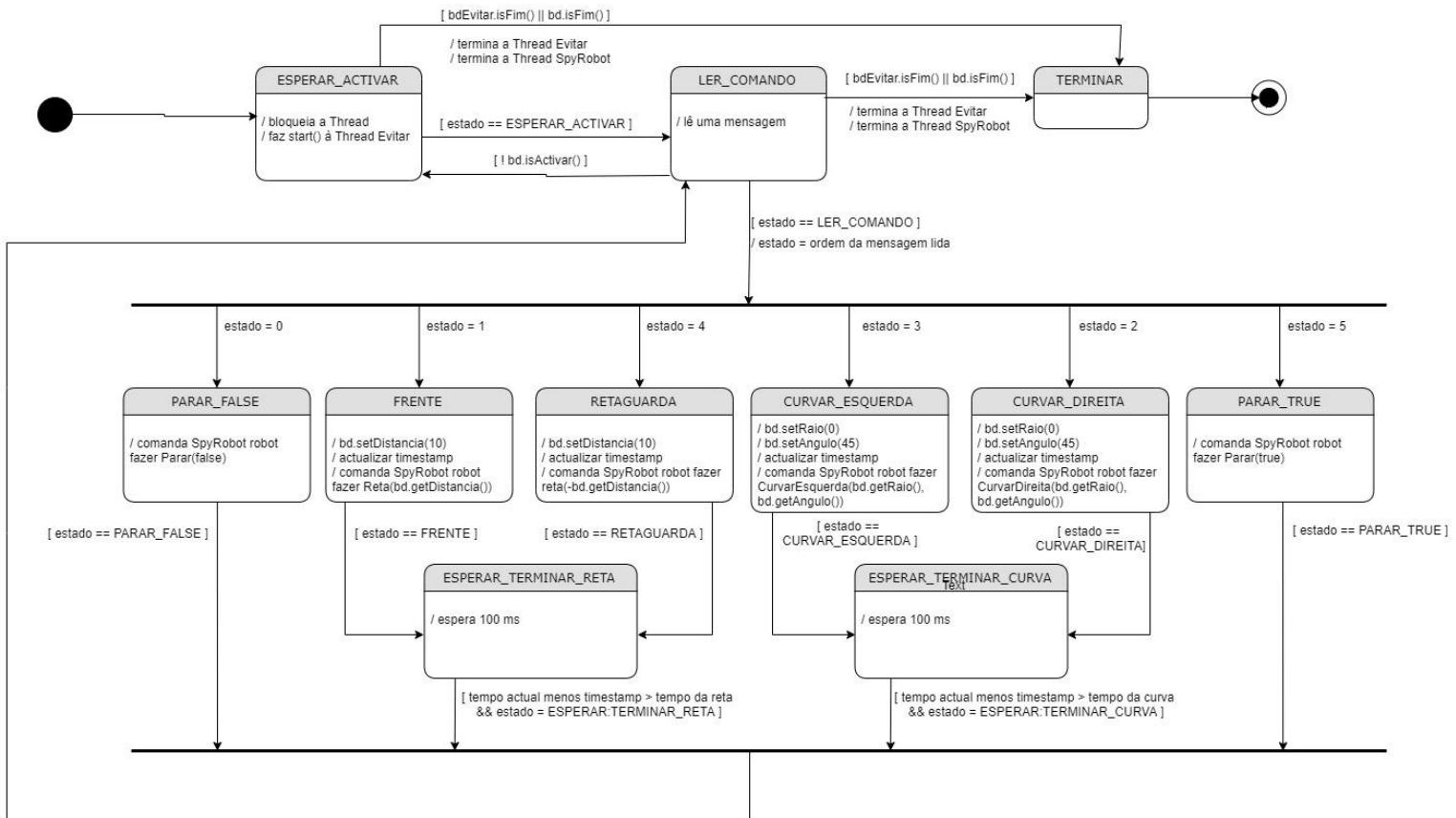


Figura 5 - Diagrama de Actividades do Dancarino

3.3. Canal de Comunicação

3.3.1. Implementação

O Canal de Comunicação foi baseado no canal desenvolvido no primeiro trabalho prático, mas com diferenças fundamentais devido aos novos objetivos e restrições impostas, assim como melhoramentos de performance.

Deixou-se de utilizar a classe `MappedByteBuffer` para a criação do buffer circular. No primeiro trabalho prático, o objetivo era permitir a comunicação entre processos distintos, por isso era utilizado um ficheiro onde ficava armazenada a informação para ser consultada pelos dois processos. Como neste trabalho os comportamentos são tarefas que fazem parte de um processo-pai (GUITP2), todos os Dançarinos e Coreógrafos possuem a mesma instância do Canal de Comunicação, providenciado pelo processo-pai. Assim sendo, a informação é partilhada por todos os processos através desse objeto, o que faz com que o acesso a um ficheiro para consultar a informação seja menos eficiente. Em alternativa, pode-se utilizar um Buffer “normal” como um `ByteBuffer` ou uma `ArrayList` para armazenamento e modificação da informação partilhada por todos os processos. Nesta implementação, optou-se pelo `ByteBuffer` visto que o algoritmo de acesso ao buffer já estava desenhado previamente no primeiro trabalho prático.

Outra diferença no canal é a integridade dos dados: no primeiro trabalho o Coreógrafo escrevia para o ficheiro sem saber se o Dançarino já tinha consumido a mensagem ou não. Agora, tem-se vários Dançarinos e quer-se garantir que todos leem as mesmas mensagens, sem mensagens repetidas ou perda de informação. É necessário haver então um sincronismo no acesso à informação.

Como as mensagens dos Coreógrafos têm de ser lidas por todos os Dançarinos, é necessário saber quantos Dançarinos estão a escutar o canal de modo a garantir a integridade dos dados. Foram criados métodos de registo no canal para ser possível controlar quantos escritores e leitores existem no momento. Isto significa que o canal assegura a distribuição correta da informação entre Coreógrafos e Dançarinos a partir do momento do registo.

Para fazer a gestão do acesso aos dados, o Canal de Comunicação também passou a ser uma tarefa, que por definição está bloqueada (por um semáforo Java iniciado a zero permits) para não consumir recursos, e, quando outras tarefas pretendem aceder ao buffer, é desbloqueada e o seu autómato serve para sincronizar as tarefas, controlando as posições do buffer que podem ser lidas e escritas através de semáforos Java.

O Canal de Comunicação dispõe de dois métodos de manipulação do buffer, um de leitura, usado pelos Dançarinos, e um de escrita, usado pelos Coreógrafos. Esses métodos são encarregues de:

- validar se a tarefa que está a chamar o método está registada no canal, através de um identificador único dado no momento do registo no canal;
- manipular o buffer através da escrita ou leitura de uma mensagem, caso o canal o permitir;
- notificar o canal que foi efetuada uma operação, registando a tarefa que fez a modificação;
- e desbloquear o autómato do canal para este proceder à gestão dos semáforos de todas as tarefas.

As posições de cada tarefa no buffer (a posição correspondente à mensagem que vão ler/escrever) estão armazenadas no canal. Para o coreógrafo, é armazenada a posição da última mensagem escrita numa variável. Isto chega para garantir o sincronismo entre Coreógrafos. Para os Dançarinos a complexidade aumenta. Segundo a implementação tomada, cada Dançarino tem um semáforo para controlar o consumo de mensagens, estando cada Dançarino dependente tanto dos Coreógrafos como dos outros Dançarinos. Além disso, Dançarinos diferentes podem estar em posições diferentes a consumir mensagens diferentes. Para gerir os Dançarinos, foi criada uma classe auxiliar para o Canal de Comunicação armazenar a informação de Dançarinos distintos – classe Descritor.

Para cada Dançarino, é criado um Descritor que contém um semáforo, que corresponde às mensagens que o Dançarino tem para ler, a posição onde se encontra no buffer e um ID, dado no seu registo do canal, que permite autenticar o Dançarino, sempre que este quiser ler.

O autómato do Canal de Comunicação bloqueia-se caso não haja chamadas provenientes de outras tarefas. Quando desbloqueado, verifica se foram escritas mensagens (quando uma mensagem é escrita, é adicionado o ID da tarefa que escreveu a uma lista de escritores) e de seguida verifica se foram lidas mensagens (processo idêntico à verificação de escrita) mudando para o estado correspondente.

No caso de leitura, atualiza o semáforo que corresponde às posições livres do buffer. Caso todos os Dançarinos tenham lido uma determinada mensagem do Coreógrafo, essa mensagem deixa de ser necessária e o Coreógrafo pode escrever nessa posição. Por fim, remove o ID do leitor da lista que regista as tarefas que acederam ao canal.

No caso da escrita, atualiza o semáforo de cada Dançarino para estes poderem ler mais uma posição no buffer sem ficarem bloqueados à espera de novas mensagens (ou caso estejam bloqueados, são desbloqueados e leem a posição que tentaram aceder antes de ficar à espera de permissão para uma nova mensagem).

O acesso ao buffer é limitado por um semáforo de exclusão mútua, ou seja, é um semáforo inicializado com uma permit para apenas uma tarefa poder mexer no buffer de cada vez.

Segundo esta lógica, também foi possível seguir as restrições impostas de que se o buffer estiver vazio, ou seja, o semáforo do seu Descriptor tem zero permits, um Dançarino que tente ler fica bloqueado até um Coreógrafo escrever, e que se o buffer estiver cheio, ou seja, o semáforo “livres” inicializado com o número total de posições do buffer encontra-se com zero permits, um Coreógrafo que tente escrever fica bloqueado até que todos os Dançarinos tenham lido a posição mais antiga.

3.3.2. Diagrama de Classes (UML) – CanalComunicacao e Dancarino

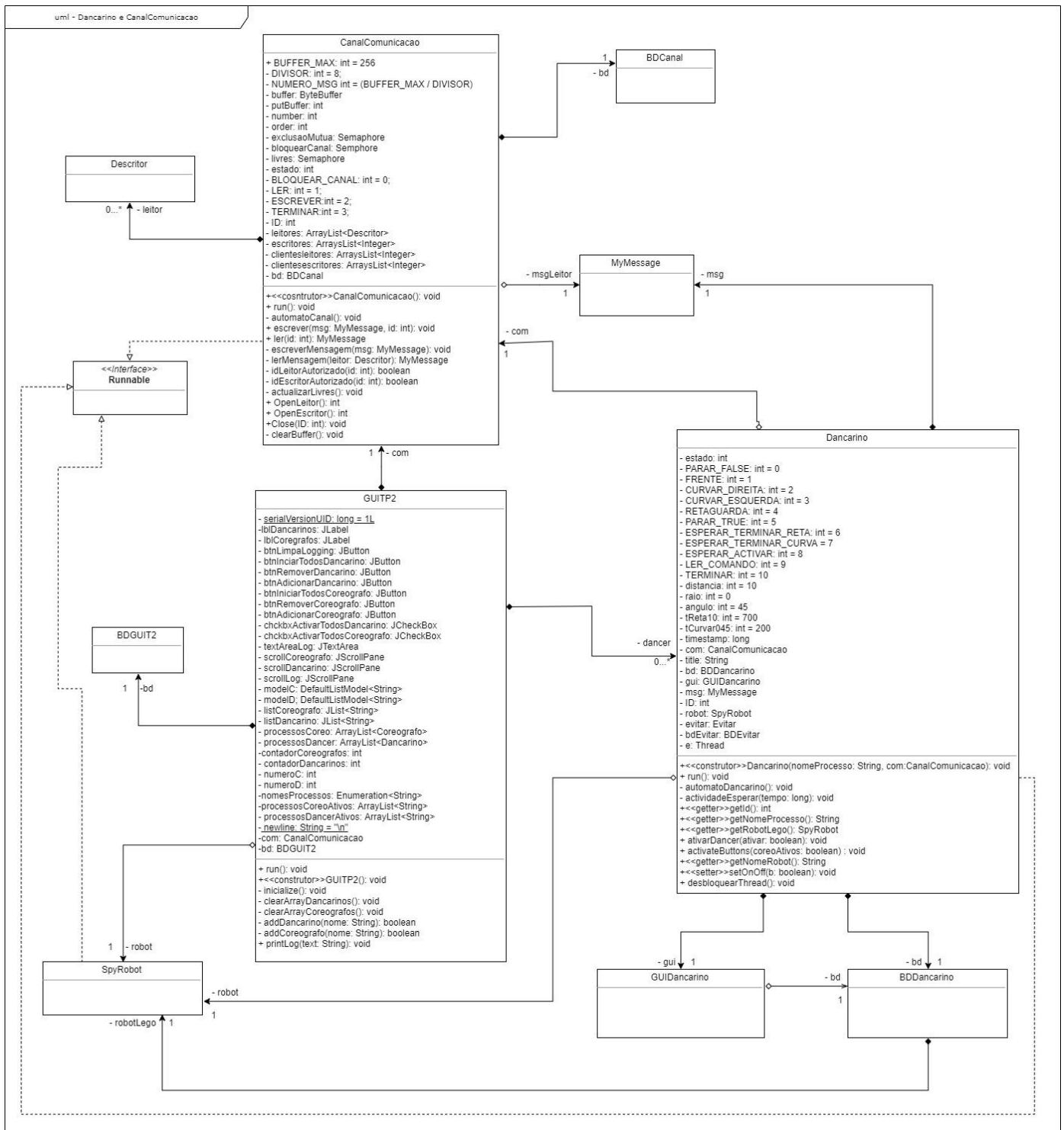


Figura 6 - UML: CanalComunicacao e Dancarino

Versão mais detalhada do UML:

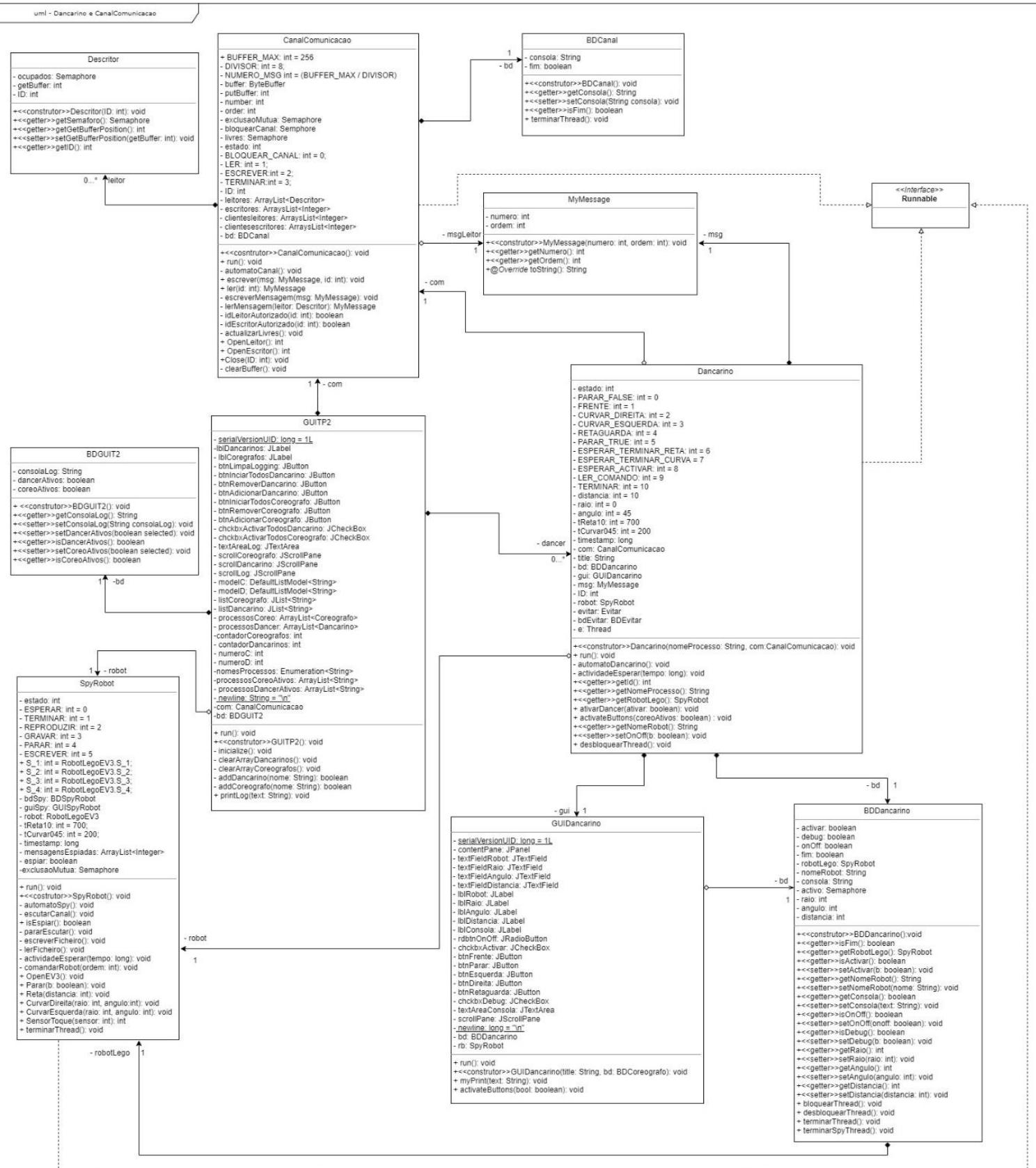


Figura 7 - UML: CanalComunicacao e Dancarino (detalhado)

3.3.3. Diagrama de Classes (UML) – CanalComunicacao e Coreografo

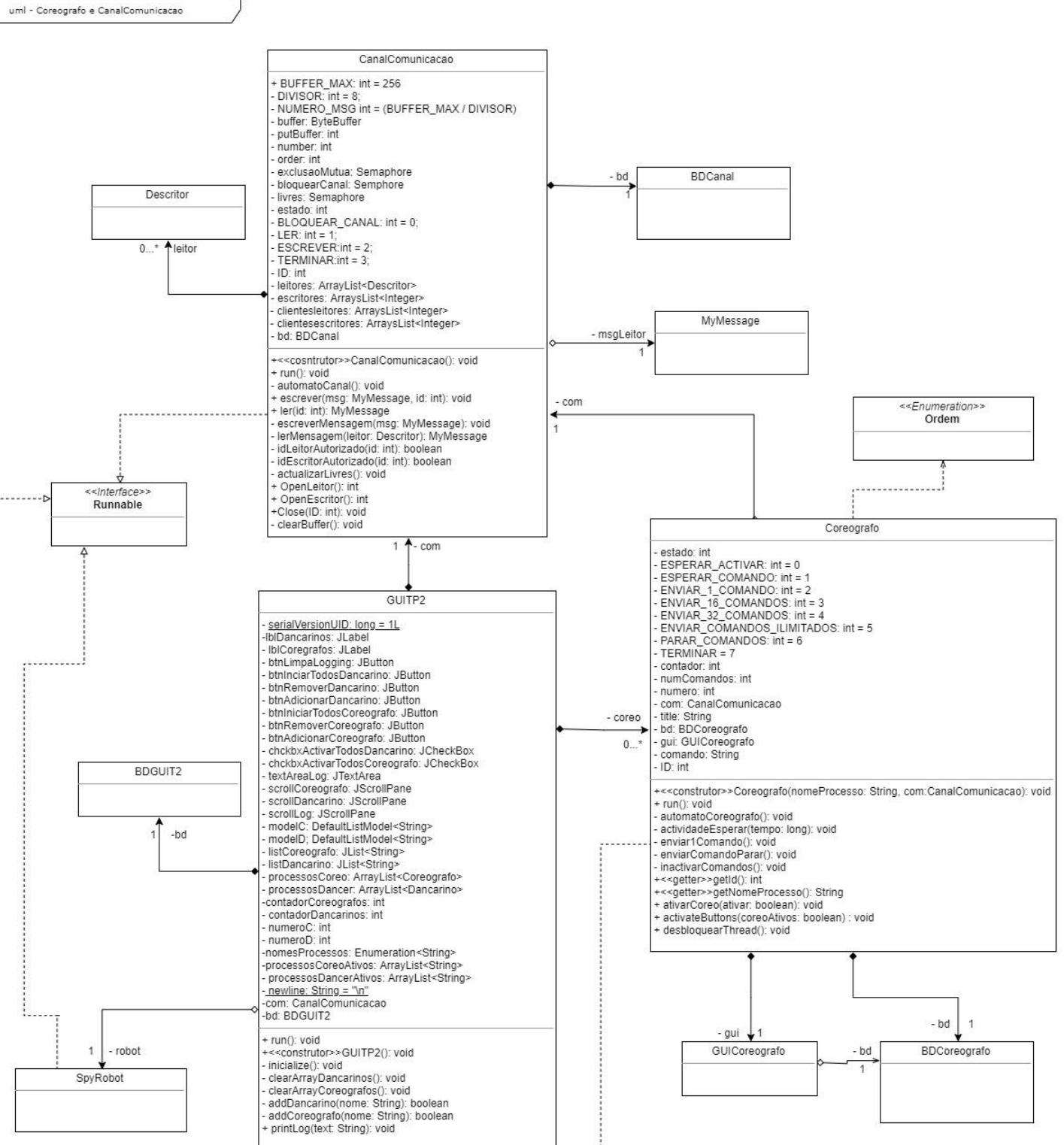


Figura 8 - UML: CanalComunicacao e Coreografo

Versão mais detalhada do UML:

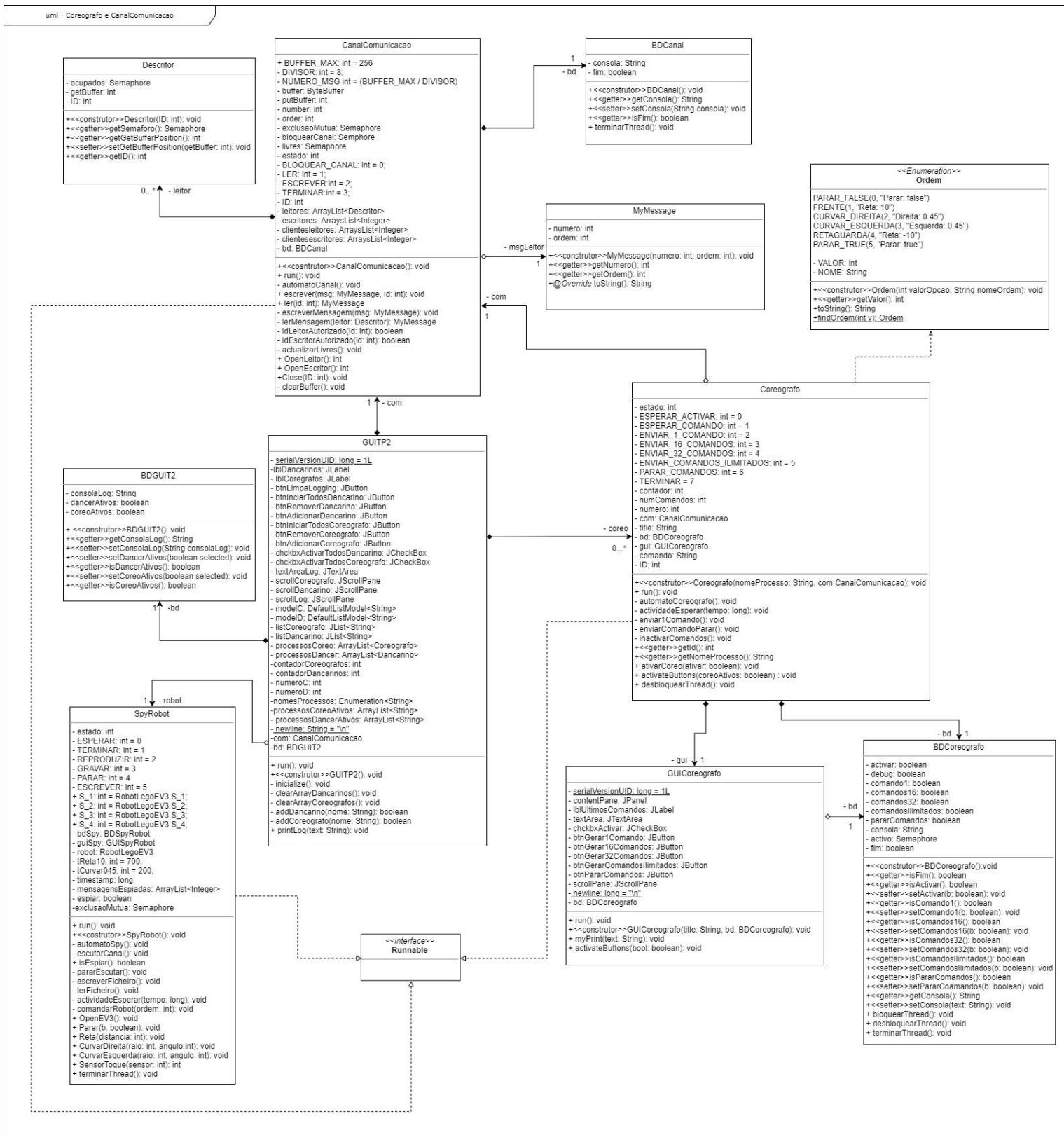


Figura 9 - UML: CanalComunicacao e Coreografo (detalhado)

3.3.4. Sincronização e Comunicação entre Coreógrafos e Dançarinos

Devido à transformação de processos independentes em processos-leves foi necessário resolver problemas como sincronismo no acesso a recursos e garantir integridade de dados, por isso foram utilizados semáforos Java para permitir o bom funcionamento do programa.

Um semáforo Java³ é um objeto Java que bloqueia a Thread que chama o método *acquire()* quando não tem permissões suficientes. Uma permissão é acrescentada ao semáforo quando uma thread faz *release()* se outras threads não estiverem bloqueadas no *acquire()* desse semáforo, ou se estiverem, é dada uma permissão a uma thread bloqueada. O semáforo é inicializado com o número de permissões que pode conceder às threads sem as bloquear. Serve para garantir o número de threads que podem aceder a um recurso.

O uso de semáforos na aplicação permite:

- controlar o acesso ao robot no caso do Dançarino e Evitar;
- gerir as posições do buffer que podem ser lidas ou escritas para todos os Dançarinos e Coreógrafos, bloqueando as tarefas se não for possível ler/escrever nesse momento;
- apenas permitir a uma tarefa interagir com o buffer de cada vez;
- poupar recursos, desbloqueando a tarefa Canal de Comunicação apenas quando é necessário efetuar operações de gestão do canal;
- No caso do Canal de Comunicação, garante que o Coreógrafo pode escrever uma mensagem porque existe espaço no buffer e que o Dançarino pode ler uma mensagem porque o Coreógrafo já escreveu;
- No caso de um recurso de acesso exclusivo, é criado um semáforo com apenas uma permit, garantindo que apenas uma thread utiliza o recurso em simultâneo.

³Fonte:

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

3.4. SpyRobot

3.4.1. Implementação

O SpyRobot foi uma tarefa criada com o intuito de intercetar os comandos que o Dançarino envia para o robot, segundo uma abordagem *man-in-the-middle*⁴, para gravar os comandos recebidos num ficheiro e ler ficheiros escritos para o robot voltar a executá-los.

O Dançarino envia os comandos para o SpyRobot (pensando que se trata do RobotLego) e o SpyRobot envia para o RobotLego executar os comandos, no entanto, como intercepta os comandos que o Dançarino envia antes de estes serem executados pelo robot, permite guardar os comandos que são enviados num ficheiro. Também permite ler o ficheiro que foi escrito (ou outro previamente) e identificar os comandos escritos e enviá-los para o robot.

A escrita é feita através da classe `FileOutputStream` que recebe um ficheiro como parâmetro. Escrevem-se as ordens através do método `write()` da classe.

A leitura é feita através da classe `InputStream` que também recebe um ficheiro como parâmetro. Enquanto existir bytes para serem lidos da stream, cria uma mensagem e envia para o robot.

O SpyRobot contém uma réplica dos métodos do RobotLego para se passar por ele e um autómato para as suas atividades próprias.

O autómato do SpyRobot está interligado com a sua interface gráfica. Por predefinição espera por ordens. Quando o utilizador clica no botão “Gravar”, da interface gráfica do SpyRobot “GUISpyRobot” (figura 10), ativa um booleano que faz com que sempre que o Dançarino envie uma mensagem para o RobotLego (através do SpyRobot) essa mensagem seja armazenada, até o utilizador clicar novamente no botão para parar de gravar.

Depois disso, passa para a atividade de escrita, onde passa as mensagens espiadas para um ficheiro definido também pelo utilizador na interface gráfica. Quando termina a escrita, volta a esperar por ordens.

O utilizador pode também clicar no botão “Reproduzir” que faz com que o autómato do SpyRobot faça a atividade de reprodução, em que faz a leitura do ficheiro que o utilizador escolheu na interface gráfica, da mesma maneira que escolhe o nome do ficheiro na escrita, e lê os comandos armazenados no ficheiro e envia-os para o robot.

⁴ Fonte: https://pt.wikipedia.org/wiki/Ataque_man-in-the-middle

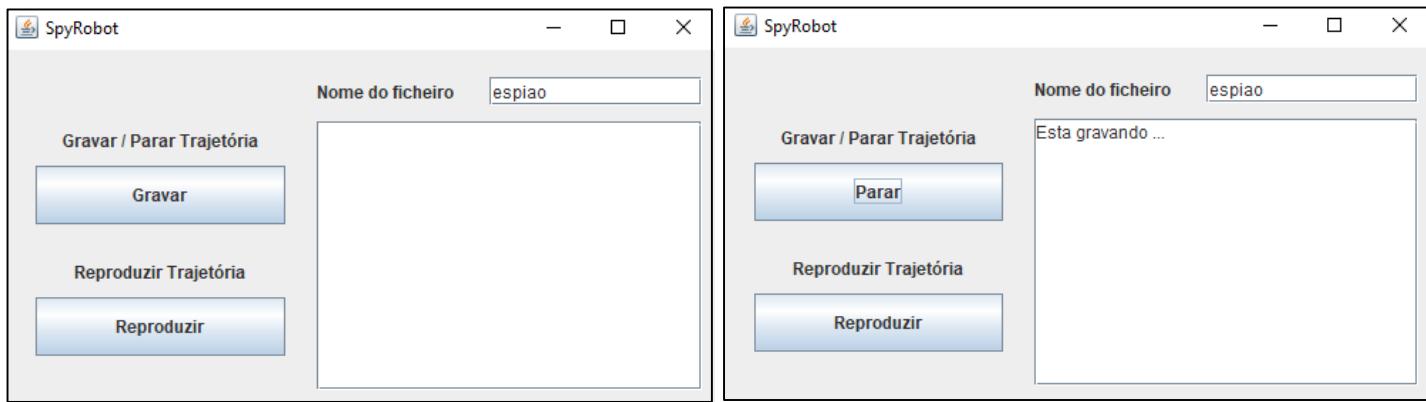


Figura 10 - Interface Gráfica "GUISpyRobot": botão "Gravar" e botão "Parar"

3.4.2. Diagrama de Classes (UML) – SpyRobot e Dancarino

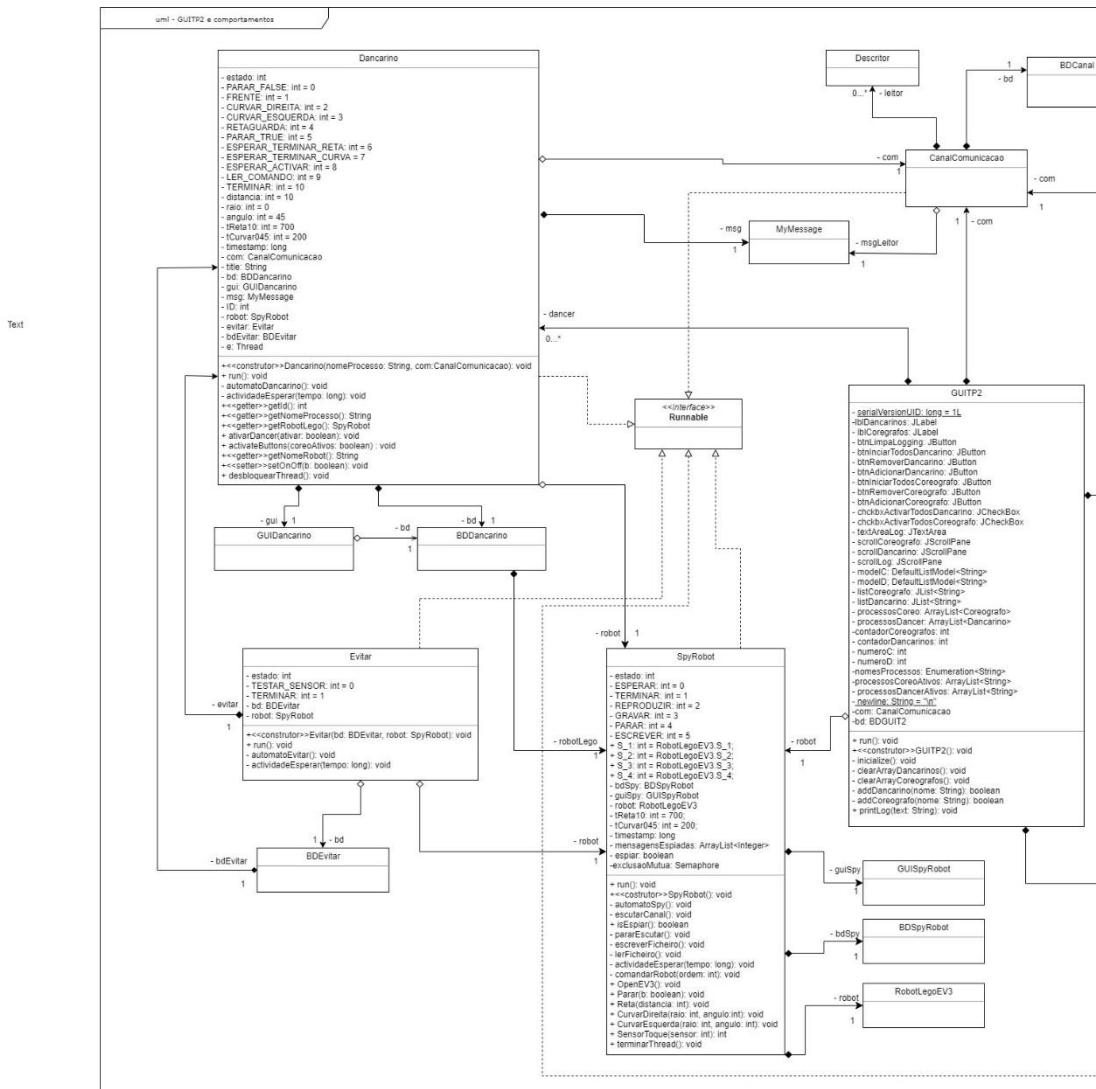


Figura 11 - UML: SpyRobot e Classes relacionadas

3.5. Evitar

O Comportamento Evitar é uma tarefa que tem a função de testar o sensor de toque do robot de 200 em 200 milissegundos para saber se o robot bateu, e, caso assim seja, deve enviar o comando “parar” para o robot parar de executar comandos e informar o Dançarino para terminar a sua execução.

É um comportamento complementar ao Dançarino, ou seja, por cada Dançarino existe obrigatoriamente um Evitar. Do ponto de vista de programação, significa que o Dançarino tem um objeto do tipo Evitar que é instanciado quando o Dançarino é iniciado. Por isso, a tarefa Dançarino é o processo-pai do Evitar assim como a tarefa GUITP2 é o processo-pai de todos os Comportamentos e do Canal de Comunicação.

3.5.1. Sincronização e Comunicação entre Dançarino e Evitar

Como ambos comunicam com o robot, o Dançarino passa no construtor do Evitar a referência do robot que utiliza, para partilharem o mesmo robot, assim como a classe auxiliar BDEvitar, onde são armazenadas as variáveis do Evitar, permitindo assim a comunicação entre o Evitar e o Dançarino. Quando o Evitar detetar o toque, ativa um booleano na sua classe auxiliar e o Dançarino faz periodicamente a verificação desse booleano para saber se o Evitar mandou terminar a execução do Dançarino.

O Dançarino envia comandos para o RobotLego (comandos que foi ler ao Canal de Comunicação, que por sua vez foram escritos pelos Coreógrafos) enquanto o Evitar interroga o robot para saber se o robot colidiu com algo.

O robot informa se colidiu através de um sensor de toque incorporado no robot, e é possível saber se colidiu através de um método da biblioteca RobotLegoEV3 *int SensorToque(int porto)* em que o porto está ligado ao sensor de toque (no nosso caso: o porto representado pela constante S_1).

Como tanto o Dançarino como o Evitar comunicam com o robot, é necessário haver sincronismo entre os comportamentos, pois não é possível o acesso ao robot ao mesmo tempo nem enviar dois comandos para o robot em simultâneo e garantir o seu funcionamento correto.

Como o acesso ao robot é exclusivo, foi implementado um semáforo de exclusão mútua (semáforo com apenas uma permissão(permit), que significa que apenas uma tarefa pode aceder ao recurso de cada vez).

Os processos, aquando a comunicação com o robot, fazem *acquire()* ao semáforo de exclusão mútua, e só após o envio do comando é que fazem *release()*, garantindo assim que apenas um comando é enviado de cada vez.

3.5.2. Diagrama de Actividades

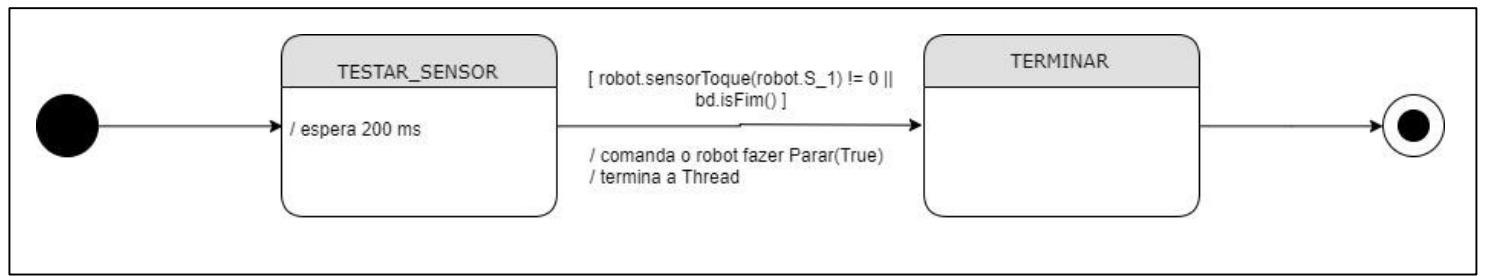


Figura 12 - Diagrama de Actividades do Evitar

3.5.3. Diagrama de Classes (UML) – Evitar e Dancarino

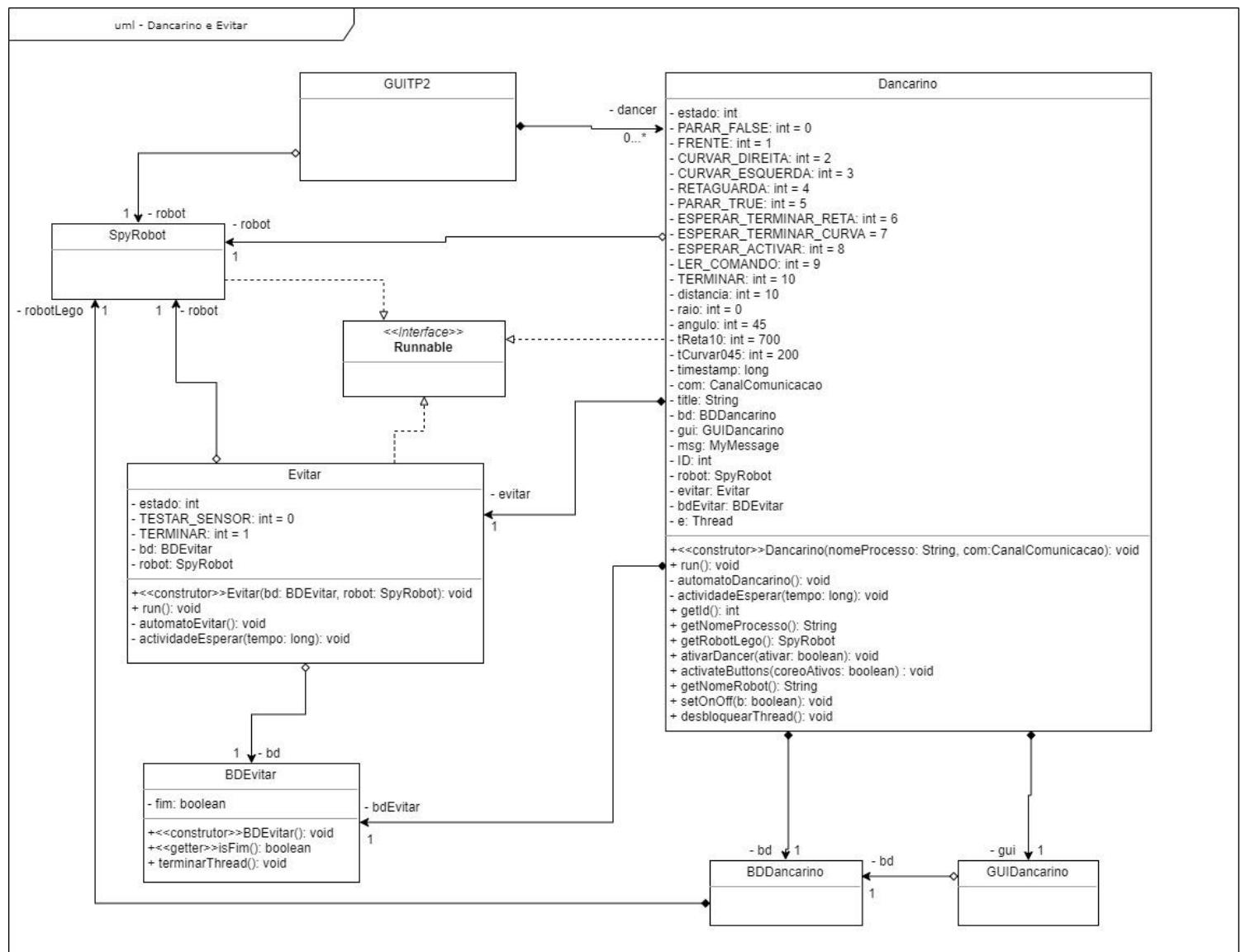


Figura 13 - UML: Dancarino e Evitar

3.6. Diagrama de Classes (UML) – Comportamentos e interface “GUITP2”

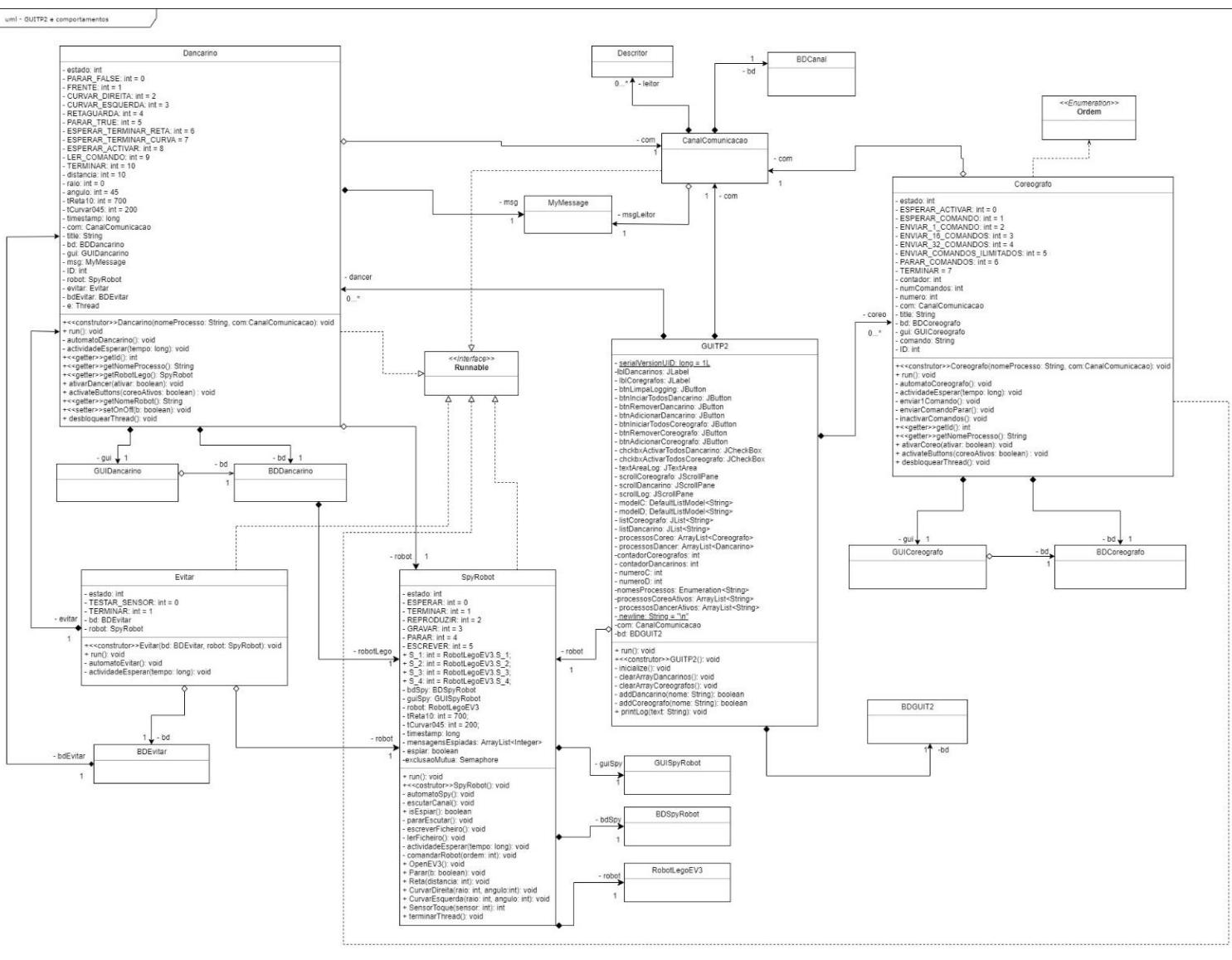


Figura 14 - UML: GUI TP2 e restantes classes

4. Conclusões

Este trabalho permitiu aprender a fazer a separação de uma aplicação em tarefas; ter vários objetivos em simultâneo e delegar o código pretendido para atingir cada objetivo em tarefas para permitir a execução dos algoritmos/autómatos em pseudo-paralelismo.

No desenvolvimento do Canal de Comunicação, aprendemos a sincronizar a partilha de informação entre tarefas.

Através de semáforos, foi possível limitar o acesso das tarefas a recursos, sincronizar diferentes tarefas, e poupar recursos do computador bloqueando tarefas não tinham de executar até certa altura.

Com o SpyRobot estudámos o funcionamento das classes FileInputStream e FileOutputStream para modificação de ficheiros.

Os diagramas de atividades e UML permitiram estruturar a lógica do programa e simplificar a implementação do código.

Após o desenvolvimento da aplicação, o programa foi testado utilizando múltiplos robots.

A comunicação entre os Coreógrafos e Dançarinos através do Canal de Comunicação estava a funcionar corretamente, mas, embora as interfaces de todos os Dançarinos tivessem recebido as mesmas mensagens, os robots apareciam não estarem a executar os mesmos comandos (ou pelo menos estarem a executá-los de forma sincronizada).

Com o auxílio do SpyRobot, foi possível guardar um conjunto de comandos e mandá-los para um robot executar. Como o comportamento do robot não era constante durante execuções repetidas do mesmo conjunto de comandos, determinámos que o problema não se encontrava no Canal.

Como havia a possibilidade do SpyRobot não guardar/reproduzir corretamente, testámos o envio direto de comandos predefinidos para o robot e analisar o seu comportamento. Verificámos que mesmo assim o comportamento dos robots não era coerente, então tentámos correr a aplicação noutras computadores, poderia ser um problema de Bluetooth, mas sem sucesso, por isso não foi possível determinar a origem do nosso problema.

5. 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

6. Anexos

6.1. Anexo A – Classe “BDDancarino”

```
1 import java.util.concurrent.Semaphore;
2
3 public class BDDancarino {
4     private SpyRobot robotLego;
5     private String nomeRobot, consola;
6     private boolean onOff, debug;
7     private int raio, angulo, distancia;
8     private boolean activar;
9     private Semaphore activo;
10    private boolean fim;
11
12    public BDDancarino() {
13        robotLego = new SpyRobot();
14        new Thread(robotLego).start();
15        nomeRobot = new String("EV7");
16        onOff = false;
17        debug = true;
18        raio = 0;
19        angulo = 45;
20        distancia = 10;
21        consola = new String("Fazendo Debug ... \n");
22        activar = false;
23        activo = new Semaphore(0);
24        fim = false;
25    }
26
27    public boolean isFim() {
28        return fim;
29    }
30
31    public SpyRobot getRobotLego() {
32        return robotLego;
33    }
34
35    public boolean isActivar() {
36        return activar;
37    }
38    public void setActivar(boolean activar) {
39        this.activar = activar;
40    }
41
42    public String getNomeRobot() {
43        return nomeRobot;
44    }
45    public void setNomeRobot(String nomeRobot) {
46        this.nomeRobot = nomeRobot;
47    }
48
49    public String getConsola() {
50        return consola;
51    }
52    public void setConsola(String consola) {
53        this.consola = consola;
54    }
55
56    public boolean isOnOff() {
57        return onOff;
58    }
59    public void setOnOff(boolean onOff) {
60        this.onOff = onOff;
61    }
62
63    public boolean isDebug() {
64        return debug;
65    }
```

```

66⊕    public void setDebug(boolean debug) {
67        this.debug = debug;
68    }
69⊕    public int getRaio() {
70        return raio;
71    }
72⊕    public void setRaio(int raio) {
73        this.raio = raio;
74    }
75⊕    public int getAngulo() {
76        return angulo;
77    }
78⊕    public void setAngulo(int angulo) {
79        this.angulo = angulo;
80    }
81⊕    public int getDistancia() {
82        return distancia;
83    }
84⊕    public void setDistancia(int distancia) {
85        this.distancia = distancia;
86    }
87⊕    public void bloquearThread() {
88        try {
89            activo.acquire();
90        } catch (InterruptedException e) {
91            System.err.println("BDDancarino bloquearThread error - Semaphore activo not working: " + e.getMessage());
92        }
93    }
94⊕    public void desbloquearThread() {
95        activo.release();
96    }
97⊕    public void terminarThread() {
98        fim = true;
99    }
100⊕   public void terminarSpyThread() {
101        robotLego.terminarThread();
102    }
103 }

```

6.2. 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 import java.awt.event.WindowAdapter;
15 import java.awt.event.WindowEvent;
16
17 public class GUIDancarino extends JFrame {
18
19     private static final long serialVersionUID = 1L;
20     private JPanel contentPane;
21     private JTextField textFieldRobot, textFieldRaio, textFieldAngulo, textFieldDistancia;
22     private JLabel lblRobot, lblRaio, lblAngulo, lblDistancia, lblConsola;
23     private JRadioButton rdbtnOnOff;
24     private JButton btnFrente, btnParar, btnEsquerda, btnDireita, btnRetaguarda;
25     private JCheckBox chckbxDebug;
26     private JTextArea textAreaConsola;
27     private JScrollPane scrollPane;
28     private JCheckBox chckbxActivar;
29
30     private final static String newline = "\n";
31
32     private BDDancarino bd;
33     private SpyRobot rb;
34
35     public void run() { }
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@ /**
47 * Create the frame.
48 */
49@ public GUIDancarino(String title, BDDancarino b) {
50     setTitle(title);
51     bd = b;
52     rb = bd.getRobotLego();
53
54@     addWindowListener(new WindowAdapter() {
55@         @Override
56@         public void windowClosing(WindowEvent arg0) {
57@             bd.terminarThread();
58@             rb.CloseEV3();
59@             System.err.println("Closed connection with robot");
60@         }
61@     });
62 //     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
63 //     setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

```

64         setBounds(100, 100, 450, 487);
65         contentPane = new JPanel();
66         contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
67
68         setContentPane(contentPane);
69         contentPane.setLayout(null);
70
71         lblRobot = new JLabel("Robot");
72         lblRobot.setBounds(77, 11, 39, 14);
73         contentPane.add(lblRobot);
74
75         textFieldRobot = new JTextField();
76         textFieldRobot.addActionListener(new ActionListener() {
77             public void actionPerformed(ActionEvent arg0) {
78
79                 bd.setNomeRobot(textFieldRobot.getText());
80                 myPrint("Nome robot: " + bd.getNomeRobot());
81             }
82         });
83         textFieldRobot.setText(bd.getNomeRobot());
84         textFieldRobot.setBounds(126, 8, 186, 20);
85         contentPane.add(textFieldRobot);
86         textFieldRobot.setColumns(10);
87
88         rdbtnOnOff = new JRadioButton("On/Off");
89         rdbtnOnOff.addActionListener(new ActionListener() {
90             public void actionPerformed(ActionEvent e) {
91
92                 if (rdbtnOnOff.isSelected()) {
93
94                     if (rb.OpenEV3(bd.getNomeRobot())) {
95                         bd.setOnOff(rdbtnOnOff.isSelected());
96                     }
97                     else {
98                         rdbtnOnOff.setSelected(false);
99                     }
100                }
101                else {
102                    bd.setOnOff(rdbtnOnOff.isSelected());
103                    rb.CloseEV3();
104                }
105                activateButtons(bd.isOnOff());
106                myPrint("OnOff: " + bd.isOnOff());
107            }
108        });
109        rdbtnOnOff.setSelected(bd.isOnOff());
110        rdbtnOnOff.setBounds(333, 7, 70, 23);
111        contentPane.add(rdbtnOnOff);
112
113         lblRaio = new JLabel("Raio");
114         lblRaio.setBounds(10, 39, 46, 14);
115         contentPane.add(lblRaio);
116
117         textFieldRaio = new JTextField();
118         textFieldRaio.addActionListener(new ActionListener() {
119             public void actionPerformed(ActionEvent e) {
120
121                 try {
122                     bd.setRaio(Integer.parseInt(textFieldRaio.getText()));
123                 } catch (NumberFormatException nfe) {
124                     textFieldRaio.setText("" + bd.getRaio());
125                     myPrint("Raio : Exception Error!");
126                 }
127                 myPrint("Raio: " + bd.getRaio());
128             }
129         });
130         textFieldRaio.setText("" + bd.getRaio());
131         textFieldRaio.setBounds(42, 36, 46, 20);
132         contentPane.add(textFieldRaio);
133         textFieldRaio.setColumns(10);
134

```

```

135     lblAngulo = new JLabel("Angulo");
136     lblAngulo.setBounds(136, 39, 39, 14);
137     contentPane.add(lblAngulo);
138
139     textFieldAngulo = new JTextField();
140     textFieldAngulo.addActionListener(new ActionListener() {
141         public void actionPerformed(ActionEvent e) {
142
143             try {
144                 bd.setAngulo(Integer.parseInt(textFieldAngulo.getText()));
145             } catch (NumberFormatException nfe) {
146                 textFieldAngulo.setText("") + bd.getAngulo());
147                 myPrint("Angulo : Exception Error!");
148             }
149             myPrint("Angulo: " + bd.getAngulo());
150         }
151     });
152     textFieldAngulo.setText("") + bd.getAngulo());
153     textFieldAngulo.setBounds(185, 39, 46, 20);
154     contentPane.add(textFieldAngulo);
155     textFieldAngulo.setColumns(10);
156
157     lblDistancia = new JLabel("Distancia");
158     lblDistancia.setBounds(301, 39, 46, 14);
159     contentPane.add(lblDistancia);
160
161     textFieldDistancia = new JTextField();
162     textFieldDistancia.addActionListener(new ActionListener() {
163         public void actionPerformed(ActionEvent e) {
164
165             try {
166                 bd.setDistancia(Integer.parseInt(textFieldDistancia.getText()));
167             } catch (NumberFormatException nfe) {
168                 textFieldDistancia.setText("") + bd.getDistancia());
169                 myPrint("Distancia : Exception Error!");
170             }
171             myPrint("Distancia: " + bd.getDistancia());
172         }
173     });
174     textFieldDistancia.setText("") + bd.getDistancia());
175     textFieldDistancia.setBounds(357, 37, 46, 20);
176     contentPane.add(textFieldDistancia);
177     textFieldDistancia.setColumns(10);
178
179     btnFrente = new JButton("Frente");
180     btnFrente.addActionListener(new ActionListener() {
181         public void actionPerformed(ActionEvent e) {
182
183             rb.Reta(bd.getDistancia());
184             rb.Parar(false);
185             myPrint("Reta: " + bd.getDistancia());
186
187         }
188     });
189     btnFrente.setBackground(Color.GREEN);
190     btnFrente.setBounds(172, 70, 101, 44);
191     contentPane.add(btnFrente);
192
193     btnParar = new JButton("Parar");
194     btnParar.addActionListener(new ActionListener() {
195         public void actionPerformed(ActionEvent e) {
196
197             rb.Parar(true);
198             myPrint("Parar");
199         }
200     });
201     btnParar.setBackground(Color.RED);
202     btnParar.setBounds(172, 125, 101, 44);
203     contentPane.add(btnParar);

```

```

205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269

```

```

btnDireita = new JButton("Direita");
btnDireita.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        rb.CurvarDireita(bd.getRaio(), bd.getAngulo());
        rb.Parar(false);
        myPrint("Direita: raio " + bd.getRaio() + " angulo " + bd.getAngulo());
    }
});
btnDireita.setBackground(Color.MAGENTA);
btnDireita.setBounds(283, 125, 101, 44);
contentPane.add(btnDireita);

btnEsquerda = new JButton("Esquerda");
btnEsquerda.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        rb.CurvarEsquerda(bd.getRaio(), bd.getAngulo());
        rb.Parar(false);
        myPrint("Esquerda: raio " + bd.getRaio() + " angulo " + bd.getAngulo());
    }
});
btnEsquerda.setBackground(Color.ORANGE);
btnEsquerda.setBounds(60, 125, 101, 44);
contentPane.add(btnEsquerda);

btnRetaguarda = new JButton("Retaguarda");
btnRetaguarda.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        rb.Reta(-bd.getDistancia());
        rb.Parar(false);
        myPrint("Reta: " + -bd.getDistancia());
    }
});
btnRetaguarda.setBackground(Color.BLUE);
btnRetaguarda.setBounds(172, 180, 101, 44);
contentPane.add(btnRetaguarda);

chkbxDebug = new JCheckBox("Debug");
chkbxDebug.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        bd.setDebug(chkbxDebug.isSelected());
        myPrint("Debug: " + bd.isDebugEnabled());
    }
});
chkbxDebug.setSelected(bd.isDebugEnabled());
chkbxDebug.setBounds(10, 212, 61, 23);
contentPane.add(chkbxDebug);

lblConsola = new JLabel("Consola");
lblConsola.setBounds(10, 242, 46, 14);
contentPane.add(lblConsola);

scrollPane = new JScrollPane();
scrollPane.setBounds(10, 257, 414, 180);
contentPane.add(scrollPane);

textAreaConsola = new JTextArea();
textAreaConsola.setText(bd.getConsola());
scrollPane.setViewportView(textAreaConsola);
textAreaConsola.setLineWrap(true);
textAreaConsola.setEditable(false);

```

```

270         chckbxActivar = new JCheckBox("Activar");
271         chckbxActivar.addActionListener(new ActionListener() {
272             public void actionPerformed(ActionEvent e) {
273
274                 if (!bd.isOnOff())
275                     chckbxActivar.setSelected(false);
276                 bd.setActivar(chckbxActivar.isSelected());
277                 myPrint("Dançarino Activo - activar: " + bd.isActivar());
278                 if (chckbxActivar.isSelected()) {
279                     bd.desbloquearThread();
280                 }
281             }
282         });
283         chckbxActivar.setBounds(354, 212, 70, 23);
284         contentPane.add(chckbxActivar);
285
286         JButton btnClear = new JButton("Clear");
287         btnClear.addActionListener(new ActionListener() {
288             public void actionPerformed(ActionEvent arg0) {
289
290                 bd.setConsola(" ");
291                 textAreaConsola.setText(bd.getConsola());
292             }
293         });
294         btnClear.setBounds(77, 212, 70, 23);
295         contentPane.add(btnClear);
296
297         activateButtons(false);
298
299         setVisible(true);
300     }
301
302
303     public void myPrint(String text) {
304         if (bd.isDebugEnabled()) {
305             bd.setConsola(text);
306             textAreaConsola.append(bd.getConsola() + newline);
307             textAreaConsola.setCaretPosition(textAreaConsola.getDocument().getLength());
308         }
309     }
310
311     protected void activateButtons(boolean onOff) {
312         btnFrente.setEnabled(onOff);
313         btnEsquerda.setEnabled(onOff);
314         btnDireita.setEnabled(onOff);
315         btnRetaguarda.setEnabled(onOff);
316         btnParar.setEnabled(onOff);
317     }
318 }

```


6.3. Anexo C – Classe “Dancarino”

```
1 import java.io.IOException;
2
3 public class Dancarino implements Runnable {
4
5     private int estado;
6     private final int PARAR_FALSE = 0;
7     private final int FRENTE = 1;
8     private final int CURVAR_DIREITA = 2;
9     private final int CURVAR_ESQUERDA = 3;
10    private final int RETAGUARDA = 4;
11    private final int PARAR_TRUE = 5;
12    private final int ESPERAR_TERMINAR_RETA = 6;
13    private final int ESPERAR_TERMINAR_CURVA = 7;
14    private final int ESPERAR_ACTIVAR = 8;
15    private final int LER_COMMANDO = 9;
16    private final int TERMINAR = 10;
17
18    private final int distancia = 10;
19    private final int raio = 0;
20    private final int angulo = 45;
21    private final int tReta10 = 700; // Tempo de reta e retaguarda
22    private final int tCurvar045 = 200; // Tempo de curvar Esquerda e curva Direita
23    private long timestamp;
24
25    private String title;
26    private BDDancarino bd;
27    private CanalComunicacao com;
28    private GUIDancarino gui;
29    private int ID;
30    private SpyRobot robot;
31    private Evitar evitar;
32    private BDEvitar bdEvitar;
33    private Thread e;
34
35    private MyMessage msg;
36
37⊕  public Dancarino(String nomeProcesso, CanalComunicacao com) {
38        estado = ESPERAR_ACTIVAR;
39
40        msg = new MyMessage(0, 0);
41
42        title = nomeProcesso;
43        this.com = com;
44        bd = new BDDancarino();
45        gui = new GUIDancarino(title, bd);
46        robot = bd.getRobotLego();
47
48        ID = com.OpenLeitor();
49
50        bdEvitar = new BDEvitar();
51        evitar = new Evitar(bdEvitar, robot);
52        e = new Thread(evitar);
53    }
54
55⊕  public void run() {
56
57    private void automatoDancarino() {
58
59        switch (estado) {
60            case ESPERAR_ACTIVAR:
61
62                bd.bloquearThread();
63                e.start();
64
65                if (bdEvitar.isFim() || bd.isFim()) {
66                    bdEvitar.terminarThread();
67                    bd.terminarSpyThread();
68                    estado = TERMINAR;
69                }
70
71            if (estado == ESPERAR_ACTIVAR) {
72                estado = LER_COMMANDO;
73            }
74
75            break;
76
77
78    }
79
80    }
81
82    }
83
84    }
85
86    }
87
88    }
89
90    }
91
92    }
93
94    }
95
96    }
97
98    }
99
100   }
101
102   }
103
104   }
105
106   }
107
108   }
109
110   }
111
112   }
113
114   }
115
116   }
117
118   }
119
120   }
121
122   }
123
124   }
125
126   }
127
128   }
129
130   }
131
132   }
133
134   }
135
136   }
137
138   }
139
140   }
141
142   }
143
144   }
145
146   }
147
148   }
149
150   }
151
152   }
153
154   }
155
156   }
157
158   }
159
160   }
161
162   }
163
164   }
165
166   }
167
168   }
169
170   }
171
172   }
173
174   }
175
176   }
177
178   }
179
180   }
181
182   }
183
184   }
185
186   }
187
188   }
189
190   }
191
192   }
193
194   }
195
196   }
197
198   }
199
200   }
201
202   }
203
204   }
205
206   }
207
208   }
209
210   }
211
212   }
213
214   }
215
216   }
217
218   }
219
220   }
221
222   }
223
224   }
225
226   }
227
228   }
229
230   }
231
232   }
233
234   }
235
236   }
237
238   }
239
240   }
241
242   }
243
244   }
245
246   }
247
248   }
249
250   }
251
252   }
253
254   }
255
256   }
257
258   }
259
260   }
261
262   }
263
264   }
265
266   }
267
268   }
269
270   }
271
272   }
273
274   }
275
276   }
277
278   }
279
280   }
281
282   }
283
284   }
285
286   }
287
288   }
289
290   }
291
292   }
293
294   }
295
296   }
297
298   }
299
299 }
```

```

80     case LER_COMMANDO:
81         try {
82             msg = com.ler(ID);
83
84             if (estado == LER_COMMANDO)
85                 estado = msg.getOrdem();
86
87         } catch (IOException e) {
88             System.err.println("Dancarino automatoDancarino() error - validate ID not working: " + e.getMessage());
89         }
90
91         if (!bd.isActivar())
92             estado = ESPERAR_ACTIVAR;
93
94         if (bdEvitar.isFim() || bd.isFim()) {
95             bdEvitar.terminarThread();
96             bd.terminarSpyThread();
97             estado = TERMINAR;
98         }
99
100        break;
101
102    case PARAR_FALSE:
103        gui.myPrint("[ " + msg.getNumero() + ", " + msg.getOrdem() + "] -> Parar: false");
104
105        robot.Parar(false);
106
107        if (estado == PARAR_FALSE)
108            estado = LER_COMMANDO;
109        break;
110
111    case FRENTE:
112        bd.setDistancia(distancia);
113
114        gui.myPrint("[ " + msg.getNumero() + ", " + msg.getOrdem() + "] -> Reta: " + bd.getDistancia());
115
116        timestamp = System.currentTimeMillis();
117
118        robot.Reta(bd.getDistancia());
119
120        if (estado == FRENTE)
121            estado = ESPERAR_TERMINAR_RETNA;
122        break;
123
124    case CURVAR_DIREITA:
125        bd.setRaio(raio);
126        bd.setAngulo(angulo);
127
128        gui.myPrint("[ " + msg.getNumero() + ", " + msg.getOrdem() + "] -> Direita: raio "
129                    + bd.getRaio() + " angulo " + bd.getAngulo());
130
131        timestamp = System.currentTimeMillis();
132        robot.CurvarDireita(bd.getRaio(), bd.getAngulo());
133
134        if (estado == CURVAR_DIREITA)
135            estado = ESPERAR_TERMINAR_CURVA;
136        break;
137
138    case CURVAR_ESQUERDA:
139        bd.setRaio(raio);
140        bd.setAngulo(angulo);
141
142        gui.myPrint("[ " + msg.getNumero() + ", " + msg.getOrdem() + "] -> Esquerda: raio "
143                    + bd.getRaio() + " angulo " + bd.getAngulo());
144
145        timestamp = System.currentTimeMillis();
146        robot.CurvarEsquerda(bd.getRaio(), bd.getAngulo());
147
148        if (estado == CURVAR_ESQUERDA)
149            estado = ESPERAR_TERMINAR_CURVA;
150        break;

```

```

152     case RETAGUARDA:
153         bd.setDistancia(distancia);
154
155         gui.myPrint("[" + msg.getNumero() + ", " + msg.getOrdem() + "] -> Reta: " + -bd.getDistancia());
156
157         timestamp = System.currentTimeMillis();
158         robot.Reta(-1*bd.getDistancia());
159
160         if (estado == RETAGUARDA)
161             estado = ESPERAR_TERMINAR_RETA;
162         break;
163
164     case PARAR_TRUE:
165         gui.myPrint("[" + msg.getNumero() + ", " + msg.getOrdem() + "] -> Parar: true");
166
167         robot.Parar(true);
168
169         if (estado == PARAR_TRUE)
170             estado = LER_COMMANDO;
171         break;
172
173     case ESPERAR_TERMINAR_RETA:
174         actividadeEsperar(100);
175         if (System.currentTimeMillis() - timestamp > tReta10) {
176             if (estado == ESPERAR_TERMINAR_RETA) {
177                 estado = LER_COMMANDO;
178             }
179         }
180         break;
181
182     case ESPERAR_TERMINAR_CURVA:
183         actividadeEsperar(100);
184         if (System.currentTimeMillis() - timestamp > tCurvar045) {
185             if (estado == ESPERAR_TERMINAR_CURVA) {
186                 estado = LER_COMMANDO;
187             }
188         }
189         break;
190     case TERMINAR:
191         break;
192
193     default:
194         System.err.println("Erro no Dançarino: Automato Dançarino - Estado: " + estado);
195         break;
196     }
197 }
198
199
200 private void actividadeEsperar(long tempo) {
201     try {
202         Thread.sleep(tempo);
203     } catch (InterruptedException e) {
204         System.err.println(
205             "Dançarino actividadeEsperar(tempo) error - Thread.sleep(tempo) not working: " + e.getMessage());
206     }
207 }
208 public int getId() {
209     return ID;
210 }
211 public String getNomeProcesso() {
212     return title;
213 }
214 public SpyRobot getRobotLego() {
215     return robot;
216 }
217 public void ativarDancer(boolean ativar) {
218     bd.setAtivar(ativar);
219 }
220 public void activateButtons(boolean coreoAtivos) {
221     gui.activateButtons(coreoAtivos);
222 }
223 public String getNomeRobot() {
224     return bd.getNomeRobot();
225 }
226 public void setOnOff(boolean b) {
227     bd.setOnOff(b);
228 }
229 public void desbloquearThread() {
230     bd.desbloquearThread();
231 }
232 }

```


6.4. Anexo D – Classe “BDCoreografo”

```
1 import java.util.concurrent.Semaphore;
2
3 public class BDCoreografo {
4     private boolean activar, comando1, comandos16, comandos32,
5         comandosIlimitados, pararComandos;
6     private String consola;
7     private Semaphore activo;
8     private boolean fim;
9
10    public BDCoreografo() {
11        comando1 = false;
12        comandos16 = false;
13        comandos32 = false;
14        comandosIlimitados = false;
15        pararComandos = false;
16        consola = new String("");
17
18        activar = false;
19        activo = new Semaphore(0);
20        fim = false;
21    }
22
23    public boolean isFim() {
24        return fim;
25    }
26    public boolean isActivar() {
27        return activar;
28    }
29    public void setActivar(boolean activar) {
30        this.activar = activar;
31    }
32    public boolean isComando1() {
33        return comando1;
34    }
35    public void setComando1(boolean comando1) {
36        this.comando1 = comando1;
37    }
38    public boolean isComandos16() {
39        return comandos16;
40    }
41    public void setComandos16(boolean comandos16) {
42        this.comandos16 = comandos16;
43    }
44    public boolean isComandos32() {
45        return comandos32;
46    }
47    public void setComandos32(boolean comandos32) {
48        this.comandos32 = comandos32;
49    }
50    public boolean isComandosIlimitados() {
51        return comandosIlimitados;
52    }
53    public void setComandosIlimitados(boolean comandosIlimitados) {
54        this.comandosIlimitados = comandosIlimitados;
55    }
56    public boolean isPararComandos() {
57        return pararComandos;
58    }
59    public void setPararComandos(boolean pararComandos) {
60        this.pararComandos = pararComandos;
61    }
62    public String getConsola() {
63        return consola;
64    }
65    public void setConsola(String consola) {
66        this.consola = consola;
67    }
```

```
68⊕ public void bloquearThread() {
69     try {
70         activo.acquire();
71     } catch (InterruptedException e) {
72         System.err.println("BDCoreografo bloquearThread() error - Semaphore activo not working: " + e.getMessage());
73     }
74 }
75⊕ public void desbloquearThread() {
76     activo.release();
77 }
78⊕ public void terminarThread() {
79     fim = true;
80 }
81 }
```

6.5. Anexo E – Classe “GUICoreografo”

```
1⑩ 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 import java.awt.event.WindowAdapter;
12 import java.awt.event.WindowEvent;
13
14 public class GUICoreografo extends JFrame {
15
16
17     private static final long serialVersionUID = 1L;
18
19     private JPanel contentPane;
20
21     private JLabel lblUltimosComandos;
22     private JTextArea textArea;
23     private JCheckBox chckbxActivar;
24⑩     private JButton btnGerar1Comando, btnGerar16Comandos, btnGerar32Comandos,
25     btnGerarComandosIlimitados, btnPararComandos;
26     private JScrollPane scrollPane;
27
28     private final static String newline = "\n";
29
30     private BDCoreografo bd;
31
32⑩     /**
33      * Launch the application.
34      */
35⑩ /* public static void main(String[] args) {
36     BDCoreografo GUI = new GUICoreografo();
37     GUI.run();
38 }
39 */
40     public void run() {}
41
42⑩     /**
43      * Create the frame.
44      */
45⑩     public GUICoreografo(String title, BDCoreografo b) {
46         setTitle(title);
47         bd = b;
48
49⑩         addWindowListener(new WindowAdapter() {
50⑩             @Override
51             public void windowClosing(WindowEvent arg0) {
52                 bd.terminarThread();
53             }
54         });
55
56 //        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
57 //        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
58         setBounds(100, 100, 523, 300);
59         contentPane = new JPanel();
60         contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
61         setContentPane(contentPane);
62         contentPane.setLayout(null);
63
64         lblUltimosComandos = new JLabel("Últimos 10 Comandos");
65         lblUltimosComandos.setBounds(77, 11, 129, 14);
66         contentPane.add(lblUltimosComandos);
67
68         scrollPane = new JScrollPane();
69         scrollPane.setBounds(10, 36, 277, 214);
70         contentPane.add(scrollPane);
71
72         textArea = new JTextArea();
73         scrollPane.setViewportView(textArea);
```

```

74     textArea.setLineWrap(true);
75     textArea.setEditable(false);
76
77     chckbxActivar = new JCheckBox("Activar");
78     chckbxActivar.addActionListener(new ActionListener() {
79         public void actionPerformed(ActionEvent arg0) {
80
81             bd.setActivar(chckbxActivar.isSelected());
82             activateButtons(chckbxActivar.isSelected());
83             myPrint("Coreografo Activo - activar: " + bd.isActivar());
84
85             if (chckbxActivar.isSelected()) {
86                 bd.desbloquearThread();
87             }
88         }
89     });
90     chckbxActivar.setBounds(368, 7, 101, 23);
91     contentPane.add(chckbxActivar);
92
93     btnGerar1Comando = new JButton("Gerar 1 comando");
94     btnGerar1Comando.addActionListener(new ActionListener() {
95         public void actionPerformed(ActionEvent e) {
96
97             bd.setComando1(true);
98         }
99     });
100    btnGerar1Comando.setBounds(297, 48, 188, 29);
101    contentPane.add(btnGerar1Comando);
102
103    btnGerar16Comandos = new JButton("Gerar 16 comandos");
104    btnGerar16Comandos.addActionListener(new ActionListener() {
105        public void actionPerformed(ActionEvent e) {
106
107            bd.setComandos16(true);
108        }
109    });
110    btnGerar16Comandos.setBounds(297, 88, 188, 29);
111    contentPane.add(btnGerar16Comandos);
112
113    btnGerar32Comandos = new JButton("Gerar 32 comandos");
114    btnGerar32Comandos.addActionListener(new ActionListener() {
115        public void actionPerformed(ActionEvent e) {
116
117            bd.setComandos32(true);
118        }
119    });
120    btnGerar32Comandos.setBounds(297, 128, 188, 29);
121    contentPane.add(btnGerar32Comandos);
122
123    btnGerarComandosIlimitados = new JButton("Gerar comandos ilimitados");
124    btnGerarComandosIlimitados.addActionListener(new ActionListener() {
125        public void actionPerformed(ActionEvent e) {
126
127            bd.setComandosIlimitados(true);
128        }
129    });
130    btnGerarComandosIlimitados.setBounds(297, 168, 188, 29);
131    contentPane.add(btnGerarComandosIlimitados);
132
133    btnPararComandos = new JButton("Parar comandos");
134    btnPararComandos.addActionListener(new ActionListener() {
135        public void actionPerformed(ActionEvent e) {
136
137            bd.setPararComandos(true);
138        }
139    });
140    btnPararComandos.setBounds(297, 209, 188, 30);
141    contentPane.add(btnPararComandos);
142

```

```
143 JButton btnClear = new JButton("Clear");
144 btnClear.addActionListener(new ActionListener() {
145     public void actionPerformed(ActionEvent arg0) {
146
147         bd.setConsola(" ");
148         textArea.setText(bd.getConsola());
149     }
150 });
151 btnClear.setBounds(216, 7, 71, 23);
152 contentPane.add(btnClear);
153
154 activateButtons(false);
155
156 setVisible(true);
157 }
158
159 public void myPrint(String text) {
160     bd.setConsola(text);
161     textArea.append(bd.getConsola() + newline);
162     textArea.setCaretPosition(textArea.getDocument().getLength());
163 }
164
165 public void activateButtons(boolean bool) {
166     btnGerar1Comando.setEnabled(bool);
167     btnGerar16Comandos.setEnabled(bool);
168     btnGerar32Comandos.setEnabled(bool);
169     btnGerarComandosIlimitados.setEnabled(bool);
170     btnPararComandos.setEnabled(bool);
171 }
172 }
173 }
```


6.6. Anexo F – Classe “Coreografo”

```
1 import java.io.IOException;
2
3 public class Coreografo implements Runnable {
4
5     private int estado;
6     private final int ESPERAR_ACTIVAR = 0;
7     private final int ESPERAR_COMMANDO = 1;
8     private final int ENVIAR_1_COMMANDO = 2;
9     private final int ENVIAR_16_COMMANDOS = 3;
10    private final int ENVIAR_32_COMMANDOS = 4;
11    private final int ENVIAR_COMMANDOS_ILIMITADOS = 5;
12    private final int PARAR_COMMANDOS = 6;
13    private final int TERMINAR = 7;
14
15    private int contador, numComandos, numero;
16
17    private CanalComunicacao com;
18    private String title;
19    private BDCoreografo bd;
20    private GUICoreografo gui;
21    private String comando;
22
23    private int ID;
24
25    public Coreografo(String nomeProcesso, CanalComunicacao com) {
26        estado = ESPERAR_ACTIVAR;
27        contador = 0;
28        numComandos = 0;
29        numero = 0;
30
31        title = nomeProcesso;
32        this.com = com;
33        bd = new BDCoreografo();
34        gui = new GUICoreografo(title, bd);
35
36        ID = com.OpenEscritor();
37    }
38
39    public void run() {
40        while (estado != TERMINAR) {
41            automatoCoreografo();
42        }
43    }
44
45    private void automatoCoreografo() {
46        switch (estado) {
47            case ESPERAR_ACTIVAR:
48
49                bd.bloquearThread();
50
51                if (bd.isFim()) {
52                    estado = TERMINAR;
53                }
54                if (estado == ESPERAR_ACTIVAR)
55                    estado = ESPERAR_COMMANDO;
56
57                break;
58
59            case ESPERAR_COMMANDO:
60
61                actividadeEsperar(100);
62
63                numComandos = 1;
64
65                if (bd.isCommando1()) {
66                    if (estado == ESPERAR_COMMANDO)
67                        estado = ENVIAR_1_COMMANDO;
68
69                } else if (bd.isComandos16()) {
70                    if (estado == ESPERAR_COMMANDO)
71                        estado = ENVIAR_16_COMMANDOS;
72
73                } else if (bd.isComandos32()) {
74                    if (estado == ESPERAR_COMMANDO)
75                        estado = ENVIAR_32_COMMANDOS;
76
77            }
78
79        }
80    }
81
82    private void actividadeEsperar(int tempo) {
83        try {
84            Thread.sleep(tempo);
85        } catch (InterruptedException e) {
86            e.printStackTrace();
87        }
88    }
89
90    private void enviarComando() {
91        if (numero < numComandos) {
92            comando = bd.getCommando();
93            numero++;
94
95            if (comando != null) {
96                com.enviarComando(comando);
97            }
98
99        }
100    }
101
102    private void pararComando() {
103        if (numero >= numComandos) {
104            numero = 0;
105            bd.parar();
106        }
107    }
108
109    private void enviarComandos() {
110        if (numero < numComandos) {
111            enviarComando();
112        } else {
113            pararComando();
114        }
115    }
116
117    private void enviarComandosILimitados() {
118        if (numero < numComandos) {
119            enviarComando();
120        } else {
121            pararComando();
122        }
123    }
124
125    private void enviarComandos16() {
126        if (numero < numComandos) {
127            enviarComando();
128        } else {
129            pararComando();
130        }
131    }
132
133    private void enviarComandos32() {
134        if (numero < numComandos) {
135            enviarComando();
136        } else {
137            pararComando();
138        }
139    }
140
141    private void enviarComando1() {
142        if (numero < numComandos) {
143            enviarComando();
144        } else {
145            pararComando();
146        }
147    }
148
149    private void terminar() {
150        if (numero < numComandos) {
151            enviarComando();
152        } else {
153            pararComando();
154        }
155    }
156
157    private void enviarComandoFinal() {
158        if (numero < numComandos) {
159            enviarComando();
160        } else {
161            pararComando();
162        }
163    }
164
165    private void enviarComandoFinal16() {
166        if (numero < numComandos) {
167            enviarComando();
168        } else {
169            pararComando();
170        }
171    }
172
173    private void enviarComandoFinal32() {
174        if (numero < numComandos) {
175            enviarComando();
176        } else {
177            pararComando();
178        }
179    }
180
181    private void enviarComandoFinalILimitados() {
182        if (numero < numComandos) {
183            enviarComando();
184        } else {
185            pararComando();
186        }
187    }
188
189    private void enviarComandoFinal1() {
190        if (numero < numComandos) {
191            enviarComando();
192        } else {
193            pararComando();
194        }
195    }
196
197    private void enviarComandoFinalFinal() {
198        if (numero < numComandos) {
199            enviarComando();
200        } else {
201            pararComando();
202        }
203    }
204
205    private void enviarComandoFinalFinal16() {
206        if (numero < numComandos) {
207            enviarComando();
208        } else {
209            pararComando();
210        }
211    }
212
213    private void enviarComandoFinalFinal32() {
214        if (numero < numComandos) {
215            enviarComando();
216        } else {
217            pararComando();
218        }
219    }
220
221    private void enviarComandoFinalFinalILimitados() {
222        if (numero < numComandos) {
223            enviarComando();
224        } else {
225            pararComando();
226        }
227    }
228
229    private void enviarComandoFinalFinal1() {
230        if (numero < numComandos) {
231            enviarComando();
232        } else {
233            pararComando();
234        }
235    }
236
237    private void enviarComandoFinalFinalFinal() {
238        if (numero < numComandos) {
239            enviarComando();
240        } else {
241            pararComando();
242        }
243    }
244
245    private void enviarComandoFinalFinalFinal16() {
246        if (numero < numComandos) {
247            enviarComando();
248        } else {
249            pararComando();
250        }
251    }
252
253    private void enviarComandoFinalFinalFinal32() {
254        if (numero < numComandos) {
255            enviarComando();
256        } else {
257            pararComando();
258        }
259    }
260
261    private void enviarComandoFinalFinalFinalILimitados() {
262        if (numero < numComandos) {
263            enviarComando();
264        } else {
265            pararComando();
266        }
267    }
268
269    private void enviarComandoFinalFinalFinal1() {
270        if (numero < numComandos) {
271            enviarComando();
272        } else {
273            pararComando();
274        }
275    }
276
277    private void enviarComandoFinalFinalFinalFinal() {
278        if (numero < numComandos) {
279            enviarComando();
280        } else {
281            pararComando();
282        }
283    }
284
285    private void enviarComandoFinalFinalFinalFinal16() {
286        if (numero < numComandos) {
287            enviarComando();
288        } else {
289            pararComando();
290        }
291    }
292
293    private void enviarComandoFinalFinalFinalFinal32() {
294        if (numero < numComandos) {
295            enviarComando();
296        } else {
297            pararComando();
298        }
299    }
299 --
```

```

77
78         } else if (bd.isComandosIlimitados()) {
79             if (estado == ESPERAR_COMANDO)
80                 estado = ENVIAR_COMANDOS_ILIMITADOS;
81
82         } else if (bd.isPararComandos())
83             if (estado == ESPERAR_COMANDO)
84                 estado = PARAR_COMANDOS;
85
86         if (!bd.isActivar())
87             estado = ESPERAR_ACTIVAR;
88         if (bd.isFim())
89             estado = TERMINAR;
90         break;
91
92     case ENVIAR_1_COMMANDO:
93         enviar1Comando();
94         bd.setComando1(false);
95         enviarComandoParar();
96         if (estado == ENVIAR_1_COMMANDO)
97             estado = ESPERAR_COMANDO;
98
99         if (!bd.isActivar())
100            estado = ESPERAR_ACTIVAR;
101        if (bd.isFim())
102            estado = TERMINAR;
103        break;
104
105    case ENVIAR_16_COMMANDOS:
106        enviar1Comando();
107
108        if (++contador == 16) {
109            bd.setComandos16(false);
110            contador = 0;
111            enviarComandoParar();
112
113            if (estado == ENVIAR_16_COMMANDOS)
114                estado = ESPERAR_COMANDO;
115        }
116
117        if (bd.isPararComandos())
118            if (estado == ENVIAR_16_COMMANDOS)
119                estado = PARAR_COMANDOS;
120
121        if (!bd.isActivar()) {
122            enviarComandoParar();
123            estado = ESPERAR_ACTIVAR;
124        }
125        if (bd.isFim())
126            estado = TERMINAR;
127        break;
128
129    case ENVIAR_32_COMMANDOS:
130        enviar1Comando();
131
132        if (++contador == 32) {
133            bd.setComandos32(false);
134            contador = 0;
135            enviarComandoParar();
136
137            if (estado == ENVIAR_32_COMMANDOS)
138                estado = ESPERAR_COMANDO;
139        }
140        if (bd.isPararComandos())
141            if (estado == ENVIAR_32_COMMANDOS)
142                estado = PARAR_COMANDOS;
143
144        if (!bd.isActivar()) {
145            enviarComandoParar();
146            estado = ESPERAR_ACTIVAR;
147        }
148        if (bd.isFim())
149            estado = TERMINAR;
150        break;

```

```

151     case ENVIAR_COMMANDOS_ILLIMITADOS:
152         enviar1Comando();
153
154         if (bd.isPararComandos())
155             if (estado == ENVIAR_COMMANDOS_ILLIMITADOS)
156                 estado = PARAR_COMMANDOS;
157
158         if (!bd.isActivar()) {
159             enviarComandoParar();
160             estado = ESPERAR_ACTIVAR;
161         }
162         if (bd.isFim())
163             estado = TERMINAR;
164         break;
165
166     case PARAR_COMMANDOS:
167         inactivarComandos();
168         enviarComandoParar();
169
170         if (estado == PARAR_COMMANDOS)
171             estado = ESPERAR_COMMANDO;
172
173         if (!bd.isActivar())
174             estado = ESPERAR_ACTIVAR;
175
176         if (bd.isFim())
177             estado = TERMINAR;
178         break;
179
180     case TERMINAR:
181         break;
182
183     default:
184         System.err.println("Erro no Coreografo: automatoCoreografo - estado: " + estado);
185         break;
186     }
187 }
188
189 private void actividadeEsperar(long tempo) {
190     try {
191         Thread.sleep(tempo);
192     } catch (InterruptedException e) {
193         System.err.println("Coreografo actividadeEsperar(long tempo) error - Thread.sleep(tempo) not working: "
194             + e.getMessage());
195     }
196 }
197
198
199 private void enviar1Comando() {
200
201     int ordem = (int) (Math.random() * ((4 - 1) + 1)) + 1;
202
203     MyMessage msg = new MyMessage(++numero, ordem);
204     try {
205         com.escrever(msg, ID);
206     } catch (IOException e) {
207         System.err.println("Coreografo enviar1Comando() error - validate ID not working: " + e.getMessage());
208     }
209
210     comando = Ordem.findOrdem(ordem).toString();
211     gui.myPrint("Mensagem " + numComandos + ": [ " + msg.getNumero() + ", " + msg.getOrdem() + "] -> " + comando);
212     numComandos++;
213     actividadeEsperar(500);
214 }
215
216 private void enviarComandoParar() {
217     MyMessage msg = new MyMessage(++numero, Ordem.PARAR_FALSE.getValor());
218     try {
219         com.escrever(msg, ID);
220     } catch (IOException e) {
221         System.err.println("Coreografo enviarComandoParar() error - validate ID not working: " + e.getMessage());
222     }
223     gui.myPrint("Mensagem " + numComandos + ": [ " + msg.getNumero() + ", " + msg.getOrdem()
224         + "] -> Parar: false (PARAR)");
225
226     actividadeEsperar(500);
227 }
```

```
228     private void inactivarComandos() {
229         bd.setComando1(false);
230         bd.setComandos16(false);
231         bd.setComandos32(false);
232         bd.setComandosIlimitados(false);
233         bd.setPararComandos(false);
234     }
235
236     public int getId() {
237         return ID;
238     }
239
240     public String getNomeProcesso() {
241         return title;
242     }
243
244     public void ativarCoreo(boolean ativar) {
245         bd.setActivar(ativar);
246     }
247
248     public void activateButtons(boolean coreoAtivos) {
249         gui.activateButtons(coreoAtivos);
250     }
251
252     public void desbloquearThread() {
253         bd.desbloquearThread();
254     }
255
256 }
```

6.7. Anexo G – Classe “Ordem”

```
1 public enum Ordem {
2     PARAR_FALSE(0, "Parar: false"),
3     FRENTE(1, "Reta: 10"),
4     CURVAR_DIREITA(2, "Direita: 0 45"),
5     CURVAR_ESQUERDA(3, "Esquerda: 0 45"),
6     RETAGUARDA(4, "Reta: -10"),
7     PARAR_TRUE(5, "Parar: true");
8
9     private final int valor;
10    private final String nome;
11    Ordem(int valorOpcão, String nomeOrdem){
12        valor = valorOpcão;
13        nome = nomeOrdem;
14    }
15    public int getValor(){
16        return valor;
17    }
18    public String toString(){
19        return nome;
20    }
21    public static Ordem findOrdem(int v) {
22        for (Ordem e : Ordem.values()) {
23            if (e.getValor() == v) {
24                return e;
25            }
26        }
27        return null;
28    }
29 }
```


6.8. Anexo H – Classe “MyMessage”

```
1  public class MyMessage {  
2      private int numero;  
3      private int ordem;  
4  
5  
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 }
```


6.9. Anexo I – Classe “Descriptor”

```
1 import java.util.concurrent.Semaphore;
2
3 public class Descriptor {
4
5     private Semaphore ocupados;
6     private int getBuffer;
7     private int ID;
8
9     public Descriptor(int ID) {
10         ocupados = new Semaphore(0);
11         getBuffer = 0;
12         this.ID = ID;
13     }
14
15     public Semaphore getSemaforo() {
16         return ocupados;
17     }
18     public int getGetBufferPosition() {
19         return getBuffer;
20     }
21     public void setGetBufferPosition(int getBuffer) {
22         this.getBuffer = getBuffer;
23     }
24     public int getID() {
25         return ID;
26     }
27 }
```


6.10. Anexo J – Classe “BDCanal”

```
1  public class BDCanal {  
2      private String consola;  
3      private boolean fim;  
4  
5      public BDCanal() {  
6          consola = new String("");  
7          fim = false;  
8      }  
9  
10     public String getConsola() {  
11         return consola;  
12     }  
13     public void setConsola(String consola) {  
14         this.consola = consola;  
15     }  
16     public boolean isFim() {  
17         return fim;  
18     }  
19     public void terminarThread() {  
20         fim = true;  
21     }  
22 }  
23 }  
24 }
```


6.11. Anexo K – Classe “CanalComunicacao”

```
1④ import java.io.IOException;
2 import java.nio.ByteBuffer;
3 import java.util.ArrayList;
4 import java.util.concurrent.Semaphore;
5
6 public class CanalComunicacao implements Runnable {
7
8     // dimensão máxima em bytes do buffer
9     final int BUFFER_MAX = 256; // 32 mensagens de 2 ints com 4 bytes por cada int - 32*8 = 256
10    // private final int BUFFER_MAX = 32; // 4 mensagens para teste
11
12    private final int DIVISOR = 8;
13    private final int NUMERO_MSG = (BUFFER_MAX / DIVISOR);
14
15    private ByteBuffer buffer;
16
17    private int putBuffer, number, order;
18
19    private Semaphore exclusaoMutua, bloquearCanal, livres;
20
21    private int estado;
22    private final int BLOQUEAR_CANAL = 0;
23    private final int LER = 1;
24    private final int ESCREVER = 2;
25    private final int TERMINAR = 3;
26
27
28    private int ID;
29    private ArrayList<Descriptor> leitores;
30    private ArrayList<Integer> escritores;
31    private ArrayList<Integer> clientesleitores;
32    private ArrayList<Integer> clienteescritores;
33
34    private BDCanal bd;
35
36④ public CanalComunicacao() {
37        estado = BLOQUEAR_CANAL;
38        buffer = ByteBuffer.allocate(BUFFER_MAX);
39
40        putBuffer = 0;
41        number = 0;
42        order = 0;
43
44        clearBuffer();
45
46        // Semafóros
47        exclusaoMutua = new Semaphore(1);
48        bloquearCanal = new Semaphore(0);
49        livres = new Semaphore(NUMERO_MSG);
50
51        leitores = new ArrayList<Descriptor>();
52        escritores = new ArrayList<Integer>();
53        clientesleitores = new ArrayList<Integer>();
54        clienteescritores = new ArrayList<Integer>();
55
56        ID = 0;
57
58        bd = new BDCanal();
59        // gui = new GUICanal(bd);
60    }
61
62
63④     public void run() {
64         while (estado != TERMINAR) {
65             automatoCanal();
66         }
67     }
68
69
```

```

70
71     private void automatoCanal() {
72         switch (estado) {
73             case BLOQUEAR_CANAL:
74                 if(bd.isFim()) {
75                     estado = TERMINAR;
76                 }
77             try {
78                 // gui.printChannel("");
79                 gui.printChannel("-> BLOQUEAR_CANAL");
80                 bloquearCanal.acquire();
81
82                 if(!clientesescritores.isEmpty()) {
83                     if(estado == BLOQUEAR_CANAL)
84                         estado = ESCREVER;
85                 }
86                 if(!clientesleitores.isEmpty()) {
87                     if(estado == BLOQUEAR_CANAL)
88                         estado = LER;
89                 }
90             } catch (InterruptedException e) {
91                 System.err.println("CanalComunicacao automatoCanal() error - Semaphore bloquearCanal not working: "
92                             + e.getMessage());
93             }
94             break;
95
96         case LER:
97             // gui.printChannel("-> LER_AUX -> ");
98             // gui.printChannel("actualizar livres ");
99             // gui.printChannel("livres antes: " + livres.availablePermits());
100            actualizarLivres();
101            // gui.printChannel("livres depois: " + livres.availablePermits());
102
103            clientesleitores.remove(0);
104
105            if (estado == LER)
106                estado = BLOQUEAR_CANAL;
107            break;
108
109        case ESCREVER:
110            // gui.printChannel("-> ESCREVER_AUX -> ");
111            // gui.printChannel("Semaforos dos leitores: ");
112            for(Descriptor d : leitores) {
113                d.getSemaphore().release();
114                gui.printChannel("- leitor " + d.getID() + ", n_atual: " + d.getSemaphore().availablePermits());
115            }
116            clientesescritores.remove(0);
117
118            if (estado == ESCREVER)
119                estado = BLOQUEAR_CANAL;
120            break;
121
122        case TERMINAR:
123            break;
124
125        default:
126            System.err.println("Erro CanalComunicacao - automatoCanal() - estado " + estado);
127        }
128    }
129
130

```

```

131@    public void escrever(MyMessage msgEscritor, int id) throws IOException {
132 //     gui.printChannel("");
133 //     gui.printChannel("escritor " + id + " quer escrever " + msgEscritor.toString() + " ! --- " + Thread.currentThread());
134
135     if(!idEscritorAutorizado(id)) {
136         throw new IOException("INVALID ID");
137     }
138
139     try {
140 //         gui.printChannel("livres will be acquired");
141 //         gui.printChannel("-> livres before: " + livres.availablePermits());
142 //         livres.acquire();
143 //         gui.printChannel("-> livres after: " + livres.availablePermits());
144
145         escreverMensagem(msgEscritor);
146
147         clientesEscritores.add(id);
148         bloquearCanal.release();
149
150     } catch (InterruptedException e1) {
151         System.err.println(
152             "CanalComunicacao escrever() error - Semaphore livres not working: "
153             + e1.getMessage());
154     }
155
156 }
157
158
159@    public MyMessage ler(int id) throws IOException {
160 //     gui.printChannel("");
161 //     gui.printChannel("leitor " + id + " quer ler ! --- " + Thread.currentThread());
162
163     MyMessage msgLeitor = new MyMessage(0,0);
164
165     if(!idLeitorAutorizado(id)) {
166         throw new IOException("INVALID ID");
167     }
168
169     public MyMessage ler(int id) throws IOException {
170 //         gui.printChannel("");
171 //         gui.printChannel("leitor " + id + " quer ler ! --- " + Thread.currentThread());
172
173     MyMessage msgLeitor = new MyMessage(0,0);
174
175     if(!idLeitorAutorizado(id)) {
176         throw new IOException("INVALID ID");
177     }
178
179     Descritor leitor = getLeitor(id);
180     Semaphore semaforoLeitor = leitor.getSemaphore();
181
182     try {
183 //         gui.printChannel("semaforoLeitor will be acquired");
184 //         gui.printChannel("-> leitor before: " + semaforoLeitor.availablePermits());
185 //         semaforoLeitor.acquire();
186 //         gui.printChannel("-> leitor after: " + semaforoLeitor.availablePermits());
187
188         msgLeitor = lerMensagem(leitor);
189
190         clientesLeitores.add(id);
191         bloquearCanal.release();
192
193     } catch (InterruptedException e) {
194         System.err.println(
195             "CanalComunicacao lerMensagem() error - Semaphore semaforoLeitor not working: "
196             + e.getMessage());
197     }
198
199 //     gui.printChannel("!!!! Mensagem lida pelo leitor " + leitor.getID() + ":" + msgLeitor + "!!!!");
200     return msgLeitor;
201 }
202

```

```

194⊕ private void escreverMensagem(MyMessage mensagem) {
195     try {
196         exclusaoMutua.acquire();
197
198         buffer.position(putBuffer);
199         buffer.putInt(mensagem.getNumero());
200         buffer.putInt(mensagem.getOrdem());
201
202         putBuffer = (putBuffer += 8) % BUFFER_MAX;
203
204         exclusaoMutua.release();
205     } catch (InterruptedException e) {
206         System.err.println("CanalComunicacao escreverMensagem() error - Semaphore exclusaoMutua not working: "
207                             + e.getMessage());
208     }
209 }
210
211
212
213⊕ private MyMessage lerMensagem(Descriptor leitor) {
214     try {
215         exclusaoMutua.acquire();
216
217         int getBuffer = leitor.getGetBufferPosition();
218
219         number = buffer.getInt(getBuffer);
220         order = buffer.getInt(getBuffer + 4);
221
222         getBuffer = (getBuffer += 8) % BUFFER_MAX;
223
224         leitor.setGetBufferPosition(getBuffer);
225
226         exclusaoMutua.release();
227
228     } catch (InterruptedException e) {
229         System.err.println("CanalComunicacao lerMensagem() error - Semaphore exclusaoMutua not working: "
230                             + e.getMessage());
231     }
232     return new MyMessage(number, order);
233 }
234
235
236⊕ private boolean idLeitorAutorizado(int id) {
237     boolean autorizado = false;
238     for (Descriptor d : leitores) {
239         if (d.getID() == id) {
240             autorizado = true;
241         }
242     }
243     return autorizado;
244 }
245
246
247⊕ private boolean idEscritorAutorizado(int id) {
248     boolean autorizado = false;
249     for (int i : escritores) {
250         if (i == id) {
251             autorizado = true;
252         }
253     }
254     return autorizado;
255 }
256
257
258⊕ private Descriptor getLeitor(int id) {
259     for (Descriptor d : leitores) {
260         if (d.getID() == id) {
261             return d;
262         }
263     }
264     return null;
265 }

```

```

268⊕    private void actualizarLivres() {
269        if (livres.availablePermits() < NUMERO_MSG) {
270            for (int i = 0; i < leitores.size(); i++) {
271                Descriptor d = leitores.get(i);
272                if ((NUMERO_MSG - livres.availablePermits()) == d.getSemaforo().availablePermits()) {
273                    break;
274                }
275                if (i == leitores.size() - 1) {
276                    livres.release();
277                }
278            }
279        }
280    }
281
282
283⊕    public int OpenLeitor() {
284        Descriptor leitor = new Descriptor(++ID);
285        leitores.add(leitor);
286        System.out.println("ID leitor: " + ID);
287        return ID;
288    }
289
290
291⊕    public int OpenEscritor() {
292        escritores.add(++ID);
293        System.out.println("ID escritor: " + ID);
294        return ID;
295    }
296
297⊕    public void Close(int ID) {
298        for (Descriptor d : leitores)
299            if (d.getID() == ID)
300                leitores.remove(d);
301        for (int id : escritores)
302            if (id == ID)
303                escritores.remove(new Integer(id));
304    }
305
306
307⊕    private void clearBuffer() {
308        for (int i = 0; i < BUFFER_MAX; i++) {
309            buffer.position(i);
310            buffer.put((byte) 0);
311        }
312    }
313 }

```


6.12. Anexo L – Classe “BDGUIT2”

```
1 public class BDGUIT2 {  
2     private String consolaLog;  
3  
4     private boolean dancerAtivos, coreoAtivos;  
5  
6     public BDGUIT2() {  
7         consolaLog = new String("");  
8     }  
9  
10    public String getConsolaLog() {  
11        return consolaLog;  
12    }  
13    public void setConsolaLog(String consolaLog) {  
14        this.consolaLog = consolaLog;  
15    }  
16  
17    public void setDancerAtivos(boolean selected) {  
18        dancerAtivos = selected;  
19    }  
20    public boolean isDancerAtivos() {  
21        return dancerAtivos;  
22    }  
23  
24    public void setCoreoAtivos(boolean selected) {  
25        coreoAtivos = selected;  
26    }  
27    public boolean isCoreoAtivos() {  
28        return coreoAtivos;  
29    }  
30  
31 }  
32 }
```


6.13. Anexo M – Classe “GUITP2”

```
1⑩ import javax.swing.JFrame;
2 import javax.swing.JScrollPane;
3
4 import javax.swing.JButton;
5 import javax.swing.JLabel;
6 import javax.swing.JTextArea;
7
8 import java.awt.event.ActionListener;
9 import java.util.ArrayList;
10 import java.util.Enumeration;
11 import java.awt.event.ActionEvent;
12 import javax.swing.JCheckBox;
13 import javax.swing.JList;
14 import javax.swing.DefaultListModel;
15
16 public class GUITP2 extends JFrame {
17
18     private static final long serialVersionUID = 1L;
19
20     private JLabel lblDancarinos, lblCoregrafos;
21⑩     private JButton btnLimpaLogging, btnIniciarTodosDancarino, btnRemoverDancarino, btnAdicionarDancarino,
22         btnIniciarTodosCoreografo, btnRemoverCoreografo, btnAdicionarCoreografo;
23     private JCheckBox chckbxActivarTodosDancarino, chckbxActivarTodosCoreografo;
24     private JTextArea textAreaLog;
25     private JScrollPane scrollCoreografo, scrollDancarino, scrollLog;
26
27     private DefaultListModel<String> modelC, modelD;
28     private JList<String> listCoreografo, listDancarino;
29     private ArrayList<Coreografo> processosCoreo;
30     private ArrayList<Dancarino> processosDancer;
31     private int contadorCoreografos, contadorDancarinos, numeroC, numeroD;
32     Enumeration<String> nomesProcessos;
33     private ArrayList<String> processosCoreoAtivos;
34     private ArrayList<String> processosDancerAtivos;
35
36     private final static String newline = "\n";
37     private CanalComunicacao com;
38     private BDGUIT2 bd;
39
40     public void run() { }
41
42
43⑩ /**
44 * Launch the application.
45 */
46⑩ public static void main(String[] args) {
47     GUITP2 gui = new GUITP2();
48     gui.run();
49 }
50
51⑩ /**
52 * Create the application.
53 */
54⑩ public GUITP2() {
55     bd = new BDGUIT2();
56
57     com = new CanalComunicacao();
58     Thread cc = new Thread(com);
59     cc.start();
60
61     initialize();
62
63     printLog("(" + cc.getName() + ")", CanalComunicacao);
64 }
65
66⑩ /**
67 * Initialize the contents of the frame.
68 */
```

```

69⊕ private void initialize() {
70
71     processosCoreo = new ArrayList<Coreografo>();
72     processosDancer = new ArrayList<Dancarino>();
73
74     processosCoreoAtivos = new ArrayList<String>();
75     processosDancerAtivos = new ArrayList<String>();
76
77     contadorCoreografos = 1;
78     contadorDancarinos = 1;
79     numeroC = 1;
80     numeroD = 1;
81
82     setTitle("GUI T2");
83     setBounds(100, 100, 579, 633);
84     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
85     getContentPane().setLayout(null);
86
87     lblCoreografos = new JLabel("Core\u00f3grafos");
88     lblCoreografos.setBounds(45, 5, 92, 55);
89     getContentPane().add(lblCoreografos);
90
91     lblDancarinos = new JLabel("Dan\u00e7arinos");
92     lblDancarinos.setBounds(298, 18, 70, 28);
93     getContentPane().add(lblDancarinos);
94
95
96     // TODO Jlist's Coreografo e Dancarino
97
98     modelC = new DefaultListModel<>();
99     listCoreografo = new JList<>(modelC);
100    listCoreografo.setLayoutOrientation(JList.VERTICAL);
101
102    scrollCoreografo = new JScrollPane(listCoreografo);
103    scrollCoreografo.setBounds(45, 50, 221, 91);
104    getContentPane().add(scrollCoreografo);
105
106    modelD = new DefaultListModel<>();
107    listDancarino = new JList<>(modelD);
108    listDancarino.setBounds(342, 50, 177, 91);
109    listDancarino.setLayoutOrientation(JList.VERTICAL);
110
111    scrollDancarino = new JScrollPane(listDancarino);
112    scrollDancarino.setBounds(298, 50, 225, 91);
113    getContentPane().add(scrollDancarino);
114
115
116     // TODO JButton's Adicionar e Remover do Coreografo e Dancarino
117
118     btnAdicionarCoreografo = new JButton("Adicionar");
119     btnAdicionarCoreografo.addActionListener(new ActionListener() {
120         public void actionPerformed(ActionEvent e) {
121
122             modelC = (DefaultListModel<String>) listCoreografo.getModel();
123             modelC.addElement("Coreógrafo " + contadorCoreografos);
124             contadorCoreografos++;
125             listCoreografo.ensureIndexIsVisible(modelC.size() - 1);
126         }
127     });
128     btnAdicionarCoreografo.setBounds(45, 152, 221, 46);
129     getContentPane().add(btnAdicionarCoreografo);
130
131     btnRemoverCoreografo = new JButton("Remover");
132     btnRemoverCoreografo.addActionListener(new ActionListener() {
133         public void actionPerformed(ActionEvent arg0) {
134
135             int index = listCoreografo.getSelectedIndex();
136             if (index != -1) {
137                 modelC.removeElementAt(index);
138             }
139
140         }
141     });

```

```

142     btnRemoverCoreografo.setBounds(45, 209, 221, 46);
143     getContentPane().add(btnRemoverCoreografo);
144
145
146     btnAdicionarDancarino = new JButton("Adicionar");
147     btnAdicionarDancarino.addActionListener(new ActionListener() {
148         public void actionPerformed(ActionEvent e) {
149
150             modelD = (DefaultListModel<String>) listDancarino.getModel();
151             modelD.addElement("Dançarino " + contadorDancarinos);
152             contadorDancarinos++;
153             listDancarino.ensureIndexIsVisible(modelD.size() - 1);
154         }
155     });
156     btnAdicionarDancarino.setBounds(298, 152, 225, 46);
157     getContentPane().add(btnAdicionarDancarino);
158
159     btnRemoverDancarino = new JButton("Remover");
160     btnRemoverDancarino.addActionListener(new ActionListener() {
161         public void actionPerformed(ActionEvent e) {
162
163             int index = listDancarino.getSelectedIndex();
164             if (index != -1) {
165                 modelD.removeElementAt(index);
166             }
167         }
168     });
169     btnRemoverDancarino.setBounds(298, 209, 225, 46);
170     getContentPane().add(btnRemoverDancarino);
171
172     // TODO JButton's IniciarTodos do Coreografo e Dancarino
173
174     btnIniciarTodosCoreografo = new JButton("Iniciar Todos");
175     btnIniciarTodosCoreografo.addActionListener(new ActionListener() {
176         public void actionPerformed(ActionEvent e) {
177
178             //get coreo model
179             nomesProcessos = modelC.elements();
180
181             //for each string
182             clearArrayCoreografos();
183             while (nomesProcessos.hasMoreElements()) {
184
185                 String nomeProcesso = nomesProcessos.nextElement();
186
187                 if(addCoreografo(nomeProcesso)) {
188                     processosCoreo.add(new Coreografo(nomeProcesso, com));
189                 }
190             }
191
192             //Inicia todas os processos coreografos
193             for(Coreografo c : processosCoreo) {
194                 Thread threadC = new Thread(c);
195                 threadC.start();
196
197                 threadC.setName("(" + threadC.getName() + "), Coreógrafo " + numeroC + ", ID no canal: " + c.getId());
198                 numeroC++;
199                 printLog(threadC.getName());
200                 processosCoreoAtivos.add(c.getNomeProcesso());
201             }
202
203             if(processosCoreo.size() > 0)
204                 chckbxActivarTodosCoreografo.setEnabled(true);
205             else
206                 chckbxActivarTodosCoreografo.setEnabled(false);
207         }
208     });
209     btnIniciarTodosCoreografo.setBounds(45, 266, 221, 46);
210     getContentPane().add(btnIniciarTodosCoreografo);

```

```

213
214    btnIniciarTodosDancarino = new JButton("Iniciar Todos");
215    btnIniciarTodosDancarino.addActionListener(new ActionListener() {
216        public void actionPerformed(ActionEvent e) {
217            //get dancer model
218            Enumeration<String> nomesProcessos = modelD.elements();
219
220            //for each string
221            clearArrayDancarinos();
222            while (nomesProcessos.hasMoreElements()) {
223
224                String nomeProcesso = nomesProcessos.nextElement();
225
226                if(addDancarino(nomeProcesso)) {
227                    processosDancer.add(new Dancarino(nomeProcesso, com));
228                }
229            }
230
231            //Inicia todas os processos dancarino
232            for(Dancarino d : processosDancer) {
233                Thread threadD = new Thread(d);
234                threadD.start();
235
236                threadD.setName("(" + threadD.getName() + "), Dançarino " + numeroD + ", ID no canal: " + d.getId());
237                numeroD++;
238                printLog(threadD.getName());
239
240                processosDancerAtivos.add(d.getNomeProcesso());
241            }
242
243            if(processosDancer.size() > 0)
244                chckbxActivarTodosDancarino.setEnabled(true);
245            else
246                chckbxActivarTodosDancarino.setEnabled(false);
247        }
248    });
249    btnIniciarTodosDancarino.setBounds(298, 266, 225, 46);
250    getContentPane().add(btnIniciarTodosDancarino);
251
252
253 // TODO CheckBox's activar Todos do Coreografo e do Dancarino
254
255 chckbxActivarTodosCoreografo = new JCheckBox("Activar Todos");
256 chckbxActivarTodosCoreografo.addActionListener(new ActionListener() {
257     public void actionPerformed(ActionEvent e) {
258
259         printLog("");
260
261         bd.setCoreoAtivos(chckbxActivarTodosCoreografo.isSelected());
262
263         String adjetivo = new String("inactivo");
264         if(chckbxActivarTodosCoreografo.isSelected())
265             adjetivo = new String("activo");
266
267         for(Coreografo c : processosCoreo) {
268             c.ativarCoreo(bd.isCoreoAtivos());
269             c.activateButtons(bd.isCoreoAtivos());
270             printLog("Coreografo " + c.getId() + " " + adjetivo);
271             if (chckbxActivarTodosCoreografo.isSelected()) {
272                 c.desbloquearThread();
273             }
274         }
275
276     }
277 });
278 chckbxActivarTodosCoreografo.setBounds(45, 319, 113, 25);
279 getContentPane().add(chckbxActivarTodosCoreografo);
280

```

```

281     chckbxActivarTodosDancarino = new JCheckBox("Activar Todos");
282     chckbxActivarTodosDancarino.addActionListener(new ActionListener() {
△283     public void actionPerformed(ActionEvent e) {
284
285         printLog("");
286
287         for(Dancarino d : processosDancer) {
288             SpyRobot rb = d.getRobotLego();
289             RobotLegoEV3 rb = d.getRobotLego();
290
291             if(chckbxActivarTodosDancarino.isSelected()) {
292                 if (rb.OpenEV3(d.getNomeRobot())) {
293                     d.setOnOff(true);
294                 }
295                 else {
296                     chckbxActivarTodosDancarino.setSelected(false);
297                 }
298             }
299             else {
300                 d.setOnOff(false);
301                 rb.CloseEV3();
302             }
303
304             bd.setDancerAtivos(chckbxActivarTodosDancarino.isSelected());
305
306             String adjetivo = new String("inactivo");
307             if(chckbxActivarTodosDancarino.isSelected())
308                 adjetivo = new String("ativo");
309
310             d.ativarDancer(bd.isDancerAtivos());
311             d.activateButtons(chckbxActivarTodosDancarino.isSelected());
312             printLog("Dancarino " + d.getId() + " " + adjetivo);
313
314             if (chckbxActivarTodosDancarino.isSelected()) {
315                 d.desbloquearThread();
316             }
317         }
318     }
319 });
320 chckbxActivarTodosDancarino.setBounds(298, 319, 113, 25);
321 getContentPane().add(chckbxActivarTodosDancarino);
322
323 // TODO JButton e TextArea do LOG
324
325
326 btnLimpaLogging = new JButton("Limpa Logging");
327 btnLimpaLogging.addActionListener(new ActionListener() {
△328     public void actionPerformed(ActionEvent e) {
329
330         bd.setConsolaLog(" ");
331         textAreaLog.setText(bd.getConsolaLog());
332
333     }
334 });
335 btnLimpaLogging.setBounds(45, 359, 478, 25);
336 getContentPane().add(btnLimpaLogging);
337
338 scrollLog = new JScrollPane();
339 scrollLog.setBounds(45, 395, 478, 187);
340 getContentPane().add(scrollLog);
341
342 textAreaLog = new JTextArea();
343 scrollLog.setViewportView(textAreaLog);
344 textAreaLog.setLineWrap(true);
345 textAreaLog.setEditable(false);
346
347 chckbxActivarTodosCoreografo.setEnabled(false);
348 chckbxActivarTodosDancarino.setEnabled(false);
349
350 setVisible(true);
351 }

```

```

353@    private void clearArrayDancarinos() {
354        int size = processosDancer.size();
355        for(int i = 0; i < size; i++) {
356            processosDancer.remove(0);
357        }
358    }
359@    private void clearArrayCoreografos() {
360        int size = processosCoreo.size();
361        for(int i = 0; i < size; i++) {
362            processosCoreo.remove(0);
363        }
364    }
365
366@    private boolean addDancarino(String nome) {
367        for (String str : processosDancerAtivos) {
368            if(str.equals(nome)) {
369                return false;
370            }
371        }
372        return true;
373    }
374@    private boolean addCoreografo(String nome) {
375        for (String str : processosCoreoAtivos) {
376            if(str.equals(nome)) {
377                return false;
378            }
379        }
380        return true;
381    }
382
383@    public void printLog(String text) {
384        bd.setConsolaLog(text);
385        textAreaLog.append(bd.getConsolaLog() + newline);
386        textAreaLog.setCaretPosition(textAreaLog.getDocument().getLength());
387    }
388 }

```

6.14. Anexo N – Classe “BDSpyRobot”

```
1 public class BDSpyRobot {
2
3     public enum estados {
4         ESPERAR(0),
5         TERMINAR(1),
6         REPRODUZIR(2),
7         GRAVAR(3),
8         PARAR(4),
9         ESCREVER(5);
10
11     private final int valor;
12     estados(int valorOpcão){
13         valor = valorOpcão;
14     }
15     public int getValor(){
16         return valor;
17     }
18 }
19
20 private String nomeFicheiro;
21 private String consola;
22 private int estado;
23 private boolean fim;
24
25 public BDSpyRobot() {
26     nomeFicheiro = new String("espio");
27     consola = new String("");
28     estado = estados.ESPERAR.getValor();
29     fim = false;
30 }
31 public boolean isFim() {
32     return fim;
33 }
34 public void setNomeFicheiro(String nomeFicheiro) {
35     this.nomeFicheiro = nomeFicheiro;
36 }
37 public String getNomeFicheiro() {
38     return nomeFicheiro;
39 }
40 public String getConsola() {
41     return consola;
42 }
43 public void setConsola(String consola) {
44     this.consola = consola;
45 }
46 public int getEstado() {
47     return estado;
48 }
49 public void setEstado(int estado) {
50     this.estado = estado;
51 }
52 public void iniciarGravacao() {
53     estado = estados.GRAVAR.getValor();
54 }
55 public void pararGravacao() {
56     estado = estados.PARAR.getValor();
57 }
58 public void iniciarReproducao() {
59     estado = estados.REPRODUZIR.getValor();
60 }
61 public void terminarThread() {
62     fim = true;
63 }
64 }
```


6.15. Anexo O – Classe “GUISpyRobot”

```
1① import javax.swing.JFrame;
2 import javax.swing.JButton;
3 import javax.swing.JTextArea;
4 import javax.swing.JLabel;
5 import javax.swing.JScrollPane;
6 import javax.swing.JTextField;
7 import java.awt.event.ActionListener;
8 import java.awt.event.ActionEvent;
9 import javax.swing.SwingConstants;
10 import java.awt.event.WindowAdapter;
11 import java.awt.event.WindowEvent;
12
13 public class GUISpyRobot extends JFrame {
14
15     private static final long serialVersionUID = 1L;
16     private static final String newline = "\n";
17
18     private JLabel lblGravarTrajetria, lblReproduzirTrajetria, lblNomeDoFicheiro;
19     private JTextField textField;
20     private JButton btnGravarParar, btnReproduzir;
21     private JTextArea textAreaConsola;
22     private JScrollPane scrollPane;
23
24     private BDSpyRobot bdSpy;
25
26② public GUISpyRobot(String title, BDSpyRobot bd) {
27     setTitle(title);
28     bdSpy = bd;
29     initialize();
30 }
31
32③ private void initialize() {
33
34④     addWindowListener(new WindowAdapter() {
35⑤         @Override
36             public void windowClosing(WindowEvent arg0) {
37                 bdSpy.terminarThread();
38             }
39         });
40
41     setBounds(100, 100, 515, 287);
42 //     setDefaultCloseOperation(EXIT_ON_CLOSE);
43     setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
44     getContentPane().setLayout(null);
45
46     lblGravarTrajetria = new JLabel("Gravar / Parar Trajet\u00f3ria");
47     lblGravarTrajetria.setHorizontalAlignment(SwingConstants.CENTER);
48     lblGravarTrajetria.setBounds(22, 56, 175, 14);
49     getContentPane().add(lblGravarTrajetria);
50
51     lblReproduzirTrajetria = new JLabel("Reproduzir Trajet\u00f3ria");
52     lblReproduzirTrajetria.setHorizontalAlignment(SwingConstants.CENTER);
53     lblReproduzirTrajetria.setBounds(22, 148, 175, 14);
54     getContentPane().add(lblReproduzirTrajetria);
55
56     lblNomeDoFicheiro = new JLabel("Nome do ficheiro");
57     lblNomeDoFicheiro.setBounds(219, 22, 111, 14);
58     getContentPane().add(lblNomeDoFicheiro);
59
60 }
```

```

61         btnGravarParar = new JButton("Gravar");
62         btnGravarParar.addActionListener(new ActionListener() {
63             public void actionPerformed(ActionEvent arg0) {
64                 if(btnGravarParar.getText().equals("Gravar")) {
65                     btnGravarParar.setText("Parar");
66                     bdSpy.iniciarGravacao();
67                     myPrint("Esta gravando ...");
68                 }
69                 else {
70                     btnGravarParar.setText("Gravar");
71                     bdSpy.pararGravacao();
72                     myPrint("Parou de gravar!");
73                 }
74             }
75         });
76         btnGravarParar.setBounds(22, 81, 175, 41);
77         getContentPane().add(btnGravarParar);
78
79         btnReproduzir = new JButton("Reproduzir");
80         btnReproduzir.addActionListener(new ActionListener() {
81             public void actionPerformed(ActionEvent arg0) {
82                 bdSpy.iniciarReproducao();
83                 myPrint("Reproduzindo ...");
84             }
85         });
86         btnReproduzir.setBounds(22, 173, 175, 41);
87         getContentPane().add(btnReproduzir);
88
89
90         textField = new JTextField();
91         textField.addActionListener(new ActionListener() {
92             public void actionPerformed(ActionEvent e) {
93
94                 bdSpy.setNomeFicheiro(textField.getText());
95                 myPrint("Nome de ficheiro: " + bdSpy.getNomeFicheiro());
96
97             }
98         });
99         textField.setText(bdSpy.getNomeFicheiro());
100        textField.setBounds(340, 19, 149, 20);
101        getContentPane().add(textField);
102        textField.setColumns(10);
103
104        scrollPane = new JScrollPane();
105        scrollPane.setBounds(219, 50, 270, 188);
106        getContentPane().add(scrollPane);
107
108        textAreaConsola = new JTextArea();
109        scrollPane.setViewportView(textAreaConsola);
110        textAreaConsola.setLineWrap(true);
111        textAreaConsola.setEditable(false);
112
113        setVisible(true);
114
115    }
116
117    public void myPrint(String text) {
118        bdSpy.setConsola(text);
119        textAreaConsola.append(bdSpy.getConsola() + newline);
120        textAreaConsola.setCaretPosition(textAreaConsola.getDocument().getLength());
121    }
122 }

```

6.16. Anexo P – Classe “SpyRobot”

```
1① import java.io.File;
2 import java.io.FileInputStream;
3 import java.io.FileOutputStream;
4 import java.io.IOException;
5 import java.io.InputStream;
6 import java.io.OutputStream;
7 import java.util.ArrayList;
8 import java.util.concurrent.Semaphore;
9
10 public class SpyRobot implements Runnable {
11
12     private int estado;
13     private final int ESPERAR = 0;
14     private final int TERMINAR = 1;
15     private final int REPRODUZIR = 2;
16     private final int GRAVAR = 3;
17     private final int PARAR = 4;
18     private final int ESCREVER = 5;
19
20     public final int S_1 = RobotLegoEV3.S_1;
21     public final int S_2 = RobotLegoEV3.S_2;
22     public final int S_3 = RobotLegoEV3.S_3;
23     public final int S_4 = RobotLegoEV3.S_4;
24
25     private BDSpyRobot bdSpy;
26     private GUISpyRobot guiSpy;
27     private MyRobotLego robot;
28 //    private RobotLegoEV3 robot;
29
30     // Tempo de reta e retaguarda sao iguais 720
31     private final int tReta10 = 700;
32     // Tempo de curvar Esquerda e curva Direita sao iguais 180
33     private final int tCurvar045 = 200;
34     private long timestamp;
35
36     private ArrayList<Integer> mensagensEspiadas;
37     private boolean espiar;
38
39     private Semaphore exclusaoMutua;
40
41② @Override
42     public void run() {
43         while (estado != TERMINAR) {
44             automatoSpy();
45         }
46     }
47
48③     public SpyRobot() {
49         estado = ESPERAR;
50         bdSpy = new BDSpyRobot();
51         guiSpy = new GUISpyRobot("SpyRobot", bdSpy);
52         robot = new MyRobotLego();
53 //        robot = new RobotLegoEV3();
54
55         timestamp = 0;
56
57         mensagensEspiadas = new ArrayList<Integer>();
58         espiar = false;
59
60         exclusaoMutua = new Semaphore(1);
61     }
62
63
64④     private void automatoSpy() {
65         switch(estado){
66             case ESPERAR:
67                 if(bdSpy.isFim()) {
68                     estado = TERMINAR;
69                 }
70                 actividadeEsperar(100);
71                 if (estado == ESPERAR)
72                     estado = bdSpy.getEstado();
73                 break;
74         }
```

```

75     case REPRODUZIR:
76         lerFicheiro();
77
78         if (estado == REPRODUZIR) {
79             estado = ESPERAR;
80             bdSpy.setEstado(estado);
81         }
82         break;
83
84     case ESCREVER:
85         escreverFicheiro();
86
87         if (estado == ESCREVER) {
88             estado = ESPERAR;
89             bdSpy.setEstado(estado);
90         }
91         break;
92
93     case GRAVAR:
94         escutarCanal();
95
96         if(estado == GRAVAR) {
97             estado = ESPERAR;
98             bdSpy.setEstado(estado);
99         }
100        break;
101
102    case PARAR:
103        pararEscutar();
104
105        if (estado == PARAR) {
106            estado = ESCREVER;
107            bdSpy.setEstado(estado);
108        }
109        break;
110
111    case TERMINAR:
112        break;
113    default:
114        System.err.println("Erro no SpyRobot: Automato - Estado: " + estado);
115        break;
116    }
117}
118
119
120@ private void escutarCanal() {
121
122    espiar = true;
123
124    mensagensEspiadas = new ArrayList<Integer>();
125
126
127}
128
129@ public boolean isEspiar() {
130    return espiar;
131}
132
133@ private void pararEscutar() {
134
135    espiar = false;
136
137    guiSpy.myPrint("");
138
139    for(Integer ordem: mensagensEspiadas) {
140        String str = Ordem.findOrdem(ordem).toString();
141        guiSpy.myPrint("Mensagem espiada: " + str);
142    }
143
144    guiSpy.myPrint("");
145
146}
147

```

```

148⊕    private void escreverFicheiro() {
149
150        OutputStream os = null;
151        File ficheiro = null;
152
153        try {
154
155            ficheiro = new File(bdSpy.getNomeFicheiro());
156            ficheiro.createNewFile();
157
158            os = new FileOutputStream(ficheiro);
159
160            // System.out.println("Ficheiro Válido? " + ficheiro.isFile()
161            // + "Permissões de escrita? " + ficheiro.canWrite()
162            // + "Dir? " + ficheiro.getAbsolutePath()
163            // + "Existe? " + ficheiro.exists());
164
165            for(Integer ordem : mensagensEspiadas) {
166
167                os.write(msg.getNumero());
168                os.write(msg.getOrdem());
169                os.write(ordem);
170            }
171
172        }catch(Exception e) {
173
174            e.printStackTrace();
175            guiSpy.myPrint("Não foi possível escrever no ficheiro fornecido");
176
177        } finally {
178
179            try {
180                os.close();
181
182            } catch (IOException e) {
183
184                e.printStackTrace();
185            }
186
187        }
188
189    }
190
191⊕    private void lerFicheiro(){
192        guiSpy.myPrint("");
193
194        String nomeFicheiro = null;
195        File ficheiro = null;
196        InputStream is = null;
197        // MyMessage msg = null;
198
199        try {
200            nomeFicheiro = bdSpy.getNomeFicheiro();
201            ficheiro = new File(nomeFicheiro);
202
203            is = new FileInputStream(ficheiro);
204
205            // int numero;
206            // int ordem;
207            // while((numero = is.read()) != -1) {
208            //     while((ordem = is.read()) != -1) {
209
210            //         ordem = is.read();
211
212            //         msg = new MyMessage(numero, ordem);
213            //         comandarRobot(ordem);
214            //         String str = Ordem.findOrdem(ordem).toString();
215            //         guiSpy.myPrint("Mensagem espiada: " + str);
216        }
217
218        }catch (IOException e) {
219            guiSpy.myPrint("Não foi possível ler o ficheiro: " + ficheiro.getPath() + e.getMessage());
...

```

```

221     }finally {
222         try {
223             is.close();
224         } catch (IOException e) {
225             e.printStackTrace();
226         }
227     }
228     guiSpy.myPrint("");
229 }
230
231 }
232
233⊕ private void atividadeEsperar(long tempo) {
234     try {
235         Thread.sleep(tempo);
236     } catch (InterruptedException e) {
237         System.err.println("Dançarino atividadeEsperar() error - Thread.sleep(tempo) not working: " + e.getMessage());
238     }
239 }
240
241⊕ private void comandarRobot(int ordem) {
242     switch(ordem) {
243         case 0:
244             Parar(false);
245             break;
246         case 1:
247             timestamp = System.currentTimeMillis();
248             Reta(10);
249             while (System.currentTimeMillis() - timestamp < tReta10);
250             break;
251         case 2:
252             timestamp = System.currentTimeMillis();
253             CurvarDireita(0, 45);
254             while (System.currentTimeMillis() - timestamp < tCurvar045);
255             break;
256         case 3:
257             timestamp = System.currentTimeMillis();
258             CurvarEsquerda(0, 45);
259             while (System.currentTimeMillis() - timestamp < tCurvar045);
260             break;
261         case 4:
262             timestamp = System.currentTimeMillis();
263             Reta(-10);
264             while (System.currentTimeMillis() - timestamp < tReta10);
265             break;
266         case 5:
267             Parar(true);
268             break;
269         default:
270             System.err.println("SpyRobot comandarRobot(msg) - Ordem inválida : " + ordem);
271     }
272 }
273
274⊕ public boolean OpenEV3(String nomeRobot) {
275     boolean abriu = false;
276     try {
277         exclusaoMutua.acquire();
278         abriu = robot.OpenEV3(nomeRobot);
279         exclusaoMutua.release();
280     } catch (InterruptedException e) {
281         e.printStackTrace();
282     }
283     return abriu;
284 }
285
286⊕ public void CloseEV3(){
287     try {
288         exclusaoMutua.acquire();
289         robot.CloseEV3();
290         exclusaoMutua.release();
291     } catch (InterruptedException e) {
292         e.printStackTrace();
293     }
294 }

```

```

295@ public void Parar(boolean b) {
296    try {
297        exclusaoMutua.acquire();
298        robot.Parar(b);
299        if (espiar) {
300            if (b) {
301                mensagensEspiadas.add(Ordem.PARAR_TRUE.getValor());
302            }
303            else {
304                mensagensEspiadas.add(Ordem.PARAR_FALSE.getValor());
305            }
306        }
307        exclusaoMutua.release();
308    } catch (InterruptedException e) {
309        e.printStackTrace();
310    }
311}
312@ public void Reta(int distancia) {
313    try {
314        exclusaoMutua.acquire();
315        robot.Reta(distancia);
316        if (espiar) {
317            if (distancia < 0) {
318                mensagensEspiadas.add(Ordem.RETAGUARDA.getValor());
319            }
320            else {
321                mensagensEspiadas.add(Ordem.FRENTE.getValor());
322            }
323        }
324        exclusaoMutua.release();
325    } catch (InterruptedException e) {
326        e.printStackTrace();
327    }
328}
329}
330@ public void CurvarDireita(int raio, int angulo) {
331    try {
332        exclusaoMutua.acquire();
333        robot.CurvarDireita(raio, angulo);
334        if (espiar) {
335            mensagensEspiadas.add(Ordem.CURVAR_DIREITA.getValor());
336        }
337        exclusaoMutua.release();
338    } catch (InterruptedException e) {
339        e.printStackTrace();
340    }
341}
342@ public void CurvarEsquerda(int raio, int angulo) {
343    try {
344        exclusaoMutua.acquire();
345        robot.CurvarEsquerda(raio, angulo);
346        if (espiar) {
347            mensagensEspiadas.add(Ordem.CURVAR_ESQUERDA.getValor());
348        }
349        exclusaoMutua.release();
350    } catch (InterruptedException e) {
351        e.printStackTrace();
352    }
353}
354@ public int SensorToque(int sensor) {
355    int sentiu = -1;
356    try {
357        exclusaoMutua.acquire();
358        sentiu = robot.SensorToque(sensor);
359        exclusaoMutua.release();
360    } catch (InterruptedException e) {
361        e.printStackTrace();
362    }
363    return sentiu;
364}
365
366@ public void terminarThread() {
367    bdSpy.terminarThread();
368}
369}

```


6.17. Anexo Q – Classe “BDEvitar”

```
2  public class BDEvitar {  
3  
4      private boolean fim;  
5  
6⊕      public BDEvitar() {  
7          fim = false;  
8      }  
9  
10⊕     public boolean isFim() {  
11         return fim;  
12     }  
13  
14⊕     public void terminarThread() {  
15         fim = true;  
16     }  
17  
18 }
```


6.18. Anexo R – Classe “Evitar”

```
1 public class Evitar implements Runnable {
2
3     private int estado;
4     private final int TESTAR_SENSOR = 0;
5     private final int TERMINAR = 1;
6
7     private BDEvitar bd;
8     private SpyRobot robot;
9
10    public Evitar(BDEvitar bd, SpyRobot robot) {
11        estado = TESTAR_SENSOR;
12        this.bd = bd;
13        this.robot = robot;
14    }
15
16    @Override
17    public void run() {
18        while (estado != TERMINAR) {
19            automatoEvitar();
20        }
21    }
22
23    private void automatoEvitar() {
24        switch (estado) {
25            case TESTAR_SENSOR:
26
27                actividadeEsperar(200);
28
29                if(robot.SensorToque(robot.S_1) != 0) {
30                    robot.Parar(true);
31                    System.err.println("Robot bateu a terminar");
32                    bd.terminarThread();
33                    estado = TERMINAR;
34                }
35                if (bd.isFim()) {
36                    estado = TERMINAR;
37                }
38                break;
39
40            case TERMINAR:
41                break;
42        }
43    }
44
45    private void actividadeEsperar(long tempo) {
46        try {
47            Thread.sleep(tempo);
48        } catch (InterruptedException e) {
49            System.err.println(
50                "Dançarino actividadeEsperar() error - Thread.sleep(tempo) not working: " + e.getMessage());
51        }
52    }
53}
54}
```


6.19. Anexo S – Classe “MyRobotLego”

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