

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA



**ÁREA DEPARTAMENTAL DE ENGENHARIA DE
ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES**

**Licenciatura em Engenharia
Informática e Multimédia**

Fundamentos de Sistemas Operativos

RELATÓRIO 2

2021SI

Docente: Eng. Carlos Gonçalves

Grupo 8

Aluno	Número
Maria Franco	46320
José Siopa	46338
Vitor Dinis	46327

30 de janeiro de 2021

ÍNDICE

ÍNDICE DE FIGURAS	III
1 OBJETIVOS.....	5
2 INTRODUÇÃO.....	7
3 DESENVOLVIMENTO.....	9
3.1 CLIENTE DO ROBOT	9
3.2 COMPORTAMENTOS	11
3.2.1 <i>Desenhar quadrado</i>	11
3.2.2 <i>Desenhar círculo</i>	12
3.2.3 <i>Espaçar formas geométricas</i>	13
3.3 BUFFER CIRCULAR	15
3.4 SERVIDOR	17
3.5 ROBOT DESENHADOR	19
3.6 ESTRUTURA, COMPONENTES E FUNCIONALIDADE DA GUI DA APLICAÇÃO	21
3.7 DIAGRAMA DE CLASSES (SEM A IMPLEMENTAÇÃO DO GRAVAR FORMAS).....	23
3.8 SINCRONIZAÇÃO E COMUNICAÇÃO ENTRE TAREFAS DESENHADORAS E O SERVIDOR.....	25
3.9 GRAVADOR DE FORMAS.....	27
3.9.1 <i>Estrutura da gravação</i>	27
3.9.2 <i>Reprodução no Robot</i>	27
3.10 SINCRONIZAÇÃO NO ACESSO AO ROBOT ENTRE TODAS AS TAREFAS	29
3.11 DIAGRAMA DE CLASSES (COM A IMPLEMENTAÇÃO DO GRAVAR FORMAS).....	31
4 CONCLUSÕES.....	33
5 BIBLIOGRAFIA	35
6 ANEXO I – CÓDIGO JAVA	37
6.1 MENSAGENS.....	37
6.1.1 <i>Classe Mensagem</i>	37
6.1.2 <i>Classe MsgCurvaDir</i>	38
6.1.3 <i>Classe MsgCurvaEsq</i>	39
6.1.4 <i>Classe MsgEspacar</i>	40
6.1.5 <i>Classe MsgParar</i>	41
6.1.6 <i>Classe MsgReta</i>	42
6.2 PRODUTOR	43
6.2.1 <i>Classe ClienteDoRobot</i>	43
6.2.2 <i>Enumerado Estado</i>	44
6.2.3 <i>Classe Comportamento</i>	45
6.2.4 <i>Classe DesenharCirculo</i>	47
6.2.5 <i>Classe DesenharQuadrado</i>	48
6.2.6 <i>Classe EspacarFormasGeometricas</i>	49
6.3 BUFFER	50

6.3.1	<i>Interface IBufferCircular</i>	50
6.3.2	<i>Interface IBufferCircularGUI</i>	51
6.3.3	<i>Classe BufferCircularBaseImpl</i>	52
6.3.4	<i>Classe BufferCircularMonitores</i>	53
6.3.5	<i>Classe BufferCircularGUI</i>	55
6.4	CONSUMIDOR.....	57
6.4.1	<i>Classe ServidorDoRobot</i>	57
6.4.2	<i>Classe GUIServidor</i>	58
6.4.3	<i>Classe RobotDesenhador</i>	60
6.5	GRAVADOR	61
6.5.1	<i>Enumerado EstadoGravador</i>	61
6.5.2	<i>Classe GravarFormas</i>	62
6.5.3	<i>Classe GUIGravarFormas</i>	65
6.6	GUI PRINCIPAL	68
6.6.1	<i>Interface IGUI</i>	68
6.6.2	<i>Classe GUILaunch</i>	69
6.6.3	<i>Classe GUI</i>	70

ÍNDICE DE FIGURAS

Figura 1 - Diagrama da classe <code>ClienteDoRobot</code>	9
Figura 2 - Diagrama da classe <code>Comportamento</code>	11
Figura 3 - Diagrama da classe <code>DesenharQuadrado</code>	12
Figura 4 - Diagrama da classe <code>DesenharCirculo</code>	13
Figura 5 - Diagrama da classe <code>EspacarFormasGeometricas</code>	13
Figura 6 - Diagrama das classes que compõem o <code>BufferCircular</code>	15
Figura 7 - GUI do <code>BufferCircular</code>	16
Figura 9 - Diagrama da classe <code>ServidorDoRobot</code>	17
Figura 8 - GUI do servidor	18
Figura 10 - Diagrama da classe <code>RobotDesenhador</code>	19
Figura 11 - GUI do Robot Desenhador.....	20
Figura 12 - GUI principal da aplicação	21
Figura 13 - Diagrama da classe <code>GUI</code>	22
Figura 14 - Confirmação de fecho da aplicação	22
Figura 15 – Diagrama de todas as classes sem o Gravar Formas	23
Figura 16 - Gravar Formas após escrita no ficheiro	27
Figura 17 - Gravar Formas no início do momento de reprodução.....	28
Figura 18 – Diagrama das classes que constituem o processo de gravar formas	28
Figura 19 – Diagrama de todas as classes simplificado.....	31
Figura 20 – Diagrama de todas as classes com o Gravar Formas.....	32

1 Objetivos

Este trabalho prático tem como principais objetivos o desenvolvimento de uma aplicação multitarefa, escrita em linguagem Java, que permite o controlo automático de um robot desenhador no espaço bidimensional. Esta deve ter em atenção a comunicação e sincronização entre tarefas, seguindo o modelo Produtor-Consumidor.

Com a utilização das classes `InputStream` e `OutputStream` da linguagem Java, será possível a manipulação de ficheiros, numa aplicação multiprocesso.

Finalmente, pretende-se a utilização do robot como sistema alvo para validar o trabalho desenvolvido.

Tomar-se-ão os seguintes passos:

- Criação de comportamentos independentes que comandam o robot;
- Criação de uma classe (produtor) que seja agregada pelos comportamentos;
- Definição dos comandos, produzidos pelos comportamentos, que irão ser utilizados para comunicar com uma tarefa (consumidor) através de um *buffer*;
- Criação de um *buffer* circular com capacidade máxima de 16 comandos;
- Criação de uma classe que implemente métodos públicos semelhantes aos do Robot EV3;
- GUIs para visualização e interação com possibilidade de geração de comandos;
- Desenvolvimento de uma *thread*, composta por GUI, que grave e reproduza comandos, armazenados num determinado ficheiro.

2 Introdução

De forma a aprofundar e demonstrar a aquisição dos conhecimentos lecionados na cadeira de Fundamentos de Sistemas Operativos, relativos ao funcionamento de aplicações multitarefa e manipulação de ficheiros, foi-nos proposto que desenvolvêssemos uma aplicação que permitisse utilizar um robot, para desenhar formas geométricas. Este, na sua forma física, será o robot EV3, ao qual deveremos conseguir enviar comandos, os quais descritos no Anexo 2 das Folhas de FSO [1].

Uma tarefa é parte de um processo. Em Java, as tarefas são objetos que derivam da classe *Thread*, que se executam dentro de um processo-pai ou aplicação.

O modelo Produtor-Consumidor, neste trabalho, será aplicado na forma de um cliente, que é agregado pelos comportamentos, desempenhando o papel de produtor. O consumidor irá ser desempenhado por um servidor. O canal de comunicação partilhado entre estes será um *buffer* circular. Um *buffer* diz-se circular quando o apontador, estando na última posição, ao ser incrementado, passa novamente à primeira posição.

A manipulação de ficheiros será feita por uma tarefa, que irá guardar os comandos enviados pelo produtor num ficheiro e, após lê-lo e reproduzi-lo, poderá enviar os mesmos comandos ao consumidor.

O presente relatório estará organizado num capítulo geral, que englobará todo o desenvolvimento do trabalho, e este será dividido em subcapítulos, correspondentes ao desenvolvimento e funcionamento das diferentes classes que compõem a aplicação.

3 Desenvolvimento

3.1 Cliente do Robot

Esta classe suporta o envio de comandos, na forma de mensagens, para o *buffer* circular, que é do tipo genérico Mensagem.

A Mensagem é uma classe abstrata que, à semelhança do trabalho prático anterior, armazena os atributos tipo, raio, angulo e parar. Conforme o tipo de Mensagem pretendida, deve-se instanciar a classe derivada, que poderá ser MsgEspacar, MsgReta, MsgCurvaEsq, MsgCurvaDir ou MsgParar, com os argumentos de que cada uma necessita e, o construtor da própria, irá invocar o da superclasse, com os atributos corretos. Estas classes encontram-se explicitadas na Figura 15.

De forma a conseguir interagir com qualquer *buffer* circular, que implemente a interface, a classe ClienteDoRobot tem um campo desse mesmo tipo. É-lhe passado como argumento, no seu construtor, ou colocada através do método setBuffer, o objeto *buffer* já inicializado, que será então guardado na variável global anteriormente referida.

Como se pode observar na Figura 1, outra variável global é do tipo Comportamento, onde é guardado o último comportamento que foi realizado. Deste, é possível obter a distância do último comportamento realizado, de forma a calcular a distância necessária para espaçar a figura anterior e a seguinte, para não se sobreponem no espaço bidimensional. O valor do espaçamento é calculado no método espacar que, à semelhança dos métodos reta, curvaEsq, curvaDir e parar, invoca o construtor da classe derivada de Mensagem correspondente. Passa como argumentos os valores recebidos como atributos do método, de distância, raio ou ângulo, e chama o método escreverBuffer, com a Mensagem que foi criada. O método escreverBuffer irá então chamar o método put do *buffer*, que irá colocar a Mensagem nele.

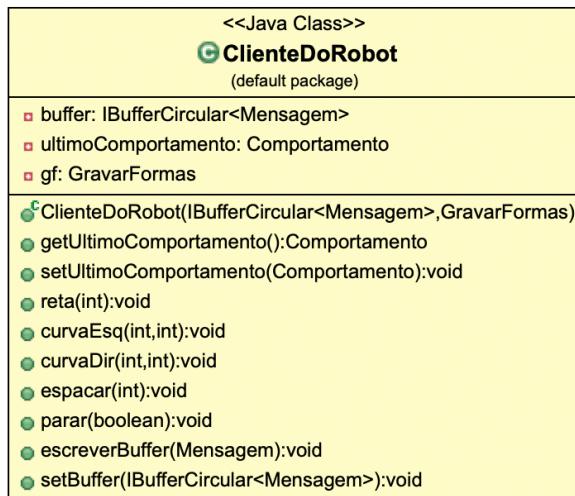


Figura 1 - Diagrama da classe ClienteDoRobot

3.2 Comportamentos

A classe abstrata **Comportamento**, conforme mostra o diagrama da Figura 2, guarda o valor da velocidade linear do robot, que no Robot Lego EV3 é de entre 27 a 30cm/s. Guarda, na variável global seguinte, um valor *booleano* que determina se as curvas que o robot irá efetuar serão para o lado esquerdo, caso seja *true*, ou para o lado direito, se for *false*. Armazena também o *clienteDoRobot*, que será um objeto da classe com o mesmo nome, passado no construtor das classes que implementem os métodos abstratos desta classe, bem como a *gui*, do tipo *IGUI*, um *booleano* que ao estar a *true* irá terminar a execução da tarefa, e o *estado* do tipo *Estado*. O enumerado *Estado* contém 2 estados possíveis: parado e ativo.

O método *run* desta tarefa funciona de acordo com a variável global *estado*, que pode assumir um Estado PARADO, e o Comportamento estará numa espera ativa, ou no Estado ATIVO, e chama o método abstrato de desenho, que irá ser executado no comportamento específico. Depois, invoca o método da GUI que indica que pode voltar a ativar os botões que permitem o desenho de mais formas e, por fim, torna ao estado PARADO.

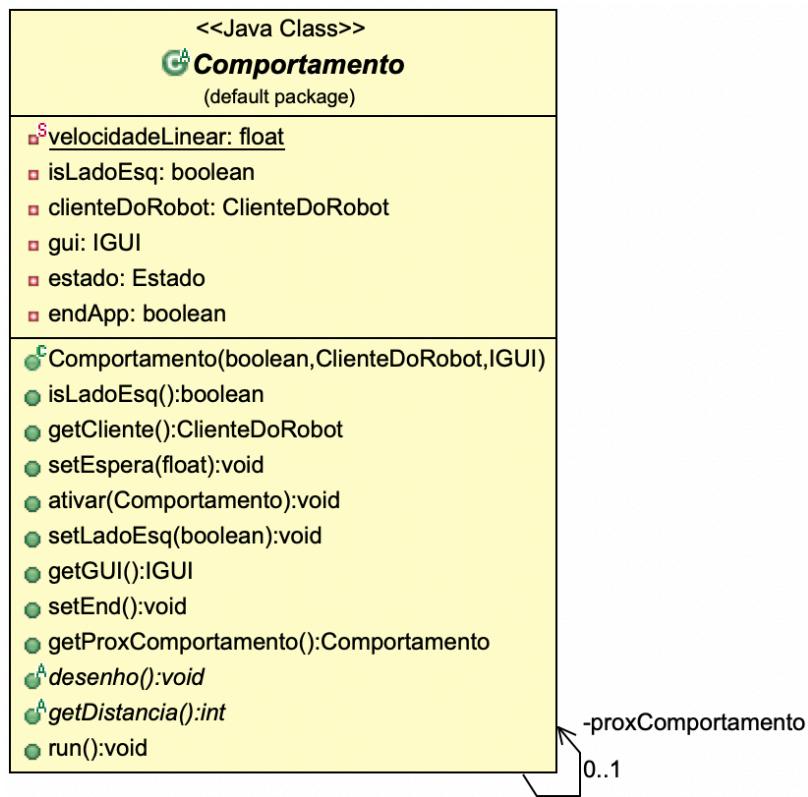


Figura 2 - Diagrama da classe **Comportamento**

3.2.1 Desenhar quadrado

A classe **DesenharQuadrado** é fruto do polimorfismo comum aos comportamentos. Assim, ao derivar de **Comportamento**, é por isso uma tarefa e herda os seus métodos que têm visibilidade pública, tendo de implementar aqueles que são abstratos. É responsável por guardar os valores do ângulo, raio e distância, necessários para criar os comandos que permitem o desenho de um quadrado.

O método desenho, ao ser invocado, obtém a distância da lateral, escolhida na GUI, cuja interface lhe foi passada como argumento no construtor e armazenada na superclasse. O ângulo é necessário para a definição da viragem do robot ou curva, que deve rondar os 90º, e o raio deve ser 0. Caso o método `isLadoEsq`, de Comportamento, retorne true, o quadrado irá ser desenhado para o lado esquerdo, caso contrário, para o lado direito.

De seguida, escolhe chamar os métodos de `ClienteDoRobot`, `reta`, `parar`, `curvaEsq`, `parar` e, de `Comportamento`, `setEspera`, nesta mesma sequência, de modo a formar um lado do quadrado. Tal repete-se mais 3 vezes, para formar um quadrado completo. Caso `isLadoEsq` retorne false, no lugar de `curvaEsq` é chamado o método `curvaDir`. O valor da espera, para aguardar que termine a lateral antes de serem criados e enviados mais comandos, é o tempo, em milissegundos, necessários a percorrer a distância e a realizar a curva, em função da velocidade do robot.

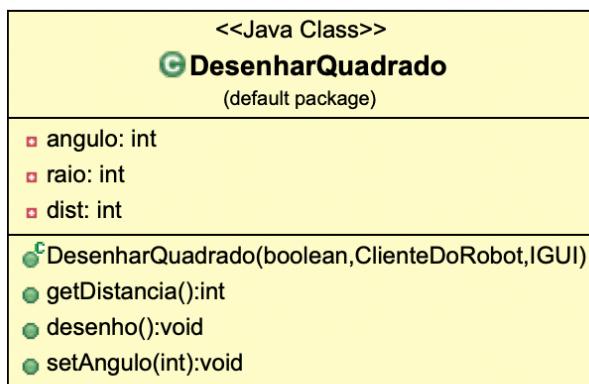


Figura 3 - Diagrama da classe `DesenharQuadrado`

3.2.2 Desenhar círculo

A classe `DesenharCirculo` também desfruta do polimorfismo inherente aos comportamentos, derivando da classe `Comportamento`, herdando os seus métodos e implementando aqueles que são abstratos. Esta armazena os valores de raio e ângulo, fundamentais para criar a forma de um círculo.

Ao ser invocado, o método `desenho` obtém o valor da distância que consta na GUI, cuja interface foi passada no construtor da superclasse e armazena essa distância como raio da circunferência. Por outro lado, o ângulo, que determina a curva do robot, vai ter valor de 360º, valor este que correspondente a uma volta completa. Caso o método `isLadoEsq` devolva true, é utilizado o método `curvaEsq` do `ClienteDoRobot`, o qual a classe recebeu como argumento no construtor, em conjunto com o método `setEspera`, herdado da classe `Comportamento`, para criar a forma pretendida. Caso `isLadoEsq` devolva false, é usado o método `curvaDir` em alternativa. O valor de espera para que o robot complete um círculo é dado em milissegundos e determinado consoante o raio em função da velocidade do robot.

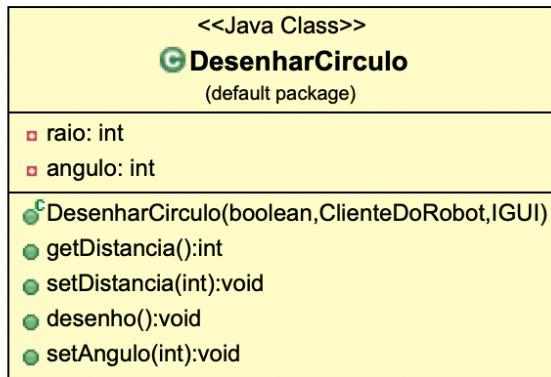


Figura 4 - Diagrama da classe **DesenharCirculo**

3.2.3 Espaçar formas geométricas

A classe **EspacarFormasGeometricas** é uma tarefa, à semelhança dos restantes comportamentos descritos nos dois subcapítulos anteriores, em que o seu método desenho apenas necessita de obter a distância necessária a percorrer para espaçar a forma geométrica anterior e a próxima a desenhar. A distância que deve percorrer, em reta, encontra-se na variável global, do tipo inteiro, *distancia*, como se pode observar na Figura 5.

O método desenho utiliza os métodos *espacar* e *parar* do *ClienteDoRobot*, e o *setEspera* de Comportamento, para desenhar a reta. Depois, invoca o *getter* da variável global de Comportamento que guarda o comportamento seguinte (*getProxComportamento()*), para o ativar. Assim, consegue-se um mecanismo de sincronismo em que o comportamento seguinte só irá ser ativo após o espaçador ter terminado o desenho da sua reta.

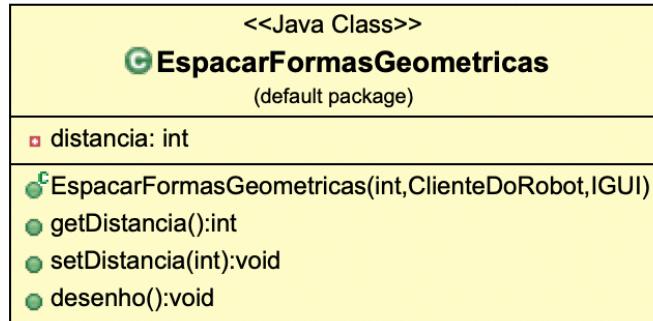


Figura 5 - Diagrama da classe **EspacarFormasGeometricas**

3.3 Buffer circular

Utilizámos como base para o *buffer* as classes fornecidas pelo docente. As únicas alterações feitas foram a alteração do método de leitura, para que este seja bloqueante sempre que uma tarefa consumidora queira ler um comando e não exista nenhum no *buffer*, e a alteração do método de escrita, para que este não seja bloqueante, podendo um processo produtor escrever sempre que quiser no *buffer*.

Assim, criou-se um *buffer* com um tamanho de 16 comandos e, com a ajuda de monitores, procedeu-se à implementação dos métodos put e get. O método put, sendo este não bloqueante, não precisa de acesso com exclusão mútua ao *buffer*. Logo, este coloca o elemento na posição em que estiver o apontador, avança com o apontador, atualiza a interface gráfica e notifica todas as tarefas que estiverem à espera de um comando para ler que já existe um novo. O método get, sendo este bloqueante, ficará à espera se não houver elementos no *buffer*. Quando aparecer um elemento, este é guardado, o apontador é incrementado, a interface gráfica é atualizada e o elemento é retornado.

O diagrama das classes que compõem o *buffer* circular é apresentado na Figura 6 e a *GUI* encontra-se na Figura 7, com o *buffer* vazio e com o *buffer* ocupado.

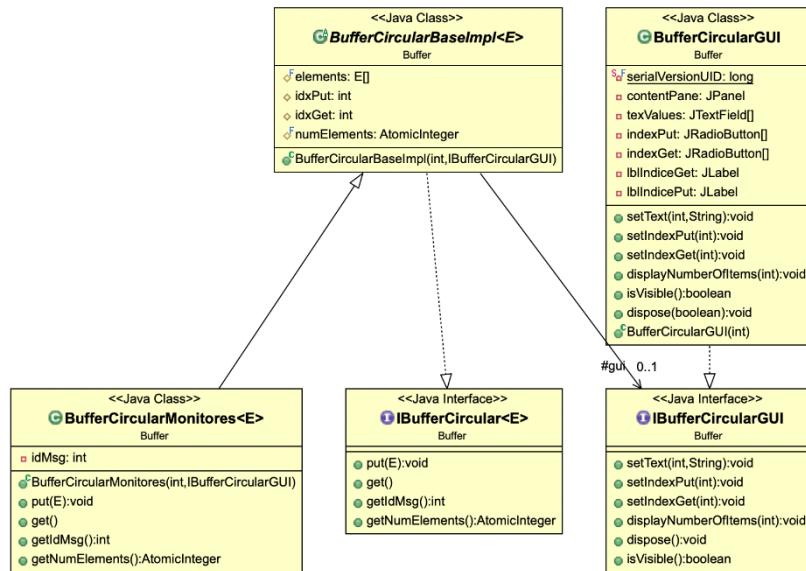


Figura 6 - Diagrama das classes que compõem o BufferCircular

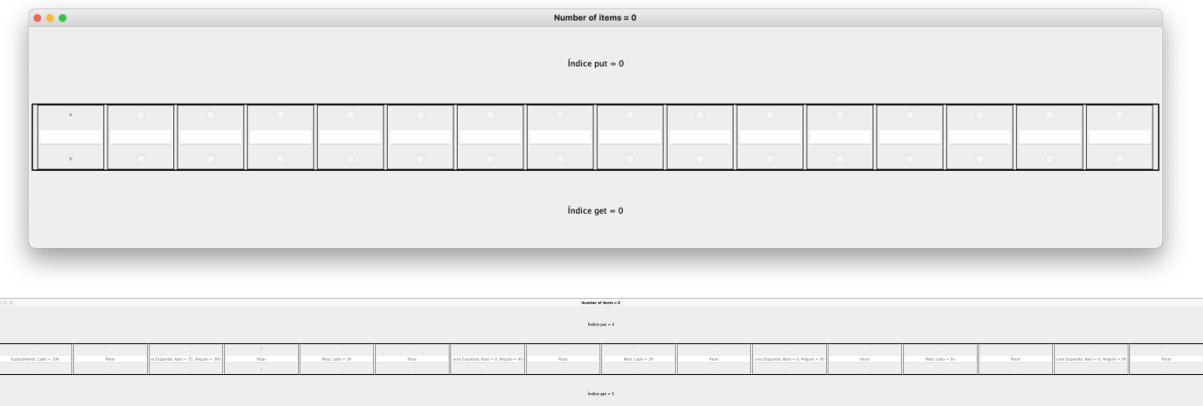


Figura 7 - GUI do BufferCircular

3.4 Servidor

A classe `ServidorDoRobot` é uma tarefa consumidora que, no seu método `run`, recorre ao `buffer` para invocar o seu método `get`. Irá ficar em espera passiva até que o número de elementos por consumir no `buffer` seja maior que 0, e aí recebe a Mensagem que, caso o `robotDesenhador` tenha sido definido, lhe envia. A variável global `endApp` permite interromper a execução da tarefa, alterável pelo método `setEnd`.

A Figura 8 mostra o diagrama da classe `ServidorDoRobot`.

A classe `ServidorDoRobot` é também composta por um objeto `GUIServidor`, cuja visibilidade é passível a alterações através do método `mostrarGUI`. Esta *Graphical User Interface* (*GUI*) encontra-se representada na Figura 9.

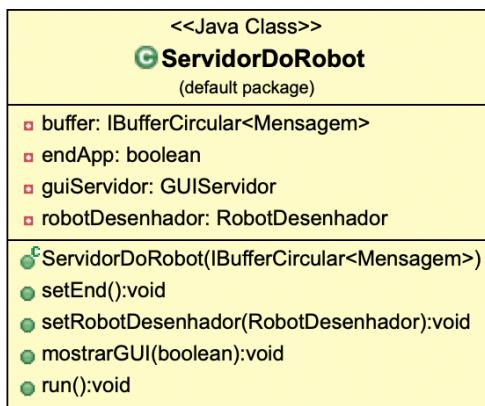


Figura 8 - Diagrama da classe ServidorDoRobot

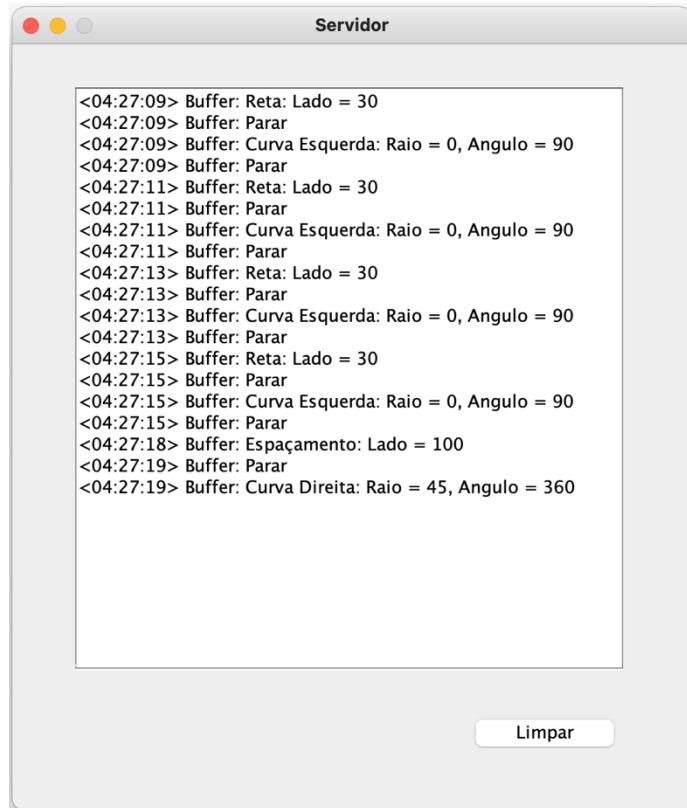


Figura 9 - GUI do servidor

3.5 Robot Desenhador

A classe RobotDesenhador funciona como uma interface de interação com os métodos da biblioteca RobotLegoEV3. Conforme o nome do robot, passado como argumento no construtor da classe, irá conectar-se ao respetivo robot no método conectar.

No seu construtor, é também criada uma *GUI* igual à do servidor, com a classe GUIServidor, para onde se enviam os *logs* das mensagens recebidas e executadas, visível na Figura 11. O envio de comandos para o robot aproveita o polimorfismo, visto que no método desenhar, conforme o tipo de objeto derivado de mensagem recebido, é executado o seu método executarComando. Este, utilizam as funções de comandos de movimento, da biblioteca RobotLegoEV3, bem como o conectar e o desconectar, que utilizam as funções de comandos de comunicação. Todas estas funções estão descritas no Anexo 2 das Folhas de FSO [1].

O diagrama de classes da classe resume estes e outros métodos e atributos que a classe contém para que funcione corretamente, presente na Figura 10.

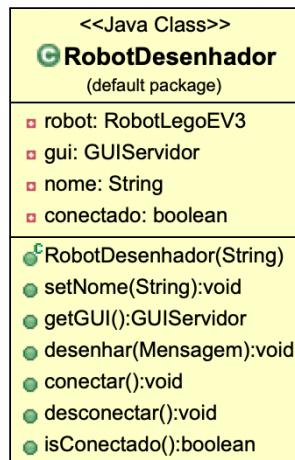


Figura 10 - Diagrama da classe RobotDesenhador

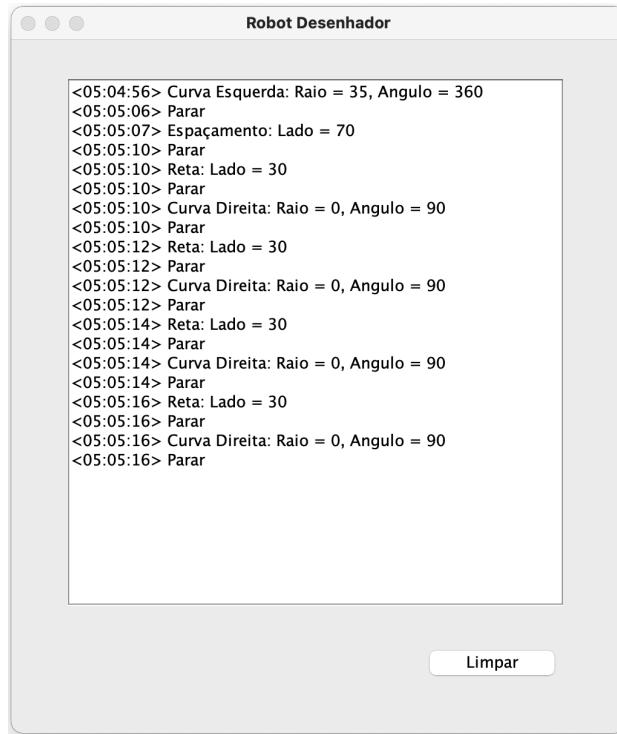


Figura 11 - GUI do Robot Desenhador

3.6 Estrutura, componentes e funcionalidade da GUI da aplicação

A GUI principal desta aplicação é uma *JFrame* e contém um *JPanel* onde estão inseridas as opções relativas aos comportamentos. Neste, estão contidos dois *JButton*, um para cada forma geométrica que é possível desenhar. Os seus *action listeners* são tratados pelas funções *handleDesenharQuadrado* e *handleDesenharCirculo*, que desativam os botões até instrução em contrário por parte dos comportamentos após o término dos desenhos, e invocam os métodos *adicionarQuadrado* e *adicionarCirculo*, respetivamente. Estes métodos ativam os comportamentos, ou seja, passam a uma variável estado a ATIVO e, caso o *clienteDoRobot* já tenha um último comportamento armazenado, invocam o método *adicionarEspacamento*. Todos estes atualizam os parâmetros dos comportamentos conforme o utilizador definiu. Para os quadrados, o atributo *distancia* é dado pelo lado, e para os círculos, o atributo *raio* é dado pelo raio. Os *JRadioButton* permitem definir a variável *isLadoEsq* de Comportamento. O *JSpinner* permite ajustar o ângulo das curvas a realizar, ao adicionar ou subtrair graus.

A *JFrame* tem também um *JTextField*, que permite definir o nome do robot e uma *JCheckBox* para executar o método *conectar* do *RobotDesenhador*. Por fim, tem três *JButton*: o primeiro permite ativar a gravação e reprodução de figuras; o segundo permite criar o *buffer* circular e o servidor; e o terceiro, que apenas é ativado após ter premido o segundo, que mostra a *GUI* do *RobotDesenhador*.

A interface gráfica resultou no que mostra a Figura 12 e o diagrama da classe encontra-se na Figura 13.

Ao fechar a aplicação, é mostrada a caixa de diálogo para que o utilizador possa efetivamente confirmar a saída, como acontece na Figura 14. Ao confirmar, todas as tarefas da aplicação, que estejam em execução, irão terminar.

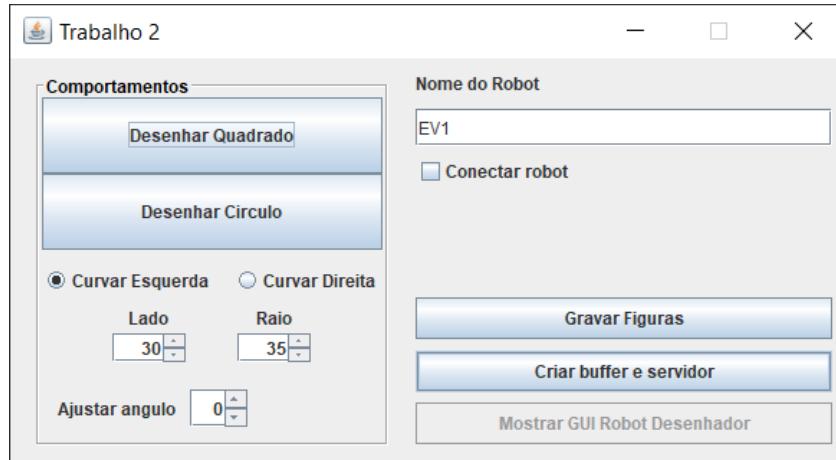


Figura 12 - GUI principal da aplicação



Figura 13 - Diagrama da classe GUI

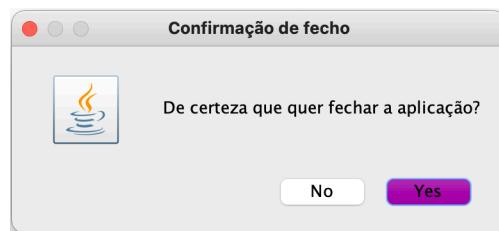


Figura 14 - Confirmação de fecho da aplicação

3.7 Diagrama de classes (sem a implementação do Gravar Formas)

Na Figura 15 apresenta-se o diagrama de todas as classes, com as variáveis e métodos de cada uma, sem a implementação do Gravar Formas.

A classe `GUILaunch` implementa a interface `Runnable` e as classes que mencionámos anteriormente como sendo tarefas derivam de `Thread`. Tal como se pode observar pelo diagrama, a *GUI* principal precisa de criar objetos das classes `ClienteDoRobot` e de cada um dos comportamentos. O `ServidorDoRobot`, ao ser criado, no seu construtor cria também um objeto `GUIServidor`. O mesmo acontece com a classe `RobotDesenhador`. Destaca-se ainda a presença de relações de herança nos comportamentos e nas mensagens.

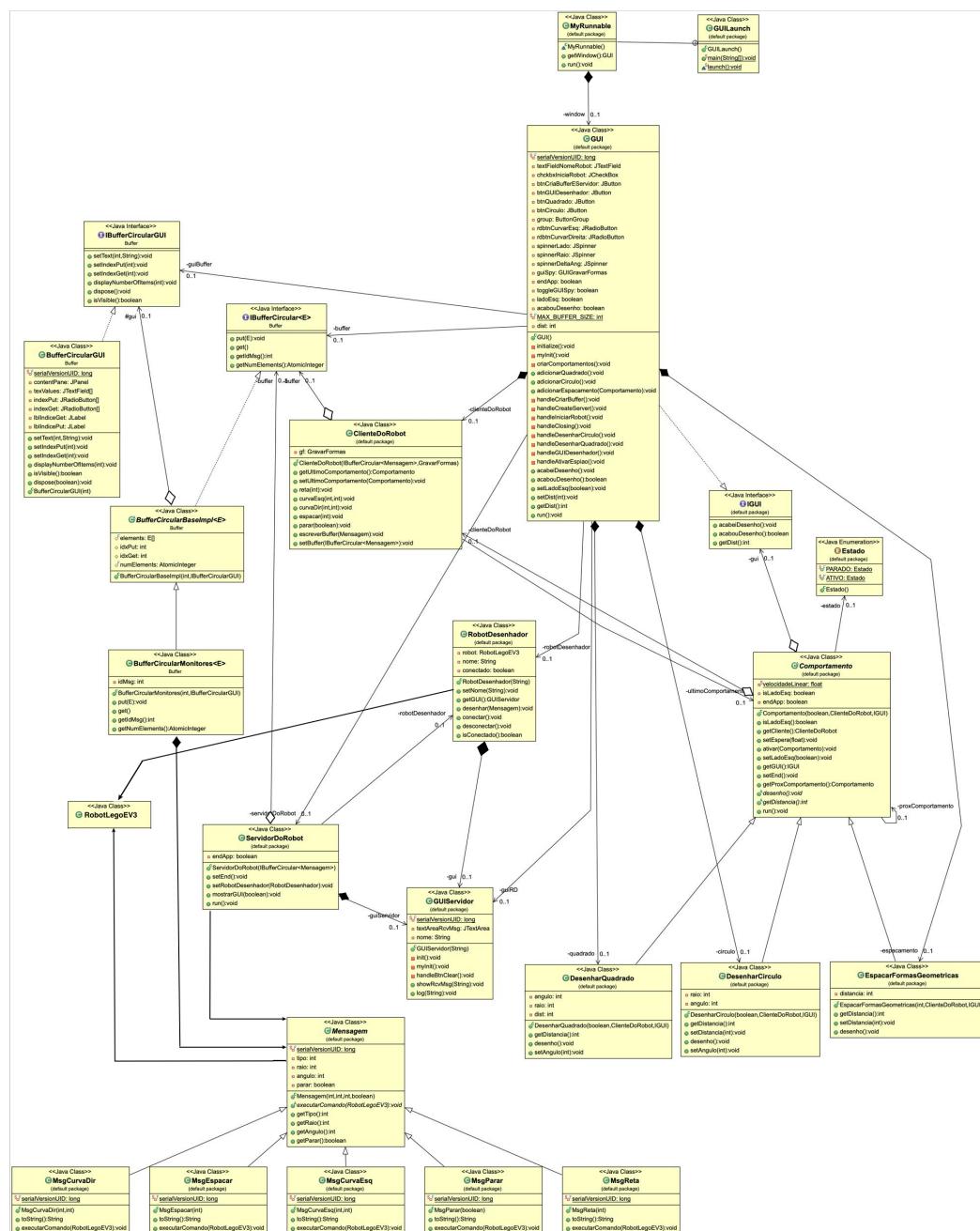


Figura 15 – Diagrama de todas as classes sem o Gravar Formas

3.8 Sincronização e comunicação entre tarefas desenhadoras e o Servidor

A comunicação entre as tarefas desenhadoras, sendo elas DesenharQuadrado, DesenharCírculo e EspacarFormasGeometricas, e o servidor, é feita pelo envio de objetos da classe Mensagem, para o *buffer* circular com o seu método put, por parte da ClienteDoRobot.

No caso do desenhador de quadrados e do desenhador de círculos, a ação destas tarefas não se sobrepõe dado que apenas é possível ativar uma de cada vez, ao premir o botão respetivo. Os botões são desativados enquanto acabe iDesenho retorna o valor falso, e este método não retorna true até que o comportamento que estiver ativo não termine o desenho.

Já no caso do espaçador, que é sempre ativado antes da execução de outra forma à exceção da primeira, a sua sincronização com os restantes comportamentos faz-se de maneira que o comportamento da forma seguinte só seja ativo, não ao premir o botão, mas no final do método desenho do EspacadorFormasGeometricas.

A *thread* ServidorDoRobot mantém-se, em espera passiva, no método get do *buffer* enquanto não for notificada da existência de posições ocupadas com mensagens para serem consumidas.

3.9 Gravador de Formas

O gravador de formas é uma tarefa que permite observar e guardar num ficheiro as mensagens que passam pelo *buffer* para depois serem lidas e reproduzidas no robot. Neste tópico, havia duas abordagens possíveis: colocar o gravador à escuta de mensagens à entrada do *buffer* ou à saída deste. Foi escolhida a primeira opção, pois assim não é necessário inicializar o *buffer* para o funcionamento do gravador. A Figura 18 mostra o diagrama de classes que integram o mecanismo de gravação e reprodução de formas.

3.9.1 Estrutura da gravação

Para o gravador observar as mensagens, criou-se um botão *toggle* que permite comutar o estado do gravador entre “Gravar” e “Parado”. Depois, criou-se o método `adicionarMensagem` no `GravarFormas` e este é utilizado dentro do `ClienteDoRobot`, recebendo como argumento a mensagem no momento antes de ser colocada no *buffer*. Dentro deste método, caso o gravador esteja no estado “Gravar”, adiciona a mensagem recebida para um *array*. Além disso, também é necessário guardar os tempos de espera entre cada momento. Então, é calculada a diferença de tempo entre o último comando recebido e o atual e esse valor é adicionado logo após a inserção da mensagem no mesmo *array*.

Tendo já as mensagens pretendidas guardadas em memória cada uma com o tempo de espera correspondente, na `GUIGravarFormas`, colocou-se um botão “Escrever no ficheiro”. Este muda o estado do gravador (`GravarFormas`) para “Escrever” e irá chamar o método `guardarMensagens` que utiliza a classe `ObjectOutputStream` para escrever objetos num ficheiro local. Estes objetos são o *array* (`arrMensagens`) que contém as mensagens e os tempos de espera. A Figura 16 apresenta o Gravar Formas após a escrita do *array* no ficheiro.

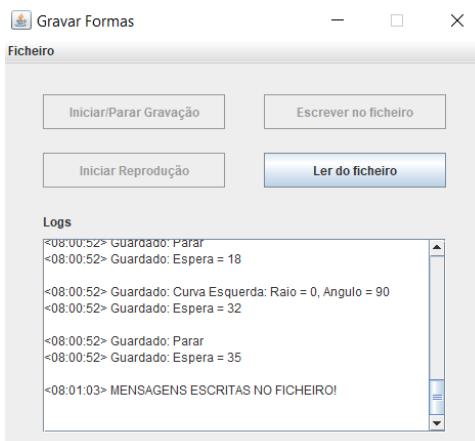


Figura 16 - Gravar Formas após escrita no ficheiro

3.9.2 Reprodução no Robot

A este ponto, tem-se um ficheiro criado localmente com as mensagens gravadas, pelo que é possível ler o conteúdo deste ficheiro sem o alterar. Para isso, através do botão “Ler do ficheiro” que muda o estado do gravador para “Ler”, irá receber o conteúdo do ficheiro utilizando a classe

`ObjectInputStream`. Este conteúdo são as mensagens e os tempos de espera tal como entraram no momento de escrita do ficheiro, que irão ser colocados num *array*.

Por fim, tendo as mensagens lidas num *array*, é possível reproduzi-las vezes sem conta no robot através do botão “Iniciar Reprodução”, que irá mudar o estado do gravador para “Reproduzir”. No momento de reprodução, o *array* de mensagens é percorrido e, sabendo que cada tempo de espera está nos índices ímpares, consegue-se simular com igualdade as mensagens como se tivessem sido enviadas pelas tarefas dos comportamentos. A Figura 17 apresenta o Gravar Formas após a leitura das mensagens do ficheiro e quando é iniciada a reprodução.

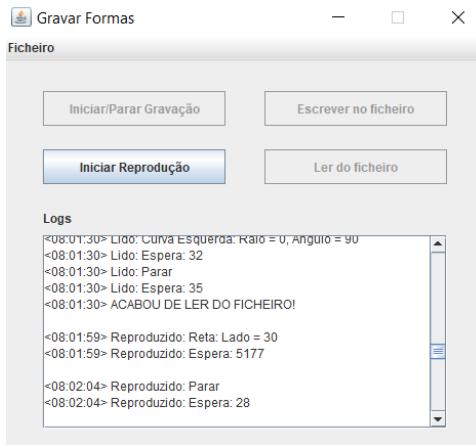


Figura 17 - Gravar Formas no início do momento de reprodução

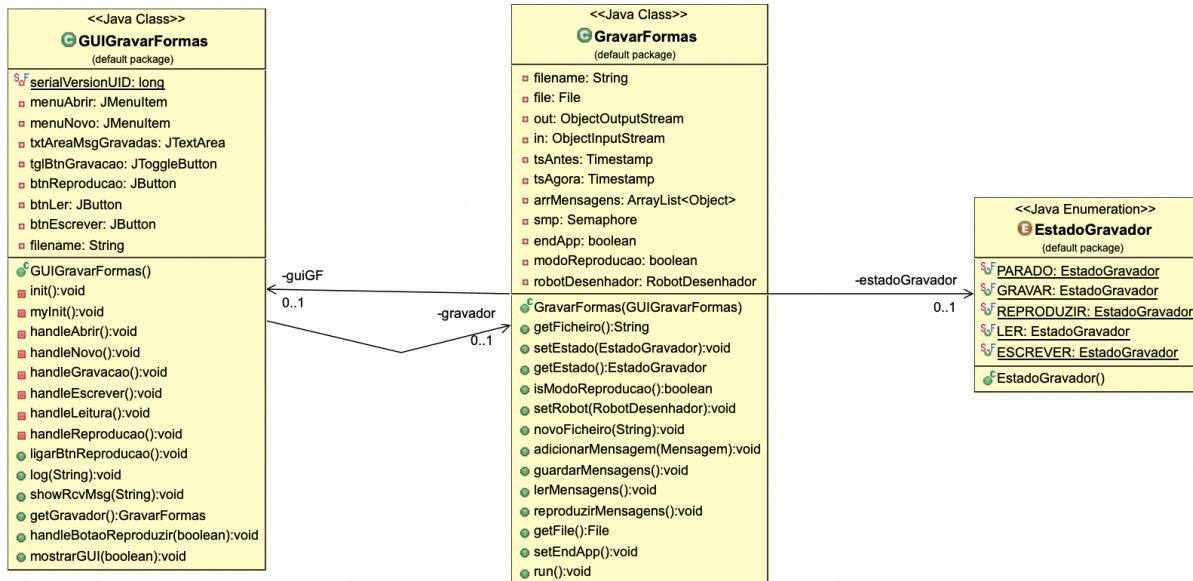


Figura 18 – Diagrama das classes que constituem o processo de gravar formas

3.10 Sincronização no acesso ao robot entre todas as tarefas

As tarefas que podem ter acesso ao robot desenhador são a tarefa do ServidorDoRobot e a do GravarFormas. Pretendia-se que, enquanto o gravador de formas reproduzisse as formas no robot, os comportamentos não poderiam gerar comandos. O gravador também não pode reproduzir se o robot estiver a receber um comportamento do *buffer*. Assim, a solução encontrada foi a desativação dos botões das *GUIs* das tarefas que iniciam a reprodução, ou seja, o gravador ao reproduzir desativa os botões para gerar formas e vice-versa.

Quando GravarFormas está no estado REPRODUZIR, o método run da GUI principal coloca os botões de desenho de formas inativos e, quando muda para outro estado, torna a ativá-los.

Enquanto houver desenho de formas por parte dos comportamentos, o método acabouDesenho devolve false, e os botões permanecem desativos. Quando o Comportamento acaba de enviar o seu último comando, este invoca o método acabeiDesenho da GUI principal e os botões de reprodução da GUI principal e do gravador de formas são ativados.

3.11 Diagrama de classes (com a implementação do Gravar Formas)

Na Figura 19 é apresentado um diagrama de classes simplificado para melhor ajudar a perceber as ligações e as dependências entre as classes, bem como a entender o funcionamento da aplicação. Na Figura 20 está o diagrama completo com as variáveis e os métodos de cada classe.

Pelo diagrama, pode-se ver que acrescentar o gravador de formas, além das alterações no sincronismo mencionadas anteriormente, apenas adicionou a necessidade de dar a conhecer a classe ao ClienteDoRobot e a de que a *GUI* principal crie uma GUIGravarFormas, que se torna visível ao pressionar o botão respetivo na interface gráfica, que por sua vez cria uma instância da classe GravarFormas.

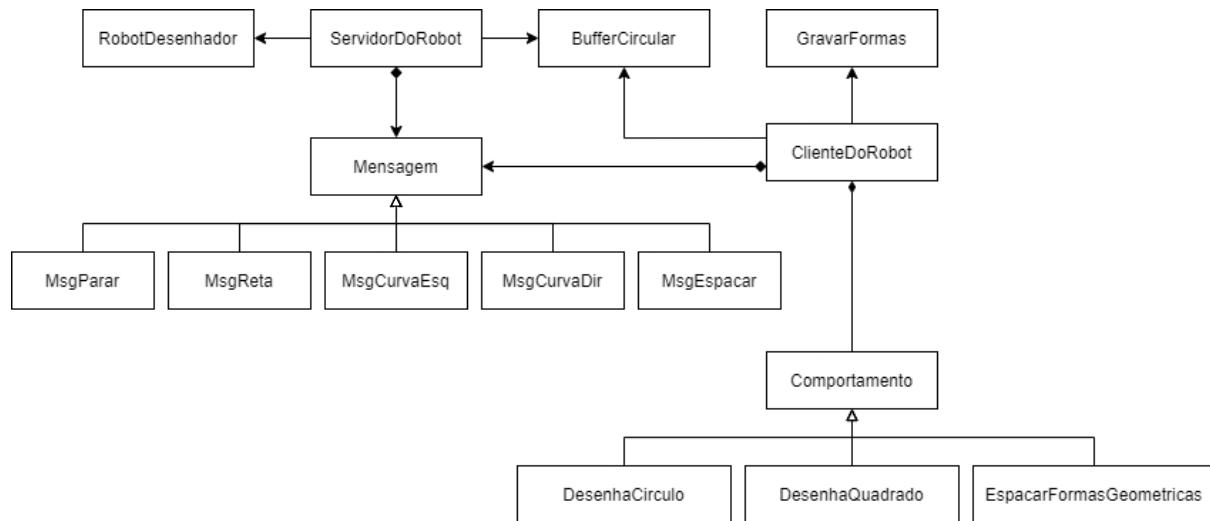


Figura 19 – Diagrama de todas as classes simplificado

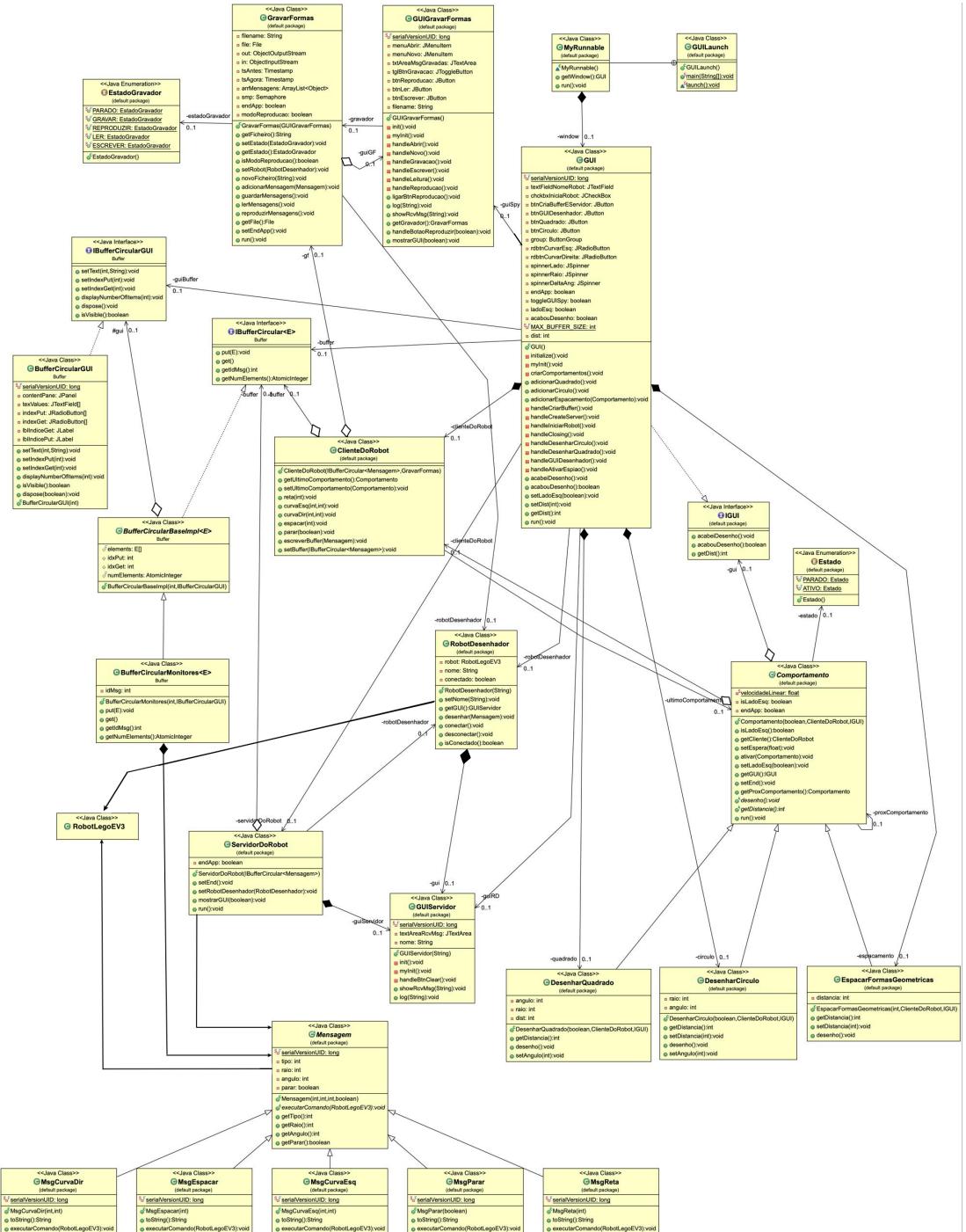


Figura 20 – Diagrama de todas as classes com o Gravar Formas

4 Conclusões

Durante a realização do relatório, o grupo apercebeu-se que estava em falta o sincronismo no acesso ao robot, pelo que a implementação referida no ponto 3.10 foi realizada depois da validação do funcionamento da aplicação.

Considera-se que o trabalho foi implementado com sucesso, dado que este foi testado múltiplas vezes, graficamente e fisicamente com a ajuda de um robot EV3, cumprindo os requisitos pedidos para a sua realização.

Assim, a realização deste trabalho prático permitiu aprofundar e praticar os conceitos adquiridos acerca da comunicação e sincronização entre tarefas Java, utilizando um modelo produtor-consumidor para obter uma melhor percepção do seu funcionamento, bem como dos mecanismos de sincronização entre tarefas.

5 Bibliografia

[1] PAIS, Jorge. "Fundamentos de Sistemas Operativos, versão 2", 2020-2021

6 ANEXO I – Código Java

6.1 Mensagens

6.1.1 Classe Mensagem

```
import java.io.Serializable;

public abstract class Mensagem implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = -5453087572800218536L;
    private int tipo;
    private int raio;
    private int angulo;
    private boolean parar;

    public Mensagem(int tipo, int raio, int angulo, boolean parar) {
        this.tipo = tipo;
        this.raio = raio;
        this.angulo = angulo;
        this.parar = parar;
    }

    public abstract void executarComando(RobotLegoEV3 r);

    public int getTipo() {
        return tipo;
    }

    public int getRaio() {
        return raio;
    }

    public int getAngulo() {
        return angulo;
    }

    public boolean getParar() {
        return parar;
    }
}
```

6.1.2 Classe MsgCurvaDir

```
import java.io.Serializable;

public class MsgCurvaDir extends Mensagem implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = -4779784962391177392L;

    public MsgCurvaDir(int raio, int angulo) {
        super(3, raio, angulo, false);
    }

    @Override
    public String toString() {
        return "Curva Direita: Raio = " + getRaio() + ", Angulo = " + getAngulo();
    }

    @Override
    public void executarComando(RobotLegoEV3 r) {
        r.CurvarDireita(getRaio(), getAngulo());
    }
}
```

6.1.3 Classe MsgCurvaEsq

```
import java.io.Serializable;

public class MsgCurvaEsq extends Mensagem implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 5232271624026947085L;

    public MsgCurvaEsq(int raio, int angulo) {
        super(1, raio, angulo, false);
    }

    @Override
    public String toString() {
        return "Curva Esquerda: Raio = " + getRaio() + ", Angulo = " + getAngulo();
    }

    @Override
    public void executarComando(RobotLegoEV3 r) {
        r.CurvarEsquerda(getRaio(), getAngulo());
    }
}
```

6.1.4 Classe MsgEspacar

```
import java.io.Serializable;

public class MsgEspacar extends Mensagem implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = -6693531347029306457L;

    public MsgEspacar(int distancia) {
        super(4, distancia, 0, false);
    }

    @Override
    public String toString() {
        return "Espaçamento: Lado = " + getRaio();
    }

    @Override
    public void executarComando(RobotLegoEV3 r) {
        r.Reta(getRaio());
    }
}
```

6.1.5 Classe MsgParar

```
import java.io.Serializable;

public class MsgParar extends Mensagem implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 7578689077290986064L;

    public MsgParar(boolean parar) {
        super(2, 0, 0, parar);
    }

    @Override
    public String toString() {
        return "Parar";
    }

    @Override
    public void executarComando(RobotLegoEV3 r) {
        r.Parar(getParar());
    }
}
```

6.1.6 Classe MsgReta

```
import java.io.Serializable;

public class MsgReta extends Mensagem implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 5771700311899782645L;

    public MsgReta(int distancia) {
        super(0, distancia, 0, false);
    }

    @Override
    public String toString() {
        return "Reta: Lado = " + getRaio();
    }

    @Override
    public void executarComando(RobotLegoEV3 r) {
        r.Reta(getRaio());
    }
}
```

6.2 Produtor

6.2.1 Classe ClienteDoRobot

```
import Buffer.IBufferCircular;

public class ClienteDoRobot {

    private IBufferCircular<Mensagem> buffer;
    private Comportamento ultimoComportamento;
    private GravarFormas gf;

    public ClienteDoRobot(IBufferCircular<Mensagem> buffer, GravarFormas gf) {
        this.buffer = buffer;
        this.gf = gf;
    }

    public Comportamento getUltimoComportamento() {
        return ultimoComportamento;
    }

    public void setUltimoComportamento(Comportamento ultimoComportamento) {
        this.ultimoComportamento = ultimoComportamento;
    }

    public void reta(int dist) {
        escreverBuffer(new MsgReta(dist));
    }

    public void curvaEsq(int raio, int angulo) {
        escreverBuffer(new MsgCurvaEsq(raio, angulo));
    }

    public void curvaDir(int raio, int angulo) {
        escreverBuffer(new MsgCurvaDir(raio, angulo));
    }

    public void espacar(int dist) {
        escreverBuffer(new MsgEspacar(ultimoComportamento.getDistancia() + dist));
    }

    public void parar(boolean value) {
        escreverBuffer(new MsgParar(value));
    }

    public void escreverBuffer(Mensagem msg) {
        try {
            // Enviar msg para o gravador
            gf.adicionarMensagem(msg);
            if (buffer == null) throw new InterruptedException();
            buffer.put(msg);
        } catch (InterruptedException e) {
            System.err.println("Problemas no buffer. Não foi enviada a mensagem " + msg);
        }
    }

    public void setBuffer(IBufferCircular<Mensagem> buffer) {
        this.buffer = buffer;
    }
}
```

6.2.2 Enumerado Estado

```
public enum Estado {  
    PARADO, ATIVO  
}
```

6.2.3 Classe Comportamento

```
public abstract class Comportamento extends Thread {

    private static float velocidadeLinear = 22.0f;
    private boolean isLadoEsq;
    private ClienteDoRobot clienteDoRobot;
    private IGUI gui;
    private Estado estado;
    private Comportamento proxComportamento;
    private boolean endApp;

    public Comportamento(boolean isLadoEsq, ClienteDoRobot clienteDoRobot, IGUI gui) {
        this.isLadoEsq = isLadoEsq;
        this.clienteDoRobot = clienteDoRobot;
        this.gui = gui;
        estado = Estado.PARADO;
    }

    public boolean isLadoEsq() {
        return isLadoEsq;
    }

    public ClienteDoRobot getCliente() {
        return clienteDoRobot;
    }

    public void setEspera(float raioOrDist) {
        try {
            Thread.sleep((long) ((raioOrDist / velocidadeLinear) * 1000.0f));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void ativar(Comportamento c) {
        proxComportamento = c;
        estado = Estado.ATIVO;
    }

    public void setLadoEsq(boolean value) {
        this.isLadoEsq = value;
    }

    public IGUI getGUI() {
        return this.gui;
    }

    public void setEnd(){
        this.endApp = true;
    }

    public Comportamento getProxComportamento() {
        return proxComportamento;
    }

    public abstract void desenho() throws InterruptedException;

    public abstract int getDistancia();

    @Override
    public void run() {
        for (;;) {
            if (endApp)
                break;
            switch (estado) {
            case PARADO:
                try {
                    Thread.sleep(10);
                
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        break;
    case ATIVO:
        try {
            desenho();
            if (!(this instanceof EspacarFormasGeometricas))
                gui.acabeiDesenho();
            estado = Estado.PARADO;
        } catch (Exception e) {
            e.printStackTrace();
        }
        break;
    }
}
}
```

6.2.4 Classe DesenharCirculo

```
public class DesenharCirculo extends Comportamento {

    private int raio = 35;
    private int angulo = 360;

    public DesenharCirculo(boolean isLadoEsq, ClienteDoRobot clienteDoRobot, IGUI gui) {
        super(isLadoEsq, clienteDoRobot, gui);
    }

    public int getDistancia() {
        return raio * 2;
    }

    public void setDistancia(int raio) {
        this.raio = raio;
    }

    public void desenho() {
        raio = getGUI().getDist();
        if (isLadoEsq())
            getCliente().curvaEsq(raio, angulo);
        else
            getCliente().curvaDir(raio, angulo);
        setEspera((long) (2 * Math.PI * raio));
        getCliente().parar(false);
        setEspera(10);
    }

    public void setAngulo(int angulo) {
        this.angulo = angulo;
    }
}
```

6.2.5 Classe DesenharQuadrado

```
public class DesenharQuadrado extends Comportamento {

    private int angulo = 91;
    private int raio = 0;
    private int dist;

    public DesenharQuadrado(boolean isLadoEsq, ClienteDoRobot clienteDoRobot, IGUI gui) {
        super(isLadoEsq, clienteDoRobot, gui);
    }

    public int getDistancia() {
        return dist;
    }

    public void desenho() {
        dist = getGUI().getDist();
        if (isLadoEsq()) {
            for (int i = 0; i < 4; i++) {
                getCliente().reta(dist);
                getCliente().parar(false);
                getCliente().curvaEsq(raio, angulo);
                getCliente().parar(false);
                setEspera(dist + 13.75f);
            }
        } else {
            for (int i = 0; i < 4; i++) {
                getCliente().reta(dist);
                getCliente().parar(false);
                getCliente().curvaDir(raio, angulo);
                getCliente().parar(false);
                setEspera(dist + 13.75f);
            }
        }
    }

    public void setAngulo(int angulo) {
        this.angulo = angulo;
    }
}
```

6.2.6 Classe EspacarFormasGeometricas

```
public class EspacarFormasGeometricas extends Comportamento {  
    private int distancia;  
  
    public EspacarFormasGeometricas(int distancia, ClienteDoRobot clienteDoRobot, IGUI gui) {  
        super(false, clienteDoRobot, gui);  
        this.distancia = distancia;  
    }  
  
    public int getDistancia() {  
        return distancia;  
    }  
  
    public void setDistancia(int dist) {  
        this.distancia = dist;  
    }  
  
    public void desenho() {  
        getCliente().espacar(distancia);  
        setEspera(distancia);  
        getCliente().parar(false);  
        setEspera(10);  
        if (getProxComportamento() != null)  
            getProxComportamento().ativar(null);  
    }  
}
```

6.3 *Buffer*

6.3.1 Interface IBufferCircular

```
package Buffer;

import java.util.concurrent.atomic.AtomicInteger;

public interface IBufferCircular<E> {

    void put(E e) throws InterruptedException;
    E get() throws InterruptedException;
    int getIdMsg();
    AtomicInteger getNumElements();
}
```

6.3.2 Interface IBufferCircularGUI

```
package Buffer;

public interface IBufferCircularGUI {

    void setText(int pos, String text);

    void setIndexPut(int pos);

    void setIndexGet(int pos);

    void displayNumberOfItems(int numberOfItems);

    void dispose();

    boolean isVisible();
}
```

6.3.3 Classe BufferCircularBaseImpl

```
package Buffer;

import java.util.concurrent.atomic.AtomicInteger;

public abstract class BufferCircularBaseImpl<E> implements IBufferCircular<E> {

    protected final E[] elements;

    protected int idxPut;
    protected int idxGet;

    protected final AtomicInteger numElements;

    protected final IBufferCircularGUI gui;

    @SuppressWarnings("unchecked")
    public BufferCircularBaseImpl(int size, IBufferCircularGUI gui) {
        this.gui = gui;

        this.elements = (E[]) new Object[size];

        this.numElements = new AtomicInteger(0);
        this.idxPut = this.idxGet = 0;

        this.gui.setIndexPut(this.idxPut);
        this.gui.setIndexGet(this.idxGet);
        this.gui.displayNumberOfItems(0);
    }
}
```

6.3.4 Classe BufferCircularMonitores

```
package Buffer;

import java.util.concurrent.atomic.AtomicInteger;

public class BufferCircularMonitores<E> extends BufferCircularBaseImpl<E> {

    private int idMsg = 0;

    public BufferCircularMonitores(int size, IBufferCircularGUI gui) {
        super(size, gui);
    }

    @Override
    public synchronized void put(E e) throws InterruptedException {

        // Colocar o novo elemento na posicao correspondente
        this.elements[this.idxPut] = e;

        // Atualizar a interface grafica (mostrar o elemento inserido)
        this.gui.setText(this.idxPut, e.toString());

        // Atualizar o proximo indice de put (posicao onde se coloca o proximo elemento)
        ++this.idxPut;
        this.idxPut %= this.elements.length;
        ++this.idMsg;

        // Atualizar a interface grafica (posicao onde se coloca o proximo elemento)
        this.gui.setIndexPut(this.idxPut);

        if (this.numElements.get() < this.elements.length) {
            // Atualizar o numero de elementos no buffer
            this.gui.displayNumberOfItems(this.numElements.incrementAndGet());
        }

        this.notifyAll();
    }

    @Override
    public synchronized E get() throws InterruptedException {
        E result;

        // Enquanto o contentor estiver vazio temos de esperar
        while (this.numElements.get() == 0) {
            this.wait();
        }

        // Neste ponto existe um elemento para consumir

        // Obter o elemento mais antigo
        result = this.elements[this.idxGet];

        // Atualizar o proximo indice de get (posicao de onde se obtém o proximo
        // elemento)
        ++this.idxGet;
        this.idxGet %= this.elements.length;

        // Atualizar a interface grafica (posicao de onde se obtém o proximo elemento)
        this.gui.setIndexGet(this.idxGet);

        // Atualizar o numero de elementos no buffer
        this.gui.displayNumberOfItems(this.numElements.decrementAndGet());

        return result;
    }

    public int getIdMsg() {
        return this.idMsg;
    }
}
```

```
}

public AtomicInteger getNumElements() {
    return this.numElements;
}
```

6.3.5 Classe BufferCircularGUI

```
package Buffer;

import javax.swing.*;
import javax.swing.border.EmptyBorder;
import javax.swing.border.LineBorder;
import java.awt.*;

public class BufferCircularGUI extends JFrame implements IBufferCircularGUI {

    private static final long serialVersionUID = -103615988849405009L;

    private JPanel contentPane;

    private JTextField[] texValues;
    private JRadioButton[] indexPut;
    private JRadioButton[] indexGet;
    private JLabel lblIndiceGet;
    private JLabel lblIndicePut;

    public void setText(int pos, String text) {
        Runnable r = () -> texValues[pos].setText(text);
        if (SwingUtilities.isEventDispatchThread()) {
            r.run();
        } else {
            SwingUtilities.invokeLater(r);
        }
    }

    public void setIndexPut(int pos) {
        Runnable r = () -> {
            indexPut[pos]. setSelected(true);
            lblIndicePut.setText(String.format("índice put = %d", pos));
        };
        if (SwingUtilities.isEventDispatchThread()) {
            r.run();
        } else {
            SwingUtilities.invokeLater(r);
        }
    }

    public void setIndexGet(int pos) {
        Runnable r = () -> {
            indexGet[pos]. setSelected(true);

            lblIndiceGet.setText(String.format("índice get = %d", pos));
        };
        if (SwingUtilities.isEventDispatchThread()) {
            r.run();
        } else {
            SwingUtilities.invokeLater(r);
        }
    }

    public void displayNumberOfItems(int numberOfItems) {
        Runnable r = () -> setTitle(String.format("Number of items = %d", numberOfItems));
        if (SwingUtilities.isEventDispatchThread()) {
            r.run();
        } else {
            SwingUtilities.invokeLater(r);
        }
    }

    public boolean isVisible() {
        return isShowing();
    }

    public void dispose(boolean state) {
```

```

        if (state) {
            dispose();
        } else {
            setVisible(true);
        }
    }

    public BufferCircularGUI(int dim) {
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        setBounds(100, 100, 550, 300);
        contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);
        contentPane.setLayout(new GridLayout(3, 1, 10, 10));

        lblIndicePut = new JLabel("\u00cdndice de Put");
        lblIndicePut.setHorizontalAlignment(SwingConstants.CENTER);
        contentPane.add(lblIndicePut);

        JPanel panelElements = new JPanel();
        panelElements.setBorder(new LineBorder(new Color(0, 0, 0), 2, true));
        contentPane.add(panelElements);
        panelElements.setLayout(new GridLayout(1, dim, 5, 5));

        this.indexPut = new JRadioButton[dim];
        this.indexGet = new JRadioButton[dim];
        this.texValues = new JTextField[dim];

        ButtonGroup bgPut = new ButtonGroup();
        ButtonGroup bgGet = new ButtonGroup();

        for (int idx = 0; idx < dim; ++idx) {
            JPanel panelElement = new JPanel();
            panelElement.setBorder(new LineBorder(new Color(0, 0, 0)));
            panelElement.setLayout(new GridLayout(3, 1, 5, 5));

            this.indexPut[idx] = new JRadioButton("");
            this.indexPut[idx].setHorizontalAlignment(SwingConstants.CENTER);
            this.indexPut[idx].setEnabled(false);
            bgPut.add(this.indexPut[idx]);
            panelElement.add(this.indexPut[idx]);

            this.texValues[idx] = new JTextField();
            this.texValues[idx].setHorizontalAlignment(SwingConstants.CENTER);
            this.texValues[idx].setText("");
            // this.texValues[ idx ].setColumns( 10 );
            this.texValues[idx].setEnabled(false);
            panelElement.add(this.texValues[idx]);

            this.indexGet[idx] = new JRadioButton("");
            this.indexGet[idx].setHorizontalAlignment(SwingConstants.CENTER);
            this.indexGet[idx].setEnabled(false);
            bgGet.add(this.indexGet[idx]);
            panelElement.add(this.indexGet[idx]);

            panelElements.add(panelElement);
        }

        lblIndiceGet = new JLabel("\u00cdndice de Get");
        lblIndiceGet.setHorizontalAlignment(SwingConstants.CENTER);
        contentPane.add(lblIndiceGet);

        this.setIndexPut(0);
        this.setIndexGet(0);

        this.setVisible(true);
        this.setSize(1800, 380);
    }
}

```

6.4 Consumidor

6.4.1 Classe ServidorDoRobot

```
import Buffer.IBufferCircular;

public class ServidorDoRobot extends Thread {
    private IBufferCircular<Mensagem> buffer;
    private boolean endApp;
    private GUIServidor guiServidor;
    private RobotDesenhador robotDesenhador;

    public ServidorDoRobot(IBufferCircular<Mensagem> buffer) {
        this.buffer = buffer;
        this.guiServidor = new GUIServidor("Servidor");
        guiServidor.setVisible(true);
        this.robotDesenhador = null;
        endApp = false;
    }

    public void setEnd() {
        this.endApp = true;
        guiServidor.dispose();
    }

    public void setRobotDesenhador(RobotDesenhador robotDesenhador) {
        this.robotDesenhador = robotDesenhador;
    }

    public void mostrarGUI(boolean value) {
        guiServidor.setVisible(value);
    }

    @Override
    public void run() {
        for (;;) {
            try {
                Thread.sleep(10);
                if (this.endApp)
                    break;

                Mensagem m = buffer.get();
                guiServidor.log("Buffer: " + m.toString());

                if (robotDesenhador != null) {
                    robotDesenhador.desenhar(m);
                    guiServidor.log("Robot Desenhador: " + m.toString());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

6.4.2 Classe GUIServidor

```
import javax.swing.*;  
  
import java.lang.reflect.InvocationTargetException;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
public class GUIServidor extends JFrame {  
    /**  
     *  
     */  
    private static final long serialVersionUID = 7852897540266445539L;  
    private JTextArea textAreaRcvMsg;  
    private String nome;  
  
    public GUIServidor(String nome) {  
        this.nome = nome;  
        init();  
        myInit();  
    }  
  
    private void init() {  
        setTitle(nome);  
        setResizable(true);  
        getContentPane().setLayout(null);  
  
        JScrollPane scrollPaneRcvMsg = new JScrollPane();  
        // redimensiona aqui  
        scrollPaneRcvMsg.setBounds(47, 32, 409, 435);  
        getContentPane().add(scrollPaneRcvMsg);  
  
        textAreaRcvMsg = new JTextArea();  
        scrollPaneRcvMsg.setViewportView(textAreaRcvMsg);  
  
        JButton btnClear = new JButton("Limpar");  
        btnClear.addActionListener(e -> handleBtnClear());  
        btnClear.setBounds(339, 500, 117, 29);  
        getContentPane().add(btnClear);  
  
        setSize(508, 600);  
        setVisible(false);  
        setResizable(false);  
    }  
  
    private void myInit() {  
        textAreaRcvMsg.setEditable(false);  
    }  
  
    private void handleBtnClear() {  
        textAreaRcvMsg.setText("");  
    }  
  
    public void showRcvMsg(String rcvd) {  
        Runnable r = new Runnable() {  
            @Override  
            public void run() {  
                textAreaRcvMsg.append(rcvd);  
            }  
        };  
  
        if (SwingUtilities.isEventDispatchThread()) {  
            r.run();  
        } else {  
            try {  
                SwingUtilities.invokeAndWait(r);  
            } catch (InvocationTargetException | InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
        }
    }

public void log(String msg) {
    Date dNow = new Date();
    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
    String log = "<" + ft.format(dNow) + "> " + msg + "\n";
    showRcvMsg(log);
}
```

6.4.3 Classe RobotDesenhador

```
public class RobotDesenhador {  
    private RobotLegoEV3 robot;  
    private GUIServidor gui;  
    private String nome;  
    private boolean conectado;  
  
    public RobotDesenhador(String nome) {  
        this.nome = nome;  
        this.gui = new GUIServidor("Robot Desenhador");  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public GUIServidor getGUI() {  
        return gui;  
    }  
  
    public void desenhar(Mensagem msg) {  
        if (conectado) msg.executarComando(robot);  
        gui.log(msg.toString());  
    }  
  
    public void conectar() {  
        robot = new RobotLegoEV3();  
        conectado = robot.OpenEV3(nome);  
        if (conectado) {  
            String msg = "Robot " + nome + " conectado!";  
            gui.log(msg);  
            System.out.println(msg);  
        }  
    }  
  
    public void desconectar() {  
        robot.CloseEV3();  
        String msg = "Robot " + nome + " desconectado!";  
        gui.log(msg);  
        System.out.println(msg);  
        gui.dispose();  
    }  
  
    public boolean isConectado() {  
        return conectado;  
    }  
}
```

6.5 Gravador

6.5.1 Enumerado EstadoGravador

```
public enum EstadoGravador {  
    PARADO, GRAVAR, REPRODUZIR, LER, ESCREVER  
}
```

6.5.2 Classe GravarFormas

```
import java.io.*;
import java.util.ArrayList;
import java.util.concurrent.Semaphore;

import com.sun.jmx.snmp.Timestamp;

public class GravarFormas extends Thread {

    private String filename = "";
    private File file;
    private GUIGravarFormas guiGF;
    private ObjectOutputStream out;
    private ObjectInputStream in;
    private Timestamp tsAntes, tsAgora;
    private ArrayList<Object> arrMensagens;
    private EstadoGravador estadoGravador;
    private Semaphore smp;
    private boolean endApp;
    private boolean modoReproducao;
    private RobotDesenhador robotDesenhador;

    public GravarFormas(GUIGravarFormas guiGF) {
        this.guiGF = guiGF;
        this.endApp = false;
        this.modoReproducao = false;
        arrMensagens = new ArrayList<Object>();
        estadoGravador = EstadoGravador.PARADO;
        smp = new Semaphore(0);
    }

    public EstadoGravador getEstado() {
        return estadoGravador;
    }

    public boolean isModoReproducao() {
        return modoReproducao;
    }

    public void setEstado(EstadoGravador estado) {
        estadoGravador = estado;
        smp.release();
    }

    public void setRobot(RobotDesenhador robotDesenhador) {
        this.robotDesenhador = robotDesenhador;
    }

    public void novoFicheiro(String filename) {
        if (filename.equals(""))
            filename = "formas.txt";
        this.filename = filename;
        file = new File(filename);
    }

    public void adicionarMensagem(Mensagem m) {
        if (estadoGravador == EstadoGravador.GRAVAR) {
            tsAgora = new Timestamp(System.currentTimeMillis());
            long espera = tsAgora.getDate() - tsAntes.getDate();

            tsAntes = tsAgora;
            arrMensagens.add(m);
            guiGF.log("Guardado: " + m.toString());

            espera = espera == 0 ? 1 : espera;
            guiGF.log("Guardado: Espera = " + espera + "\n");
            arrMensagens.add((int) espera));
        }
    }
}
```

```

        }

    public void guardarMensagens() {
        try {
            if (out == null)
                out = new ObjectOutputStream(new FileOutputStream(file));
            while (arrMensagens.size() > 0) {
                out.writeObject(arrMensagens.remove(0));
            }
            out.writeObject(null);
            out.flush();
            out.close();
            guiGF.log("MENSAGENS ESCRITAS NO FICHEIRO!\n");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void lerMensagens() {
        try {
            if (in == null)
                in = new ObjectInputStream(new FileInputStream(filename));
            Object o;
            int count = 0;
            while ((o = in.readObject()) != null) {
                if (count % 2 == 0) {
                    Mensagem m = (Mensagem) o;
                    arrMensagens.add(m);
                    guiGF.log("Lido: " + m.toString());
                } else {
                    int espera = (int) o;
                    arrMensagens.add(espera);
                    guiGF.log("Lido: Espera: " + espera);
                }
                count++;
            }
            guiGF.log("ACABOU DE LER DO FICHEIRO!\n");
            in.close();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void reproduzirMensagens() {
        for (int i = 0; i < arrMensagens.size(); i++) {
            if (i % 2 == 0) {
                Mensagem m = (Mensagem) arrMensagens.get(i);
                if (robotDesenhador != null)
                    robotDesenhador.desenhar(m);
                guiGF.log("Reproduzido: " + m);
            } else {
                try {
                    int espera = (int) arrMensagens.get(i);
                    guiGF.log("Reproduzido: Espera: " + espera + "\n");
                    Thread.sleep(espera);

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        guiGF.log("ACABOU DE REPRODUZIR!\n");
    }

    public void setEndApp() {
        this.endApp = true;
        smp.release();
    }

    @Override
    public void run() {
        for (;;) {
            if (this.endApp)

```

```
        break;

    switch (estadoGravador) {
        case PARADO:
            try {
                smp.acquire();
                tsAntes = new Timestamp(System.currentTimeMillis());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            break;
        case GRAVAR:
            break;
        case ESCREVER:
            guardarMensagens();
            estadoGravador = EstadoGravador.PARADO;
            break;
        case LER:
            lerMensagens();
            modoReproducao = true;
            estadoGravador = EstadoGravador.PARADO;
            break;
        case REPRODUZIR:
            reproduzirMensagens();
            guiGF.ligarBtnReproducao();
            estadoGravador = EstadoGravador.PARADO;
            break;
    }
}
```

6.5.3 Classe GUIGravarFormas

```
import javax.swing.*;  
  
import java.lang.reflect.InvocationTargetException;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
public class GUIGravarFormas extends JFrame {  
    /**  
     *  
     */  
    private static final long serialVersionUID = -3798051427778409493L;  
    private JMenuItem menuAbrir;  
    private JMenuItem menuNovo;  
    private JTextArea txtAreaMsgGravadas;  
    private JToggleButton tglBtnGravacao;  
    private JButton btnReproducao;  
    private JButton btnLer;  
    private JButton btnEscrever;  
  
    private GravarFormas gravador;  
    private String filename = "";  
  
    public GUIGravarFormas() {  
        init();  
        myInit();  
    }  
  
    private void init() {  
        setTitle("Gravar Formas");  
        setResizable(false);  
        getContentPane().setLayout(null);  
  
        tglBtnGravacao = new JToggleButton("Iniciar/Parar Grava\u00e7\u00e3o");  
        tglBtnGravacao.setBounds(40, 30, 178, 34);  
        tglBtnGravacao.addActionListener(e -> handleGravacao());  
        getContentPane().add(tglBtnGravacao);  
  
        JScrollPane scrollPaneRcvMsg = new JScrollPane();  
        scrollPaneRcvMsg.setBounds(40, 171, 394, 189);  
        getContentPane().add(scrollPaneRcvMsg);  
  
        txtAreaMsgGravadas = new JTextArea();  
        scrollPaneRcvMsg.setViewportView(txtAreaMsgGravadas);  
        txtAreaMsgGravadas.setEditable(false);  
  
        JLabel mensagensGravadas = new JLabel("Logs");  
        mensagensGravadas.setBounds(40, 148, 164, 13);  
        getContentPane().add(mensagensGravadas);  
  
        btnReproducao = new JButton("Iniciar Reprodu\u00e7\u00e3o");  
        btnReproducao.addActionListener(e -> handleReproducao());  
        btnReproducao.setEnabled(false);  
        btnReproducao.setBounds(40, 87, 178, 34);  
        getContentPane().add(btnReproducao);  
  
        btnLer = new JButton("Ler do ficheiro");  
        btnLer.addActionListener(e -> handleLeitura());  
        btnLer.setEnabled(false);  
        btnLer.setBounds(256, 87, 178, 34);  
        getContentPane().add(btnLer);  
  
        btnEscrever = new JButton("Escrever no ficheiro");  
        btnEscrever.addActionListener(e -> handleEscrever());  
        btnEscrever.setEnabled(false);  
        btnEscrever.setBounds(256, 30, 178, 34);
```

```

getContentPane().add(btnEscrever);

setSize(493, 450);

JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);

JMenu menuFicheiro = new JMenu("Ficheiro");
menuBar.add(menuFicheiro);

menuAbrir = new JMenuItem("Abrir");
menuAbrir.addActionListener(e -> handleAbrir());
menuFicheiro.add(menuAbrir);

menuNovo = new JMenuItem("Novo");
menuNovo.addActionListener(e -> handleNovo());
menuFicheiro.add(menuNovo);
setVisible(false);
setResizable(false);
}

private void myInit() {
    gravador = new GravarFormas(this);
    gravador.start();
    tglBtnGravacao.setEnabled(false);
}

private void handleAbrir() {
    JFileChooser chooser = new JFileChooser();
    int returnVal = chooser.showOpenDialog(null);
    if (returnVal == JFileChooser.APPROVE_OPTION) {
        filename = chooser.getSelectedFile().getAbsolutePath();
        gravador.novoFicheiro(filename);
        tglBtnGravacao.setEnabled(true);
        btnReproducao.setEnabled(false);
        btnEscrever.setEnabled(false);
        btnLer.setEnabled(true);
    }
}

private void handleNovo() {
    filename = JOptionPane.showInputDialog("Introduza o nome e a extens\u00e3o do ficheiro.");
    if (filename == null) return;
    gravador.novoFicheiro(filename);
    tglBtnGravacao.setEnabled(true);
}

private void handleGravacao() {
    if (tglBtnGravacao.isSelected()) {
        gravador.setEstado(EstadoGravador.GRAVAR);
        btnLer.setEnabled(false);
        btnEscrever.setEnabled(false);
        btnReproducao.setEnabled(false);
    } else {
        gravador.setEstado(EstadoGravador.PARADO);
        btnEscrever.setEnabled(true);
    }
}

private void handleEscrever() {
    gravador.setEstado(EstadoGravador.ESCREVER);
    tglBtnGravacao.setEnabled(false);
    btnEscrever.setEnabled(false);
    btnLer.setEnabled(true);
    btnReproducao.setEnabled(false);
}

private void handleLeitura() {
    gravador.setEstado(EstadoGravador.LER);
    btnLer.setEnabled(false);
    btnReproducao.setEnabled(true);
    btnEscrever.setEnabled(false);
    tglBtnGravacao.setEnabled(false);
}

```

```

private void handleReproducao() {
    gravador.setEstado(EstadoGravador.REPRODUZIR);
    btnReproducao.setEnabled(false);
}

public void ligarBtnReproducao() {
    btnReproducao.setEnabled(true);
}

public void log(String msg) {
    Date dNow = new Date();
    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
    String log = "<" + ft.format(dNow) + "> " + msg + "\n";
    showRcvMsg(log);
}

public void showRcvMsg(String rcvd) {
    Runnable r = new Runnable() {
        @Override
        public void run() {
            txtAreaMsgGravadas.append(rcvd);
        }
    };
    if (SwingUtilities.isEventDispatchThread()) {
        r.run();
    } else {
        try {
            SwingUtilities.invokeAndWait(r);
        } catch (InvocationTargetException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public GravarFormas getGravador() {
    return gravador;
}

public void handleBotaoReproduzir(boolean ativo) {
    btnReproducao.setEnabled(ativo);
}

public void mostrarGUI(boolean value) {
    setVisible(value);
}
}

```

6.6 GUI Principal

6.6.1 Interface IGUI

```
public interface IGUI {  
    public void acabeiDesenho();  
  
    public boolean acabouDesenho();  
  
    public int getDist();  
}
```

6.6.2 Classe GUILaunch

```
import java.awt.EventQueue;
import java.lang.reflect.InvocationTargetException;

public class GUILaunch {

    public static void main(String[] args) {
        Launch();
    }

    static class MyRunnable implements Runnable {
        private GUI window;

        public GUI getWindow() {
            return this.window;
        }

        public void run() {
            this.window = new GUI();
        }
    }

    static void launch() {
        MyRunnable r = new MyRunnable();

        try {
            EventQueue.invokeAndWait(r);
        } catch (InvocationTargetException | InterruptedException e) {
            e.printStackTrace();
        }

        r.getWindow().run();
    }
}
```

6.6.3 Classe GUI

```
import Buffer.BufferCircularGUI;
import Buffer.BufferCircularMonitores;
import Buffer.IBufferCircular;
import Buffer.IBufferCircularGUI;

import java.awt.Color;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.concurrent.Semaphore;
import javax.swing.border.EtchedBorder;
public class GUI extends JFrame implements IGUI {
    /**
     *
     */
    private static final long serialVersionUID = 8502496328362365351L;
    private JTextField textFieldNomeRobot;
    private JCheckBox chckbxIniciaRobot;
    private JButton btnCriaBufferEServidor;
    private JButton btnGUIDesenhadador;
    private JButton btnQuadrado;
    private JButton btnCirculo;
    private ButtonGroup group;
    private JRadioButton rdbtnCurvarEsq;
    private JRadioButton rdbtnCurvarDireita;
    private JSpinner spinnerLado;
    private JSpinner spinnerRaio;
    private JSpinner spinnerDeltaAng;

    private ServidorDoRobot servidorDoRobot;
    private RobotDesenhadador robotDesenhadador;
    private ClienteDoRobot clienteDoRobot;
    private GUIServidor guiRD;
    private GUIGravarFormas guiSpy;
    private DesenharQuadrado quadrado;
    private DesenharCirculo circulo;
    private EspacarFormasGeometricas espacamento;

    private boolean endApp = false;
    private boolean toggleGUISpy = false;
    private boolean ladoEsq;
    private boolean acabouDesenho;
    private IBufferCircularGUI guiBuffer;
    private IBufferCircular<Mensagem> buffer;
    private final static int MAX_BUFFER_SIZE = 16;
    private int dist = 0;

    public GUI() {
        initialize();
        myInit();
    }

    private void initialize() {
        setTitle("Trabalho 2");
        setResizable(false);
        getContentPane().setLayout(null);

        btnCriaBufferEServidor = new JButton("Criar buffer e servidor");
        btnCriaBufferEServidor.addActionListener(e -> {
            handleCriarBuffer();
            handleCreateServer();
        });
        btnCriaBufferEServidor.setBounds(282, 203, 290, 29);
        getContentPane().add(btnCriaBufferEServidor);

        btnGUIDesenhadador = new JButton("Mostrar GUI Robot Desenhadador");
    }
}
```

```

btnGUIDesenador.addActionListener(e -> {
    handleGUIDesenador();
});
btnGUIDesenador.setBounds(282, 239, 290, 29);
getContentPane().add(btnGUIDesenador);

JLabel lblNomeRobot = new JLabel("Nome do Robot");
lblNomeRobot.setBounds(282, 9, 125, 16);
getContentPane().add(lblNomeRobot);

textFieldNomeRobot = new JTextField("EV1");
textFieldNomeRobot.setBounds(282, 35, 290, 26);
getContentPane().add(textFieldNomeRobot);
textFieldNomeRobot.setColumns(10);

 JPanel panelComportamentos = new JPanel();
panelComportamentos.setBorder(new TitledBorder(new EtchedBorder(EtchedBorder.LOWERED, null,
null), "Comportamentos", TitledBorder.LEADING, TitledBorder.TOP, null, new Color(0, 0, 0)));
panelComportamentos.setBounds(16, 11, 249, 260);
getContentPane().add(panelComportamentos);
panelComportamentos.setLayout(null);

btnQuadrado = new JButton("Desenhar Quadrado");
btnQuadrado.setBounds(6, 16, 237, 53);
panelComportamentos.add(btnQuadrado);
btnQuadrado.addActionListener(e -> handleDesenharQuadrado());

btnCirculo = new JButton("Desenhar Circulo");
btnCirculo.setBounds(6, 69, 237, 53);
btnCirculo.addActionListener(e -> handleDesenharCirculo());
panelComportamentos.add(btnCirculo);

rdbtnCurvarEsq = new JRadioButton("Curvar Esquerda");
rdbtnCurvarEsq.setBounds(6, 122, 131, 43);
panelComportamentos.add(rdbtnCurvarEsq);

rdbtnCurvarDireita = new JRadioButton("Curvar Direita");
rdbtnCurvarDireita.setBounds(139, 122, 104, 43);
panelComportamentos.add(rdbtnCurvarDireita);

spinnerLado = new JSpinner();
spinnerLado.setModel(new SpinnerNumberModel(30, 20, 60, 1));
spinnerLado.setBounds(55, 180, 51, 20);
panelComportamentos.add(spinnerLado);

spinnerRaio = new JSpinner();
spinnerRaio.setModel(new SpinnerNumberModel(35, 0, 360, 1));
spinnerRaio.setBounds(142, 180, 51, 20);
panelComportamentos.add(spinnerRaio);

JLabel lblLado = new JLabel("Lado");
lblLado.setHorizontalAlignment(SwingConstants.CENTER);
lblLado.setBounds(55, 161, 51, 17);
panelComportamentos.add(lblLado);

JLabel lblRaio = new JLabel("Raio");
lblRaio.setHorizontalAlignment(SwingConstants.CENTER);
lblRaio.setBounds(142, 161, 51, 17);
panelComportamentos.add(lblRaio);

JLabel ajusteAnguloTxt = new JLabel("Ajustar angulo");
ajusteAnguloTxt.setHorizontalAlignment(SwingConstants.CENTER);
ajusteAnguloTxt.setBounds(6, 224, 104, 17);
panelComportamentos.add(ajusteAnguloTxt);

spinnerDeltaAng = new JSpinner();
spinnerDeltaAng.setModel(new SpinnerNumberModel(0, -180, 180, 1));
spinnerDeltaAng.setBounds(109, 219, 40, 26);
panelComportamentos.add(spinnerDeltaAng);

chkbxIniciaRobot = new JCheckBox("Conectar robot");
chkbxIniciaRobot.addActionListener(e -> handleIniciarRobot());
chkbxIniciaRobot.setBounds(282, 67, 128, 23);
getContentPane().add(chkbxIniciaRobot);

```

```

        JButton btnSpyrobot = new JButton("Gravar Figuras");
        btnSpyrobot.addActionListener(e -> handleAtivarEspiao());
        btnSpyrobot.setBounds(282, 166, 290, 29);
        getContentPane().add(btnSpyrobot);

        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        setSize(600, 332);
        setVisible(true);
    }

    private void myInit() {
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                // Ask for confirmation before terminating the program.
                int option = JOptionPane.showConfirmDialog(null, "De certeza que quer fechar a
aplicação?", "Confirmação de fecho", JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
                if (option == JOptionPane.YES_OPTION) {
                    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
                    handleClosing();
                }
            }
        });
    }

    // Group the radio buttons.
    group = new ButtonGroup();
    rdbtnCurvarEsq.setSelected(true);
    rdbtnCurvarEsq.setActionCommand("0");
    rdbtnCurvarDireita.setActionCommand("1");
    group.add(rdbtnCurvarEsq);
    group.add(rdbtnCurvarDireita);
    btnGUIDesenador.setEnabled(false);

    guiSpy = new GUIGravarFormas();
    clienteDoRobot = new ClienteDoRobot(buffer, guiSpy.getGravador());
    criarComportamentos();
}

private void criarComportamentos() {
    quadrado = new DesenharQuadrado(true, clienteDoRobot, this);
    circulo = new DesenharCirculo(true, clienteDoRobot, this);
    espacamento = new EspacarFormasGeometricas(dist, clienteDoRobot, this);

    quadrado.start();
    circulo.start();
    espacamento.start();
}

public void adicionarQuadrado() {
    quadrado.setLadoEsq(ladoEsq);
    quadrado.setAngulo(90 + (int) spinnerDeltaAng.getValue());
    if (clienteDoRobot.getUltimoComportamento() != null) {
        adicionarEspacamento(quadrado);
    } else {
        quadrado.ativar(null);
    }

    clienteDoRobot.setUltimoComportamento(quadrado);
    acabouDesenho = false;
}

public void adicionarCirculo() {
    circulo.setLadoEsq(ladoEsq);
    circulo.setAngulo(360 + (int) spinnerDeltaAng.getValue());
    if (clienteDoRobot.getUltimoComportamento() != null) {
        adicionarEspacamento(circulo);
    } else {
        circulo.ativar(null);
    }
    clienteDoRobot.setUltimoComportamento(circulo);
    acabouDesenho = false;
}

```

```

public void adicionarEspacamento(Comportamento c) {
    try {
        espacamento.setDistancia(clienteDoRobot.getUltimoComportamento().getDistancia());
        espacamento.ativar(c);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void handleCriarBuffer() {
    if (textFieldNomeRobot.getText().trim().equals("")) {
        String msg = "Tem de indicar o nome do robot antes de criar o servidor.";
        JOptionPane.showMessageDialog(null, msg, "Erro", JOptionPane.ERROR_MESSAGE);
        return;
    }
    try {
        guiBuffer = new BufferCircularGUI(MAX_BUFFER_SIZE);
        buffer = new BufferCircularMonitores<>(MAX_BUFFER_SIZE, guiBuffer);
        clienteDoRobot.setBuffer(buffer);
    } catch (Exception e) {
        e.printStackTrace();
        if (this.guiBuffer != null) {
            this.guiBuffer.dispose();
        }
    }
}

private void handleCreateServer() {
    if (textFieldNomeRobot.getText().trim().equals(""))
        return;
    servidorDoRobot = new ServidorDoRobot(buffer);
    servidorDoRobot.start();
    btnCriaBufferEServidor.setEnabled(false);
    btnGUIDesenhador.setEnabled(true);
}

private void handleIniciarRobot() {
    if (buffer == null || servidorDoRobot == null) {
        String msg = "Tem de criar o buffer e o servidor antes de iniciar o robot.";
        JOptionPane.showMessageDialog(null, msg, "Erro", JOptionPane.ERROR_MESSAGE);
        chckbxIniciaRobot.setSelected(false);
        return;
    }

    if (textFieldNomeRobot.getText().trim().equals("")) {
        String msg = "Tem de indicar o nome do robot antes de conectar.";
        JOptionPane.showMessageDialog(null, msg, "Erro", JOptionPane.ERROR_MESSAGE);
        chckbxIniciaRobot.setSelected(false);
        return;
    }

    if (robotDesenhador == null) {
        robotDesenhador = new RobotDesenhador(textFieldNomeRobot.getText().trim());
        servidorDoRobot.setRobotDesenhador(robotDesenhador);
        guiSpy.getGravador().setRobot(robotDesenhador);
    }

    robotDesenhador.setNome(textFieldNomeRobot.getText().trim());

    if (chckbxIniciaRobot.isSelected()) {
        robotDesenhador.conectar();
        guiSpy.getGravador().setRobot(robotDesenhador);
    } else {
        robotDesenhador.desconectar();
    }
}

private void handleClosing() {
    if (robotDesenhador != null && robotDesenhador.isConectado())
        robotDesenhador.desconectar();
    if (guiRD != null)
        guiRD.dispose();
    if (guiSpy != null) {
        guiSpy.dispose();
    }
}

```

```

        guiSpy.getGravador().setEndApp();
    }
    if (guiBuffer != null)
        guiBuffer.dispose();
    if (servidorDoRobot != null)
        servidorDoRobot.setEnd();
    if (circulo != null)
        circulo.setEnd();
    if (quadrado != null)
        quadrado.setEnd();
    if (espacamento != null)
        espacamento.setEnd();
    endApp = true;
}

private void handleDesenharCirculo() {
    if (clienteDoRobot == null)
        return;
    btnCirculo.setEnabled(false);
    btnQuadrado.setEnabled(false);
    setLadoEsq(rdbtnCurvarEsq.isSelected());
    setDist((int) spinnerRaio.getValue());
    adicionarCirculo();
}

private void handleDesenharQuadrado() {
    if (clienteDoRobot == null)
        return;
    btnCirculo.setEnabled(false);
    btnQuadrado.setEnabled(false);
    setLadoEsq(rdbtnCurvarEsq.isSelected());
    setDist((int) spinnerLado.getValue());
    adicionarQuadrado();
}

private void handleGUIDesenhador() {
    if (robotDesenhador == null) {
        robotDesenhador = new RobotDesenhador(textFieldNomeRobot.getText().trim());
        servidorDoRobot.setRobotDesenhador(robotDesenhador);
    }
    robotDesenhador.getGUI().setVisible(!toggleGUISpy);
}

private void handleAtivarEspiao() {
    guiSpy.mostrarGUI(!toggleGUISpy);
}

public void acabeiDesenho() {
    acabouDesenho = true;
}

public boolean acabouDesenho() {
    return acabouDesenho;
    if (guiSpy != null && guiSpy.getGravador().isModoReproducao())
        guiSpy.handleBotaoReproduzir(true);
}

public void setLadoEsq(boolean ladoEsq) {
    this.ladoEsq = ladoEsq;
}

public void setDist(int dist) {
    this.dist = dist;
}

public int getDist() {
    return dist;
}

public void run() {
    for (;;) {
        try {
            if (endApp)
                break;
        }
    }
}

```

```
        Thread.sleep(100);
        if (clienteDoRobot != null && acabouDesenho()
        && (guiSpy != null && guiSpy.getGravador().getEstado() != EstadoGravador.REPRODUZIR)) {
            btnQuadrado.setEnabled(true);
            btnCirculo.setEnabled(true);
            continue;
        }
        btnQuadrado.setEnabled(false);
        btnCirculo.setEnabled(false);
        if (guiSpy != null)
            guiSpy.handleBotaoReproduzir(false);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```