



ADDETC – Área Departamental de Engenharia Eletrónica e Telecomunicações
e de Computadores

LEIM -Licenciatura Engenharia informática e multimédia

Fundamentos de Sistemas Operativos

Trabalho prático 2

Turma:

LEIM-31D-b

Trabalho realizado por:

Miguel Silvestre N°45101

Miguel Távora N°45102

Docente:

Carlos Carvalho

Data: 25/01/2021

Índice

1. INTRODUÇÃO.....	1
2. DESENVOLVIMENTO	3
2.1 ENQUADRAMENTO TEÓRICO.....	3
1.1 DEFINIÇÃO DE PROCESSOS LEVES	3
1.2 NECESSIDADE DE SINCRONIZAÇÃO	3
1.3 EXCLUSÃO MÚTUA	4
1.4 SEMÁFOROS.....	4
1.5 MONITORES	5
1.6 SISTEMA DE FICHEIROS	5
2.2 IMPLEMENTAÇÃO TRABALHO	6
1.1 CLIENTE.....	6
1.2 SINCRONIA NOS CLIENTES	8
1.3 IMPLEMENTAÇÃO DO CLIENTE	9
1.4 IMPLEMENTAÇÃO DOS COMPORTAMENTOS	11
2.1 BUFFER CIRCULAR	15
2.2 FORMATO DAS MENSAGENS	18
3.1 SERVIDOR.....	21
3.2 ROBÔ DESENHADOR	22
4.1 MANIPULAÇÃO DE FICHEIROS	24
4.2 INTERFACE GRÁFICA DA MANIPULAÇÃO DE FICHEIROS.....	27
5.1 APLICAÇÃO	29
5.2 INTERFACE GRÁFICA DA APLICAÇÃO	31
6.1 FUNCIONAMENTO GLOBAL DA APLICAÇÃO	33
3. CONCLUSÕES.....	37
4. BIBLIOGRAFIA	39
5. ANEXO.....	41

Índice ilustrações

Figura 1 - Diagrama classes dos comportamentos dos comportamentos.....	7
Figura 2 - Implementação da classe Draw	8
Figura 3 - Excerto da classe RobotClient.....	9
Figura 4 - Diagrama de classes entre os comportamentos e o cliente	10
Figura 5 - Método construtor da classe DrawSpaceForms	12
Figura 6 - Método draw da classe DrawSpaceForms	13
Figura 7 - Diagrama classes do cliente completo	14
Figura 8 - Método de escrita do buffer	16
Figura 9 - Método leitura do buffer	16
Figura 10 - Diagrama de classes do BufferCircular	17
Figura 11 - Formato das mensagens	18
Figura 12 - Implementação da mensagem reta.....	19
Figura 13 - Diagrama de classes das mensagens.....	20
Figura 14 - Método convertMessage da classe RobotServer	21
Figura 15 - Excerto da classe RobotDesigner	22
Figura 16 - Diagrama classes servidor com robô desenhador.....	23
Figura 17 - Máquina estados da classe StoreFile.....	25
Figura 18 - Exemplo de mensagem escrita no ficheiro.....	26
Figura 19 - Interface gráfica a gravar.....	27
Figura 20 - Interface gráfica a executar	27
Figura 21 - Diagrama classes da manipulação de ficheiros	28
Figura 22 - Máquina estados da aplicação	30
Figura 23 - Interface gráfica da aplicação	31
Figura 24 - Diagrama classes da aplicação	32
Figura 25 - Diagrama estados para a aplicação	34
Figura 26 - Diagrama classes sem escrita no ficheiro.....	35
Figura 27 - Diagrama classes total.....	36

1. Introdução

No segundo trabalho prático de Fundamentos de Sistemas Operativos é pretendido desenvolver uma aplicação multitarefa baseado no modelo Produtor-Consumidor. O desenvolvimento da comunicação e sincronização entre tarefas são um fator fundamental para o desenvolvimento da aplicação.

A aplicação consiste no desenvolvimento de um controlo automático sobre um robot desenhador no espaço bidimensional. Os desenhos consistem essencialmente em formas geométricas. Desta forma o robô irá desenhar quadrados, círculos e também deve espaçar uma determinada distância, todos estes desenhos são comportamentos que por sua vez também são clientes. A distância do espaçar formas serve para não sobrepor os desenhos realizados anteriormente com os que irão ser posteriormente desenhados.

O modo de funcionamento da aplicação é baseado no modelo Produtor-Consumidor, quer isto dizer que existe um produtor, que será referido como cliente, responsável por gerar as mensagens e enviar para um mecanismo que guarda as mensagens. O consumidor, que corresponde ao servidor, é o responsável por ler as mensagens guardadas pelo cliente e executar essas mesmas mensagens.

O mecanismo responsável por armazenar as mensagens criadas pelo cliente e lidas pelo servidor é um mecanismo designado por buffer circular. Este buffer é responsável por deixar o cliente escrever no seu interior e por deixar o servidor ler o que o cliente escreveu. Para que todo este mecanismo funcione é necessário que este recurso garanta sincronia entre as diferentes tarefas no acesso ao recurso, isto porque é um recurso partilhado.

Cada cliente possui uma interface gráfica que vai exibindo as mensagens que envia para o buffer. O servidor também possui uma interface gráfica que vai exibindo os comandos que lê do buffer e simula o robô físico.

O cliente e o servidor conseguem comunicar por intermédio do buffer que utiliza um tipo específico de mensagens conhecidas por ambos os comportamentos.

Por fim será utilizado o sistema de ficheiros para fazer a escrita de comandos previamente executados. Após os comando serem escritos, é possível ler os comandos e executá-los no robô.

2. Desenvolvimento

2.1 Enquadramento teórico

1.1 Definição de processos leves

Os processos leves, também designados tarefas, são executados num único computador e são partes de um único processo designado por processo-pai que as lança e executa. As tarefas cooperam dentro de um processo-pai onde cada uma realiza um comportamento pretendido. As tarefas são realizadas em pseudo-parallelismo num processador de um core e parallelismo puro num processo múltiplos-cores.

As tarefas leves concorrem e partilham recursos do computador e partilham dados do processo-pai. A implementação de uma tarefa dentro de um processo possui uma estrutura de dados e código independente das outras tarefas.

Tarefas em Java podem ser implementados através de uma classe Java por derivação de Thread ou implementação de Runnable lançada posteriormente como uma Thread.

1.2 Necessidade de sincronização

A comunicação entre tarefas exige sincronização, isto porque o tempo de processamento de cada tarefa e a sua execução é imprevisível num computador, quer isto dizer que é executado assincronamente. Para que a aplicação funcione numa sequência lógica é necessário haver pontos sincronismo nas atividades a executar.

Quando duas ou mais tarefas acedem a recursos não partilháveis, estes devem ser protegidos do acesso simultâneo das tarefas. Para prevenir essa situação é utilizado o mecanismo de exclusão mútua.

1.3 Exclusão mútua

Recursos não partilháveis só podem ser acedidos por uma tarefa de cada vez, sendo que o programa desenvolvido é que deve garantir essa exclusão.

Uma forma de garantir exclusão mútua é através de uma variável pela instrução *getAndSet* que existe na Java Virtual Machine. Esta instrução permite aceder e alterar o valor lógico da variável numa única instrução *hardware* e desta forma apenas uma variável executa as instruções que se seguem de cada vez. Contudo esta metodologia está sistematicamente a verificar o valor da variável consumindo CPU.

1.4 Semáforos

Os semáforos são uma entidade que permite garantir exclusão mútua no acesso a um recurso, ficando desta forma a tarefa bloqueada sem gastar CPU.

As operações são feitas baseadas num número inteiro superior ou igual a 0, onde é possível incrementar o valor quando se dá *signal* ou em Java *release()* de um semáforo. A operação de decrementar um número inteiro quando este é superior a 1 é feito pelo *wait* em Java *acquire()* do semáforo. Quando um semáforo é iniciado com o valor 1 e é feito um *acquire()* o valor é reduzido para 0, na segunda execução do *acquire()* como o valor é 0 este fica bloqueado até que alguém faça *release()* e liberte uma permissão. A partilha do semáforo é definida pelo valor inicial do semaforo, se o valor for apenas 1 somente uma tarefa pode aceder ao recurso de cada vez.

Quando processos ficam à espera de um recurso e este não é libertado por falha algoritmica, ou outra falha, ocorre uma situação designada de *deadlock*. Um *deadlock* pode ocorrer por uma sequência incorreta de execução de *acquires* ou *releases* sobre semáforos.

1.5 Monitores

Os monitores são uma alternativa aos semáforos, onde o seu objetivo é executar métodos em exclusividade por múltiplos processos sem a complexidade inerente dos semaforos, que são fontes de erros.

Monitores Lampson – processo executa o acordar (`notify()`) e continua a sua execução, os processos desbloqueados são colocados na fila interna no estado “pronto para execução” que são executados no terminar da tarefa que fez o *notify*.

Os monitores em Java tem uma *flag lock* através do uso direto *synchronized* antes do nome do método. Cada objeto Java tem associado um monitor implementado a partir duma *flag lock*.

1.6 Sistema de ficheiros

O sistemas de ficheiros é um programa que é parte do sistema operativo que gera a entidade ficheiros.

Um ficheiro é guardado e manipulado pelo sistema de ficheiros.

O suporte físico é um dispositivo físico que permite armazenamento permanente de bits, onde a sua organização é feita em blocos de dimensão fixa.

Manipulação de ficheiro em Java é feito por `Input` e `OutputStream`. Uma *Stream* é uma entidade lógica que se comporta como um acesso a ficheiros ou a dispositivos de *Input/Output*.

2.2 Implementação trabalho

A implementação da aplicação foi baseada no modelo produtor-consumidor. Esta implementação tem como objeto existir uma entidade que produz alguma informação e alguém que irá ler essa informação. No caso do trabalho prático a ideia é existir um cliente, que corresponde ao produtor, e um servidor, que corresponde ao consumidor. Para que exista a comunicação entre cliente e servidor existe um ponto de acesso mútuo para ambas as entidades que é designado de buffer circular. Este buffer é onde fica armazeada a informação gerada pelo cliente, informação essa que será depois lida pelo servidor. As quatro grandes entidades da aplicação são o cliente, o buffer circular, o servidor e o manipulador de ficheiros. Estas entidades são todas agrupadas numa entidade que é a aplicação.

1.1 Cliente

O cliente como foi dito anteriormente é a entidade responsável por gerar informação. Neste caso a informação são nomeadamente mensagens para o robô desenhador executar, estas mensagens são colocadas no buffer através de um método de escrita.

O cliente é formado por três tipos de clientes cada um distinto dos restantes, onde os clientes são nomeadamente as classes `DrawCircle`, `DrawSquare` e `DrawSpaceForms`. As classes por serem todos do tipo cliente possuem comportamentos iguais na forma das suas atividades, mas diferentes no conteúdo da atividade. Quer isto dizer que ambas as classes estão sempre à espera de receber uma mensagem para enviar, mas as mensagens que enviam são distintas entre elas. Por isso todas as classes estendem de uma classe que implementa o comportamento comum das classes, por sua vez a classe é abstrata e obriga a implementação do método designado por `draw()`, método esse que será então específico de cada classe. Esta classe abstrata implementa a interface `Runnable` visto que todos os clientes são tarefas e possui o método `run()` já implementado.

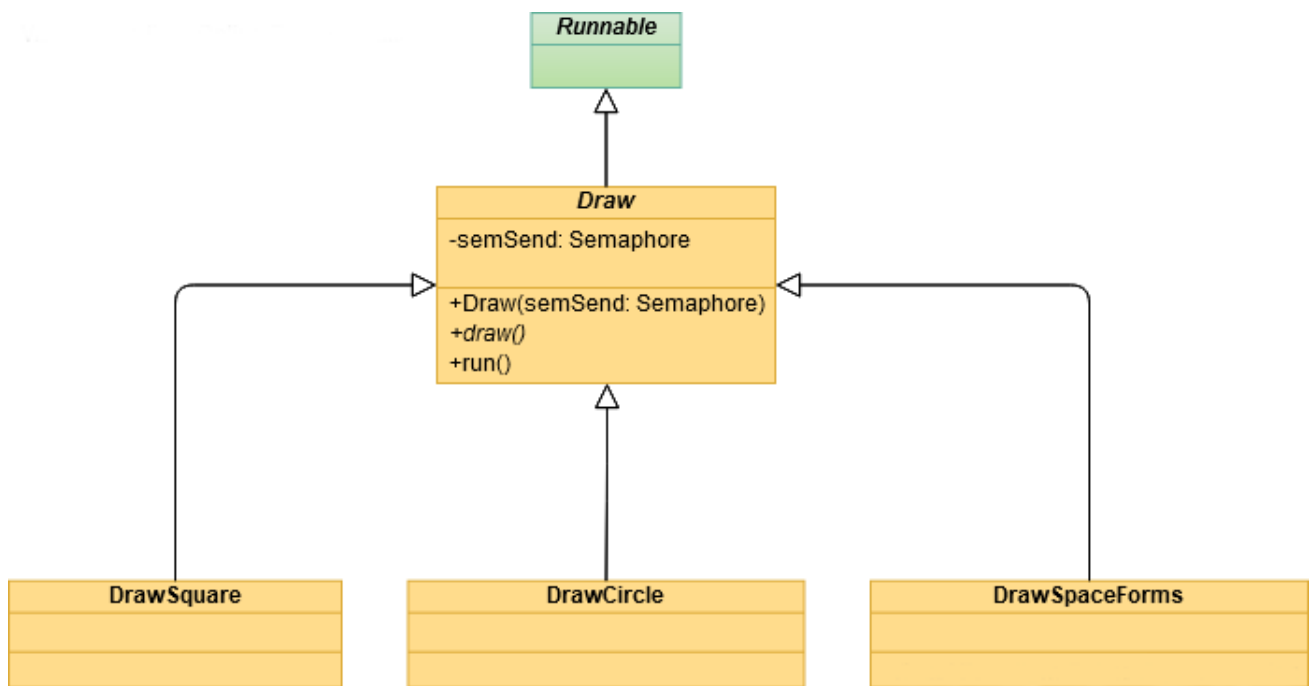


Figura 1 - Diagrama classes dos comportamentos dos comportamentos

1.2 Sincronia nos clientes

Todos os clientes são tarefas independentes e como foi dito anteriormente a execução de uma tarefa é imprevisível num sistema operativo. Desta forma a geração de comandos no cliente será feita assincronamente. O objetivo pretendido é desenhar formas geométricas sempre com um espaçamento antes da forma geométrica, contudo com assincronia na execução do código o robô poderia desenhar primeiro a forma geométrica e só depois espaçar as formas, não correspondendo ao objetivo pretendido. Desta forma é necessário garantir sincronia entre tarefas, esta sincronia foi obtida através de semáforos, como se pode observar na figura que se segue:

```
public abstract class Draw implements Runnable{

    private Semaphore semSend;

    public Draw(Semaphore semSend) {
        this.semSend = semSend;
    }

    public abstract void draw();

    //método que fica a correr até se clicar numa forma geometrica
    //que liberta uma permissão e então pode desenhar
    @Override
    public void run() {
        for (;;) {
            try {
                semSend.acquire();
                draw();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figura 2 - Implementação da classe Draw

O método run é constituído de um acquire() sobre um semáforo criado com zero permissões criado na aplicação, ficando assim bloqueado até ser feito um release(). Este release() é feito quando o utilizador clica num botão para gerar uma forma geométrica, desta maneira é feito o método draw() de cada comportamento que estende da classe Draw.

1.3 Implementação do cliente

Cada comportamento utiliza uma classe designada RobotClient criada na aplicação e passada a referência aos comportamentos. Esta classe é responsável essencialmente por meter mensagens no buffer pelo método sendMessage() e também disponibiliza métodos para o comportamento DrawSpaceForms para saber a distância que deve percorrer para não sobrepor a forma anterior. Para isso disponibiliza dois métodos de espaçamento designados de spaceFormsCircle() e spaceFormsSquare(), visto que o espaçar formas não tem acesso direto ao buffer.

```
private BufferCircular buffer;

public RobotClient(BufferCircular buffer) {
    this.buffer = buffer;
}

//envia mensagens para o buffer
public void sendMessage(Message m) {
    buffer.write(m);
}

// buscar o tamanho das formas anteriores para retornar o tamanho do espaçamento do circulo
public int spaceFormsCircle(int radius) {
    Message m = buffer.readLastMessage(2);
    Message m2 = buffer.readLastMessage(4);

    if (m != null) {
        if ((m.getId() == LeftCurve.LEFT_CURVE || m.getId() == RightCurve.RIGHT_CURVE) && m.
            return m.getRay() + INCREASE_SPACE + radius;
    }

    if (m2 != null) {
        if (m2.getId() == Line.LINE)
            return m2.getDistance() + INCREASE_SPACE + radius;
    }
    return DEFAULT_SPACE;
}
```

Figura 3 - Excerto da classe RobotClient

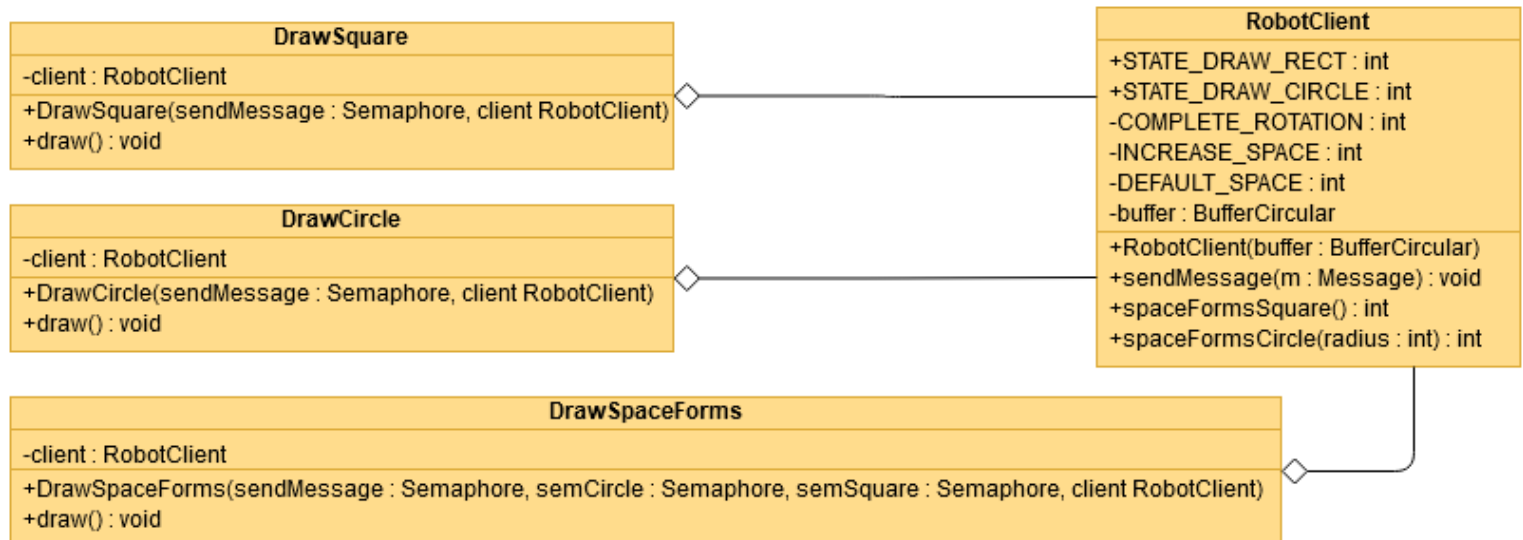


Figura 4 - Diagrama de classes entre os comportamentos e o cliente

1.4 Implementação dos comportamentos

Cada comportamento realiza uma funcionalidade diferente na aplicação. Nesse sentido cada classe possui uma implementação distinta do método `draw()`, uma interface gráfica para exibir comandos gerados pelo comportamento e uma referência para o cliente criado na aplicação.

O método `draw()` do desenhar quadrado será então composto por 12 comandos no total sendo eles primeiro uma reta, de seguida um parar, posteriormente uma curva á esquerda de 90° finalizando com um parar e isto repete-se quatro vezes, obtendo assim um quadrado. O método `draw()` do circulo é constituído de um círculo de 360° seguindo de um parar. O método `draw()` do espaçar formas é constituído por uma reta e um parar.

Para que os comportamentos consigam saber qual a distância que deve ser percorrida eles possuem um atributo do tipo inteiro e um método público para fazer *set* do valor da variável. Como a aplicação é quem faz o acesso ao valor escrito pelo utilizador na interface gráfica, o seu valor é posteriormente passado pelo método *set* para os comportamentos. A partir destas informações são geradas instâncias de classes correspondentes ás mensagens, que são depois enviadas através do cliente para o buffer.

Contudo como todos os comportamentos são lançados como Threads estes correm simultâneamente não havendo uma possível previsão dos resultados obtidos, desta forma foram utilizados semáforos para garantir a execução sequencial pretendida das instruções. Para obter esses resultados foram utilizados três semáforos um para cada comportamento. A classe Draw possui o método `run()` que inicia por realizar um `acquire()` ficando bloqueado. O utilizador quando seleciona um botão para realizar uma forma geométrica é libertada uma permissão na classe `DrawSpaceForms`. A classe `DrawSpaceForms` recebe também uma referência para os semáforos dos outros comportamentos, nomeadamente o desenhar quadrado e desenhar circulo. A partir das referências quando acaba a sua execução liberta uma permissão ou para a classe `DrawSquare` ou `DrawCircle` dependendo se o utilizador selecionou um quadrado ou um circulo. Desta maneira é garantido que existe uma sequência na ordem de execução das instruções onde o robô inicia sempre com o espaçamento de formas e de seguida executa a

forma geométrica pretendida.

Durante o lançamento das mensagens existe também um tempo de espera associado a cada mensagem que corresponde ao tempo de comunicação das mensagens por bluetooth, ou o tempo que o robô demora a realizar o comando. Assim é garantido que o robô recebe todas as mensagens enviadas pelo cliente e nenhuma se perde na execução das instruções.

Enquanto o programa está a desenhar uma forma geométrica o utilizador pode seleccionar outra forma geométrica. Caso isso aconteça a execução realiza-se logo após o robô desenhar a primeira forma geométrica. Contudo se o utilizador seleccionar mais do que uma a ordem de execução, por ser assíncrona, pode não ser feita pela ordem que o utilizador clicou nos botões, mas não é perdida nenhuma informação.

```
//para garantir que só correm após o espaçar formas
private Semaphore semCircle;
private Semaphore semSquare;

public DrawSpaceForms(Semaphore sendMessage, Semaphore
    super(sendMessage);
    this.client = client;
    this.semCircle = semCircle;
    this.semSquare = semSquare;
    gui = new DesignerGui("Espaçar formas");
    gui.setVisible();
}
```

Figura 5 - Método construtor da classe DrawSpaceForms


```
@Override
public void draw() {
    Message spaceForms;

    try {
        int space = 0;
        if (flag == SQUARE)
            space = client.spaceFormsSquare();

        else if (flag == CIRCLE)
            space = client.spaceFormsCircle(radius);

        spaceForms = new Line(space);
        client.sendMessage(spaceForms);
        gui.setTextToTextArea(gui.getTextFromTextArea() + "Reta: " + space + "\n");
        Thread.sleep(sleepTime(space));

        Message stop = new Stop();
        client.sendMessage(stop);
        gui.setTextToTextArea(gui.getTextFromTextArea() + "Parar\n\n");
        Thread.sleep(TIME_SLEEP_STOP);

        //libertar permissão quadrado ou circulo
        releaseForm(flag);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Figura 6 - Método draw da classe DrawSpaceForms

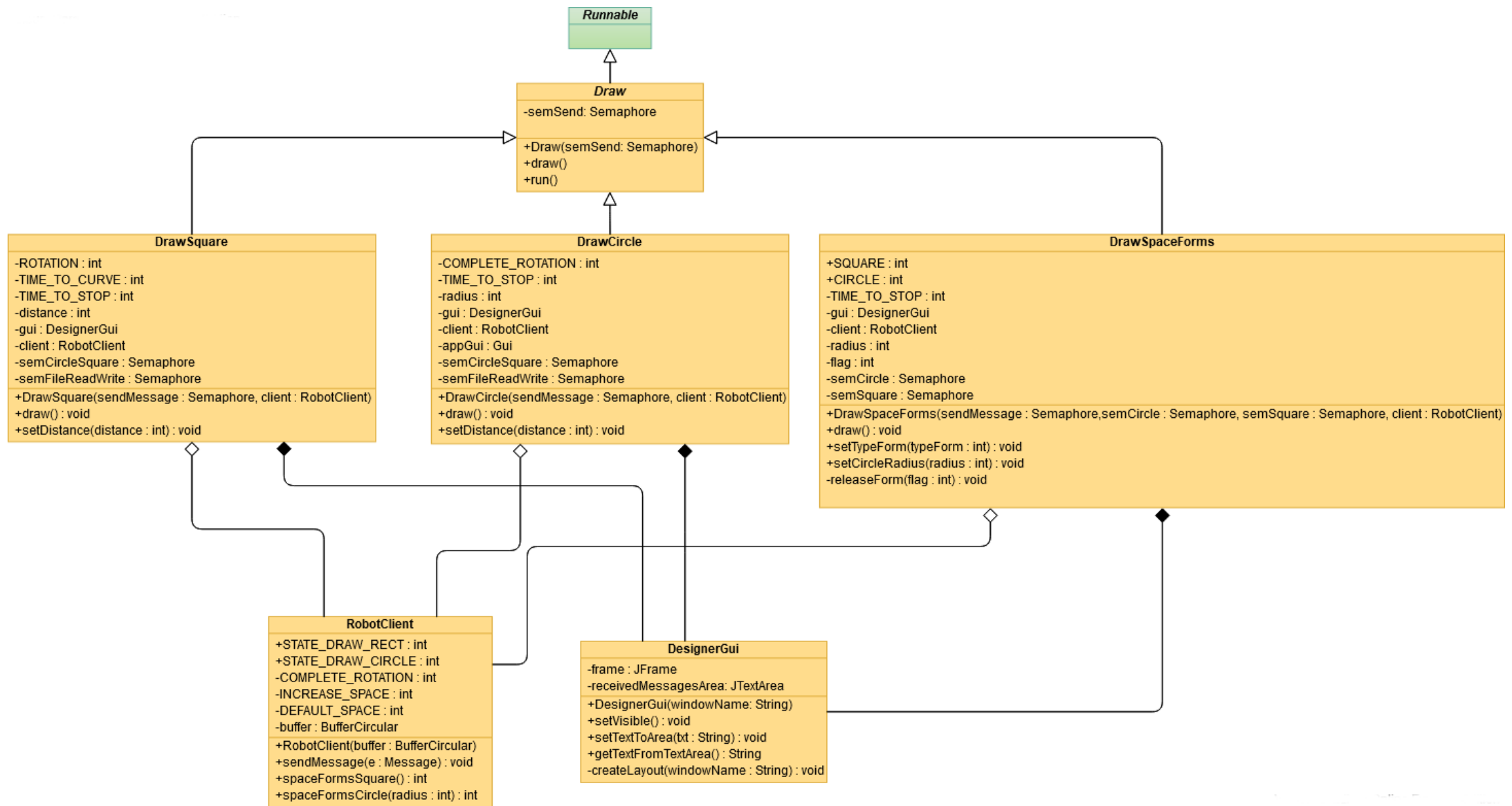


Figura 7 - Diagrama classes do cliente completo

2.1 Buffer Circular

A entidade buffer circular é responsável por armazenar as mensagens que são geradas pelo cliente e lidas pelo servidor. Neste sentido a entidade funciona como ponto de comunicação entre o cliente e o servidor. O fator de ser circular está no modo de funcionamento, isto deve-se a que o número total de mensagens guardadas é limitado a 16 no máximo. Quando todas as 16 posições estão ocupadas o buffer escreve por cima da mensagem existente na primeira posição, a mensagem que segue na segunda posição e assim sucessivamente.

O buffer no seu interior possui um sistema de sincronismo devido a ser um ponto de acesso múltiplo por diversas tarefas. Caso não existisse sincronismo o que poderia acontecer era uma tarefa estar a ler e outra a escrever simultaneamente, a tarefa de escrita poder escrever mais rápido do que a tarefa que está a ler, visto que o sistema operativo executa as tarefas assincronamente, desta maneira é lida uma mensagem que não a pretendida. Para prevenir esta situação foram utilizados os semáforos como sistema de sincronização, mas somente no método de leitura, no método de escrita é referido no enunciado que não deve existir sincronização do mesmo. A sincronização é feita através de um semáforo com zero permissões, quando o servidor tenta ler fica bloqueado e incrementa um contador a dizer que existe uma tarefa á espera de mensagens. Quando um dos clientes escreve no buffer verifica se existe consumidores para ler, caso exista é libertada uma permissão e assim o servidor já consegue ler a mensagem escrita.

O modo de funcionamento das classes é feito de maneira que a aplicação instância a classe BufferCircular e passa a sua instância(refência) aos comportamentos e ao servidor através de argumentos do construtor. Desta forma tanto os clientes como o servidor conseguem comunicar devido a tratar-se do mesmo objeto.

```
//escreve a mensagem no buffer
public void write(Message m) {
    try {
        readWrite.acquire();
        mensagens[pointWrite] = m;
        pointWrite = ++pointWrite % MAX_MENSAGENS;
        numMessagesToRead++;

        if(numConsumersToRead > 0) {
            numConsumersToRead--;
            semRead.release();
        }
        readWrite.release();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Figura 8 - Método de escrita do buffer

```
//lê a mensagem corrente do buffer
public Message read() {
    Message m = null;
    try {
        readWrite.acquire();
        if (numMessagesToRead == 0) {
            numConsumersToRead++;
            readWrite.release();
            semRead.acquire();
        }
        else
            readWrite.release();

        readWrite.acquire();
        m = mensagens[pointRead];
        pointRead = ++pointRead % MAX_MENSAGENS;
        numMessagesToRead--;
        readWrite.release();
    }
    catch (InterruptedException e1) {
        e1.printStackTrace();
    }

    return m;
}
```

Figura 9 - Método leitura do buffer

A classe possui ainda alguns métodos auxiliares que permitem nomeadamente ir buscar mensagens antigas escritas no buffer. Estas mensagens são para o comportamento Espaço Formas saber qual a distância que deve percorrer para não sobrepor a nova forma na forma realizada anteriormente.

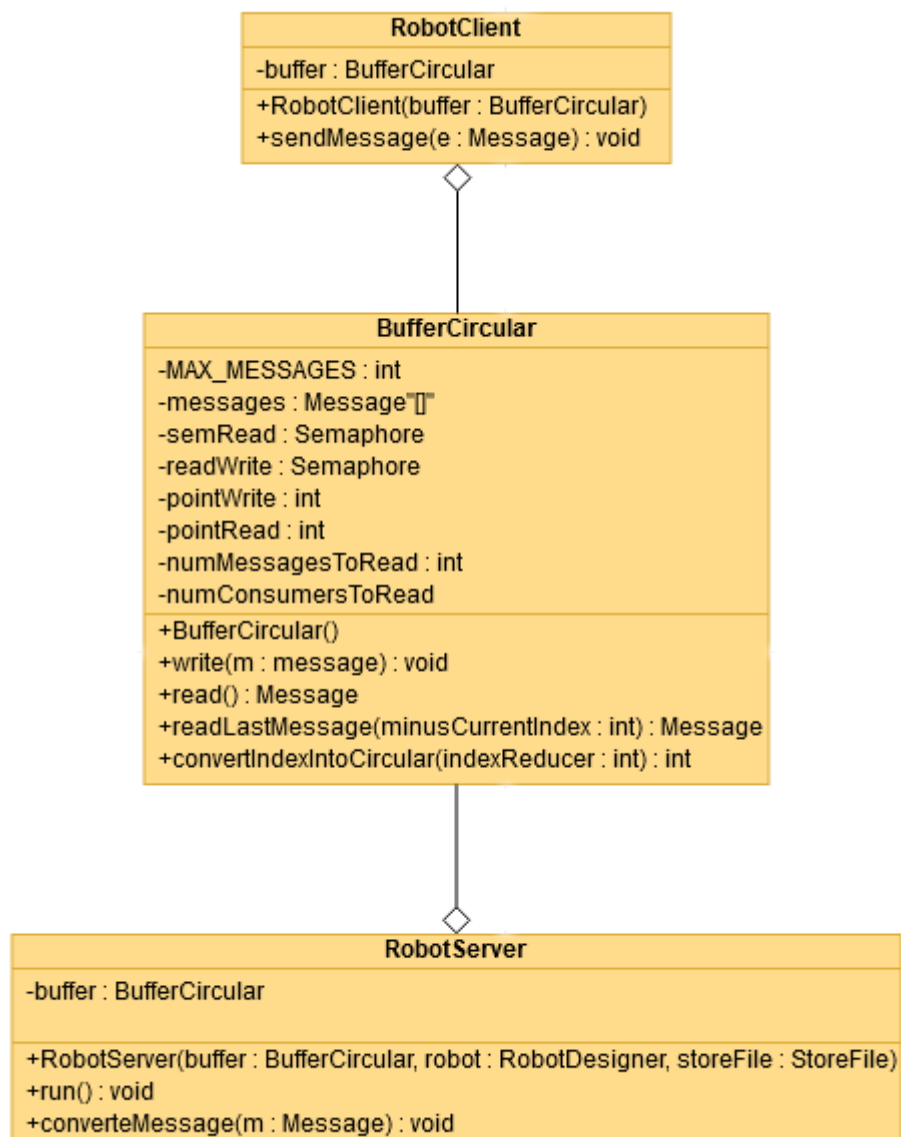


Figura 10 - Diagrama de classes do BufferCircular

2.2 Formato das Mensagens

As mensagens possuem um formato conhecido tanto para o cliente como para o servidor, desta forma é possível as duas entidades comunicarem entre si através do buffer. Estas mensagens obedecem todos ao mesmo formato feito através da interface Message. Todas as classes que representam uma mensagem implementam a interface Message, por sua vez implementam os métodos abstratos da interface e assim existe um padrão comum nos métodos permitindo a comunicação. No total existem quatro tipos diferentes de mensagens sendo elas : reta, curvar direita, curvar esquerda e parar. As mensagens possuem o seguinte formato:

id	distancia	raio	angulo
----	-----------	------	--------

Figura 11 - Formato das mensagens

Todas os campos são representados por números inteiros e no caso da distância, raio e ângulo é representada por distância em centímetros. A interface possui também um método designado por `sleepTime()` que retorna a quantidade de tempo que o comportamento que a desenvolveu deve esperar até realizar a próxima ação.

Cada classe possui desta forma o seu construtor que recebe os parâmetros que necessita e faz a afetação para os atributos, os métodos que não são precisos para a determinada mensagem retornam valor 0 e possui ainda um atributo estático que serve como identificador da classe. Um exemplo de implementação de uma mensagem é a Figura 12.

```
public class Line implements Message {  
  
    public static final int LINE = 1;  
  
    // distancia que vai ser percorrida  
    private int distance;  
  
    public Line(int distance) {  
        this.distance = distance;  
    }  
  
    @Override  
    public int getId() {  
        return LINE;  
    }  
  
    @Override  
    public int getDistance() {  
        return distance;  
    }  
  
    @Override  
    public int getRadius() {  
        return 0;  
    }  
  
    @Override  
    public int getAngle() {  
        return 0;  
    }  
  
    @Override  
    public int sleepTime() {  
        return (distance * 1000) / 30;  
    }  
}
```

Figura 12 - Implementação da mensagem reta

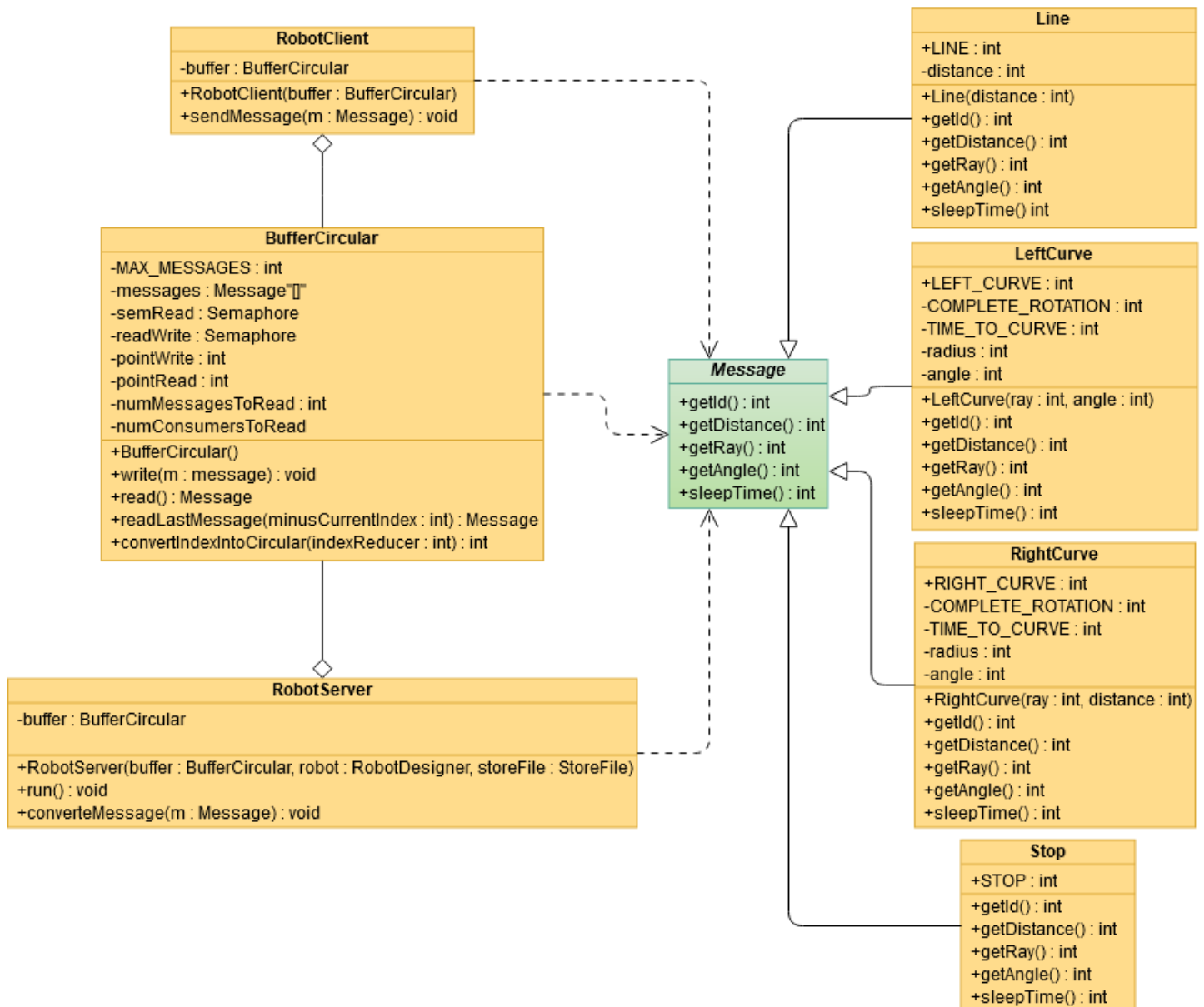


Figura 13 - Diagrama de classes das mensagens

3.1 Servidor

O servidor é a entidade responsável por ler as mensagens, interpretar e posteriormente executar. O servidor lê as mensagens do buffer circular pelo método `run()`, método esse que fica bloqueado pelo buffer, quando recebe uma mensagem este é desbloqueado. No desbloqueio do método `run()` a mensagem é interpretada e executada pelo método `convertMessage(Message m)`. A interpretação é feita pela comparação do id dado pelos atributos de cada classe e pelo método `getId()` da instância recebida, caso seja igual ele realiza a respectiva mensagem. Para executar a mensagem ele exibe a mensagem recebida na interface gráfica do servidor e também tem uma referência para a classe `RobotDesigner` e executa o método correspondente da mensagem. O método `convertMessage(Message m)` possui um semáforo de sincronização no seu interior para garantir que o método não é executado mais do que uma vez no mesmo instante temporal.

```
// função que interpreta a informação do buffer e age conforme a informação
private void convertMessage(Message m) {
    try {
        semWrite.acquire();
        int id = m.getId();

        if(storeFile.getFileChosen() && storeFile.getState() != storeFile.STATE_PLAY) {
            storeFile.setContentToFile(id,m.getDistance(),m.getRadius(),m.getAngle());
            semFileWrite.acquire();
        }

        if (id == Line.LINE) {
            robot.Reta(m.getDistance());
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Reta: " + m.getDistance()+"\n");
        }

        else if (id == RightCurve.RIGHT_CURVE) {
            robot.CurvarDireita(m.getRadius(), m.getAngle());
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Curvar direita: " + m.getRadius() .
        }

        else if (id == LeftCurve.LEFT_CURVE) {
            robot.CurvarEsquerda(m.getRadius(), m.getAngle());
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Curvar esquerda: " + m.getRadius()
        }

        else if (id == Stop.STOP) {
            robot.Parar(false);
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Parar\n");
        }
        semWrite.release();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Figura 14 - Método `convertMessage` da classe `RobotServer`

3.2 Robô desenhador

A execução dos comandos do robô é feita pela classe `RobotDesigner`. Esta classe é criada ao nível da aplicação e passada a sua referência ao servidor. A classe possui uma interface gráfica e uma instância da classe que executa comandos no robô designada por `RobotLegoEV3`. A interface gráfica é responsável pela simulação e a instância do `RobotLegoEV3` por chamar os métodos que fazem o robô físico mover-se baseado no método chamado. Para ambas as formas funcionarem é utilizado um atributo passado no construtor, quando o seu valor é verdade executa os comandos do robô físico, caso contrário realiza somente a simulação. Como se pode observar na figura 15.

```
private DesignerGui gui;
private boolean isPhysical = false;
private RobotLegoEV3 robot;

public RobotDesigner(boolean isPhysical) {
    this.isPhysical = isPhysical;
    gui = new DesignerGui("Mensagens executadas",700,500);
    robot = new RobotLegoEV3();
    gui.setVisible();
}

public boolean OpenEV3(String robotName) {
    if(isPhysical) {
        setTextToGui("OpenEV3\n");
        return robot.OpenEV3(robotName);
        //return true;
    }
    else {
        setTextToGui("OpenEV3\n");
        return true;
    }
}

public void Reta(int distance) {
    if(isPhysical) {
        setTextToGui("Reta: " + distance+"\n");
        robot.Reta(distance);
    }
    else
        setTextToGui("Reta: " + distance+"\n");
}
```

Figura 15 - Excerto da classe `RobotDesigner`

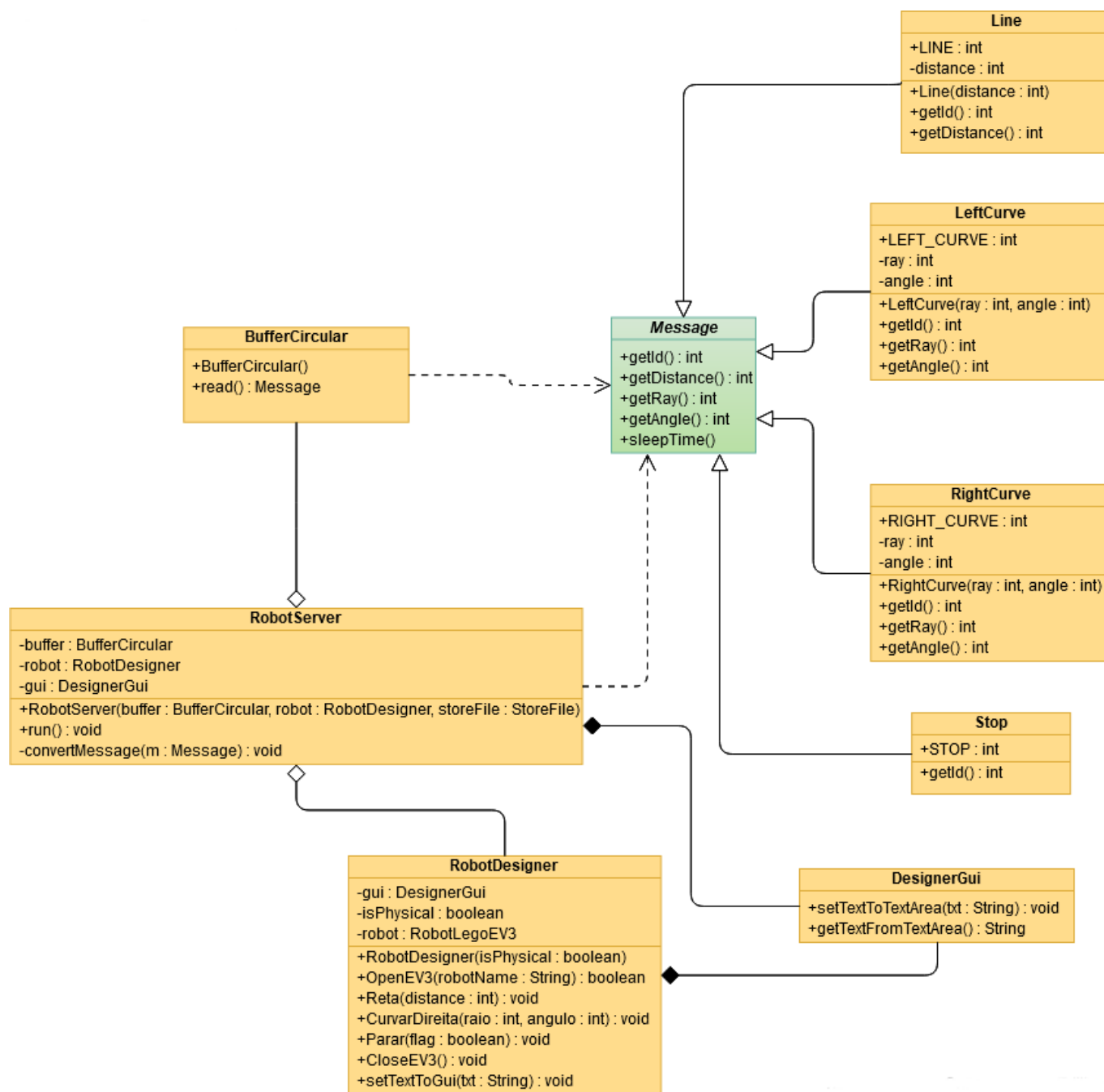


Figura 16 - Diagrama classes servidor com robô desenhador

4.1 Manipulação de ficheiros

A manipulação dos ficheiros é feita pelo sistema de ficheiros do sistema operativo. Para existir a manipulação deste é necessário a criação do ficheiro. A criação do ficheiro é feito pela classe `FileDialog` que permite definir tanto a diretoria do ficheiro como o nome do mesmo. A partir desses dois parâmetros é possível instanciar a classe `File` e assim criar o ficheiro físico pelo método `createNewFile()`. O ficheiro criado é no formato `txt` para ser possível de visualizar e interpretar por humanos o conteúdo escrito no seu interior.

A classe que comporta a manipulação de ficheiros designa-se `StoreFile`, esta classe no seu interior possui uma máquina de estados. A máquina de estados possui quatro estados distintos designados por: seleção do ficheiro, gravar, parar e executar.

- O estado de seleção de ficheiro tal como o nome indica permite realizar a tarefa disrita anteriormente de seleccionar a diretoria e criar o ficheiro em memória para ser posteriormente utilizado.
- O estado gravar permite gravar as intruções dadas ao robô lidas pelo servidor, quer isto dizer que a grvação é feita no ato de interpretação dos comandos no servidor. A gravação é feita com recurso a uma classe designada de `BufferedWriter` que permite escrever num ficheiro pelo método `write(String s)`, no fim da sua utilização é necessário fechar o recurso. O método possui sincronização no seu interior para que a leitura e a escrita não se realizem as duas no mesmo intante temporal.
- O estado parar é o estado utilizado para quando o programa não realiza qualquer ação, ou para parar a execução do executar, ou parar a gravação.
- O estado executar é quando o programa lê as mensagens escritas no ficheiro, interpreta e manda executar essas mesmas mensagens. A leitura é feita pela classe `BufferedReader` que possui um comportamento similar ao método de escrita mas permite ler.

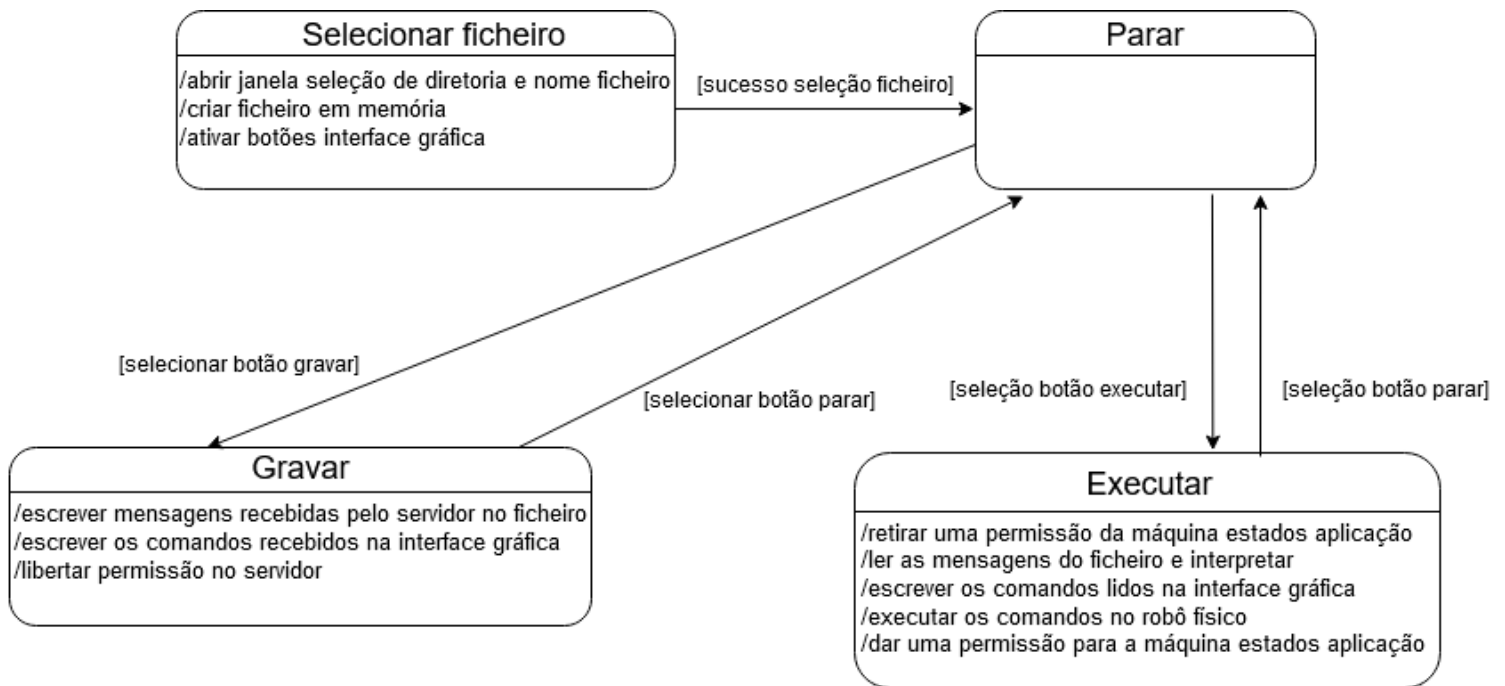


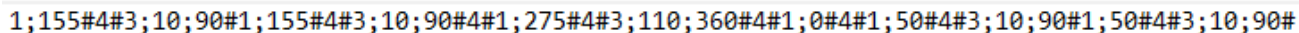
Figura 17 - Máquina estados da classe StoreFile

A máquina de estados funciona de tal maneira que é iniciado no estado [Selecionar ficheiro] que só fica possível realizar as suas ações quando na aplicação se conecta com o robô ou por simulação ou ao robô físico. Quando selecionado o ficheiro com sucesso a aplicação transita para o estado [Parar]. Pela interface grafica, através de cliques em botões, o utilizador consegue realizar as transições na máquina de estados. As transições pode ser do estado [Parar] para [Gravar] e vice-versa ou do estado [Parar] para [Executar] e vice-versa.

No estado [Gavar] como o servidor possui uma referência para a instância de StoreFile, desta forma consegue saber se a classe está a gravar os comandos ou não. O estado [Gravar] bloqueia a execução da interpretação do servidor, onde só liberta essa execução após ser executada a informação da máquina estados da classe StoreFile. Desta forma é garantido que só após ser escrito a informação no ficheiro é que o servidor manda executar os comandos.

No estado [Executar] este como utiliza o robô e os clientes também utilizam o robô, se o utilizador mandar executar o gravar formas e logo de seguida enviar uma forma geométrica, isto daria então um comportamento que seria alvo de uma fonte de erros, contudo foi utilizado um semáforo para prevenir isso. A sequência de execução das instruções será então primeiro o [Executar] acabar as suas intruções e de seguida é executado a forma geométrica pretendida pelo utilizador. Caso o utilizador envie várias formas como as libertações de permissões são assincronas a forma geométrica executada primeiro pode não ser a primeira a ter sido clicada.

A estrutura das mensagens ficou implementada de tal maneira que mensagens diferentes são separadas pelo caracter # e informações dentro da mesma mensagem ficou dividida com o caracter ;. Para a identificação das mensagens foi utilizada, assim como está implementado na classe Message, a utilização do Id e os valores necessários para a realização da mensagem vem também em número separados por ; a seguir ao Id. A mensagem vem em formato de String escrita pelo método write(String s) como foi dito anteriormente. Uma visualização das mensagens seria a figura 18.



1;155#4#3;10;90#1;155#4#3;10;90#4#1;275#4#3;110;360#4#1;0#4#1;50#4#3;10;90#1;50#4#3;10;90#

Figura 18 - Exemplo de mensagem escrita no ficheiro

Nas mensagem o número 1 representa uma reta, o número 2 representa uma curva á direita, o número 3 uma curva á esquerda e o número 4 um parar a falso. Para esta classe saber quanto tempo deve esperar até executar a próxima mensagem é reconstruído o objeto do tipo Message e chamado o método sleepTime().

4.2 Interface gráfica da manipulação de ficheiros

A interface gráfica da manipulação de ficheiro possui quatro botões que vão sendo ativados e desativados conforme a utilização do utilizador. Quando o utilizador seleciona o botão é redirecionado para uma página do FileDialog e quando é feito com sucesso são desbloqueados os botões gravar e executar. Quando o utilizador seleciona um dos dois botões são desativados ambos os botões e ativado o botão de parar, que permite parar a execução do executar ou parar a gravação. Quando selecionado o botão parar são novamente ativados os botões de executar e gravar.

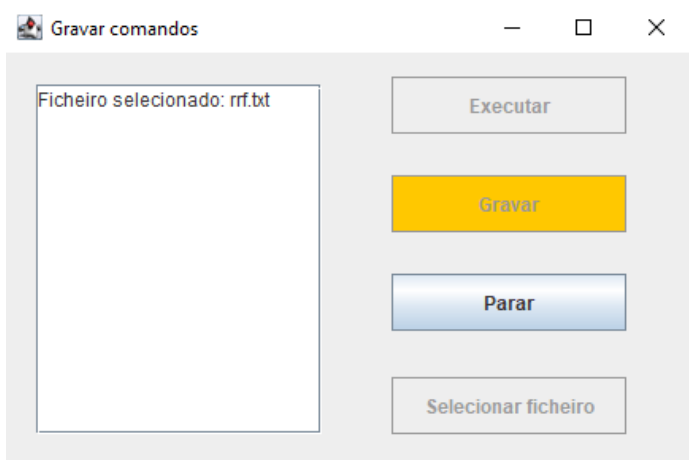


Figura 19 - Interface gráfica a gravar

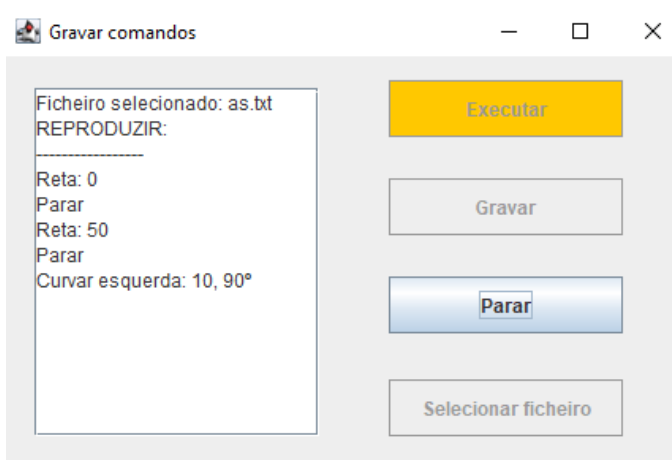


Figura 20 - Interface gráfica a executar

A interface também permite exibir os comandos que estão a ser escritos no ficheiro e também exibir os comandos que estão a ser lidos do ficheiro.

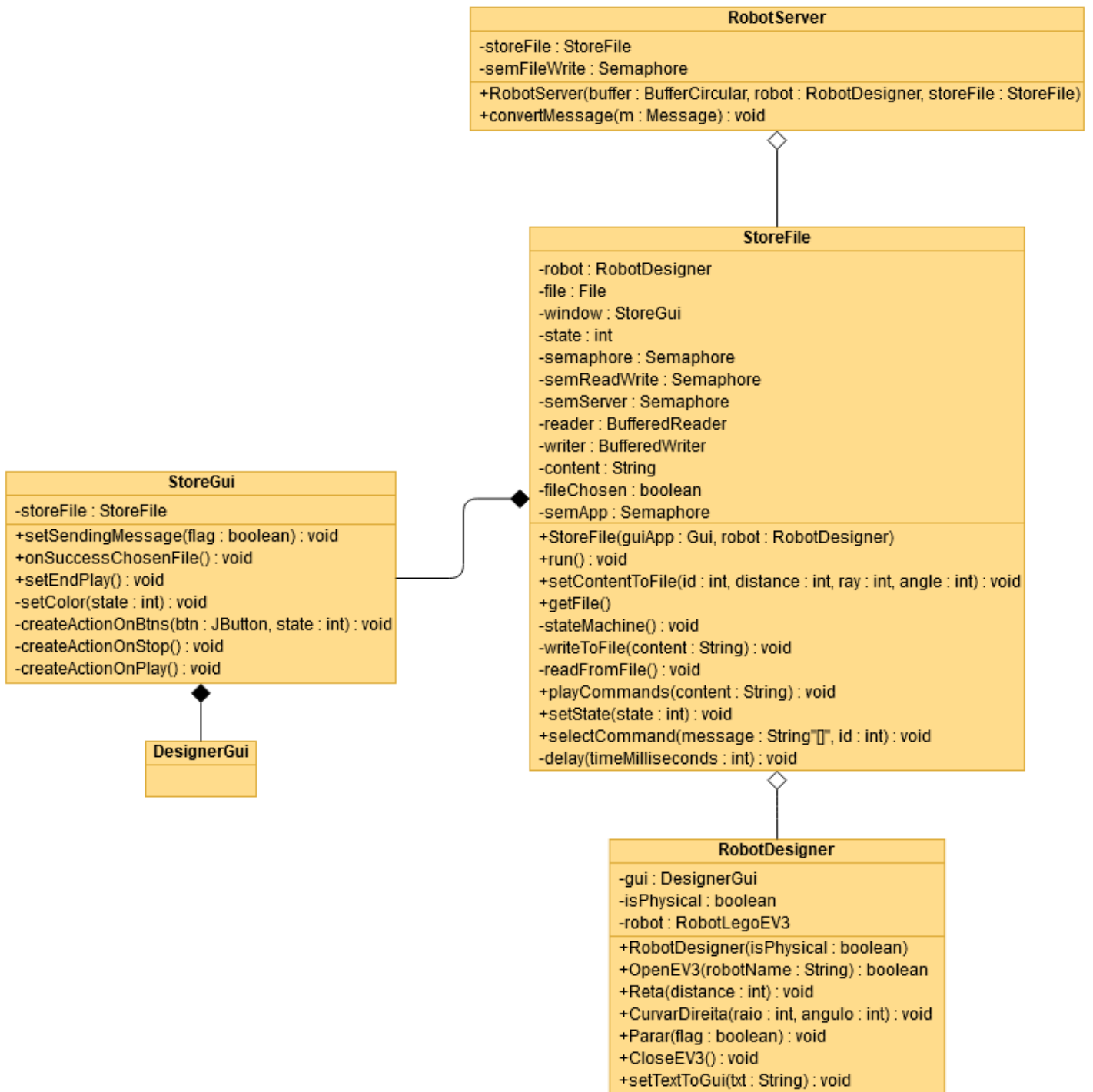


Figura 21 - Diagrama classes da manipulação de ficheiros

5.1 Aplicação

A aplicação é a entidade de maior nível, que possui no seu interior todas as restantes entidades, sendo por isso quem possui o método *main*. A aplicação tem associada uma interface gráfica que permite desenhar formas geométricas e entregar essa informação aos comportamentos, dando seguimento a toda a sequência lógica da aplicação. A aplicação é que realiza o comando de abertura e fecho da comunicação com o robô. Como a aplicação é responsável por instanciar a classe RobotDesigner pela alteração da variável IS_PHYSICAL_ROBOT para *true* passa a funcionar com o robô físico.

A aplicação é responsável por instanciar todas as entidades e lançar algumas dessas entidades como Threads pela utilização do método *start()*.

Para entregar as mensagens aos clientes esta classe possui uma máquina de estados com somente dois estados e semáforos que permite uma execução ordeira das instruções. Existe um semáforo com zero permissões para bloquear a execução da máquina de estados antes de esta executar qualquer instrução, a permissão é libertada quando se clica num botão na interface gráfica ou desenhar quadrado ou desenhar círculo. Existem mais três semáforos um para cada cliente, estes servem para serem chamados numa sequência ordeira, onde primeiro é executado o espaçar formas só de seguida a forma geométrica pretendida. Isto é feito pelos três semaforos todos criados com zero permissões e passados como argumento do construtor dos clientes, o funcionamento é tal que primeiro é libertada uma permissão para o espaçar formas e no fim da execução do método *draw()* é libertada uma permissão para a forma geométrica pretendida.

Existem mais dois semáforos necessários para o funcionamento sequencial da aplicação. Um dos semáforos serve para bloquear a execução quando está uma forma geométrica em execução e o utilizador clica num botão para executar outra forma geométrica. Desta forma a segunda forma geométrica não é perdida, mas é executada após a primeira execução acabar as suas instruções. O segundo semáforo serve para quando está a ser feita uma forma geométrica e o utilizador clica no botão executar para ler os comandos escritos no ficheiro ou vice-versa. Da

mesma forma que o anterior semáforo, primeiro acaba a execução que está a decorrer e de seguida executa a tarefa pendente.

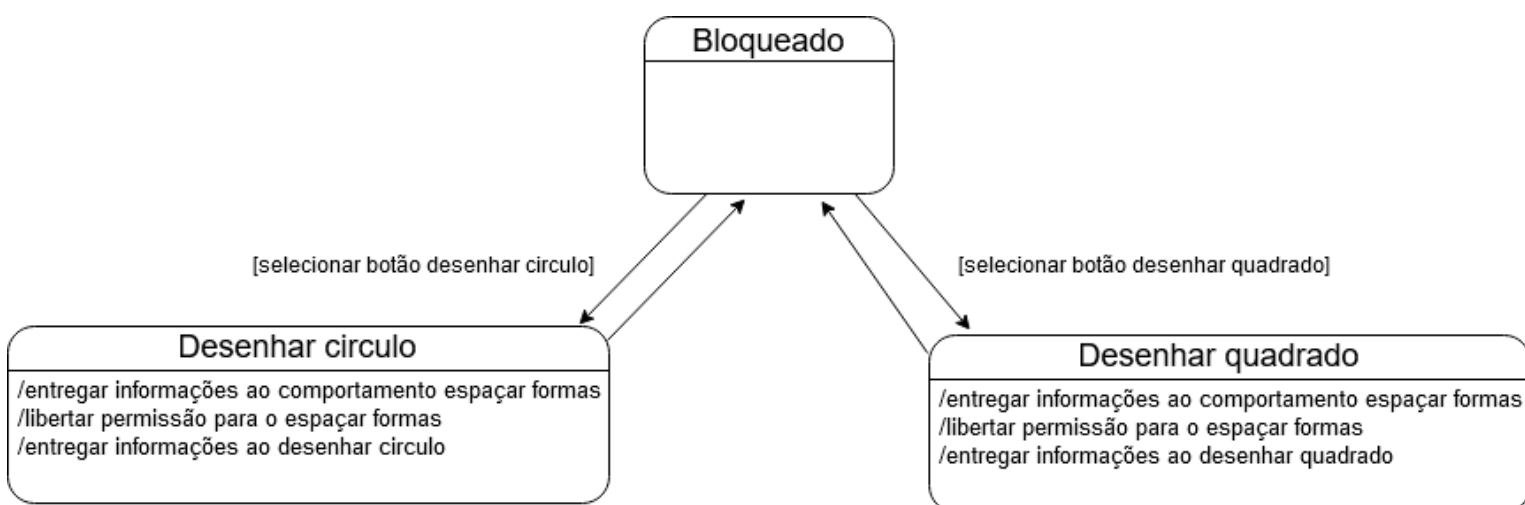


Figura 22 - Máquina estados da aplicação

5.2 Interface gráfica da aplicação

A interface gráfica da aplicação serve para enviar informação para a aplicação ou chamar métodos da aplicação diretamente da interface gráfica. A interface permite definir o nome do robô onde existe um nome por omissão, mas é possível alterar dinamicamente o nome e ser conectado a um robô que não o nome por omissão. A interface possui um *radio button* que permite ligar ao robô, quando ligado é possível então enviar comandos para o robô definindo um número, tamanho em centímetros, da forma geometrica pretendida. Quando se clica no botão para desenhar internamente a interface possui validações que não permitem o utilizador de definir valores inválidos ou realizar comportamentos que possam vir a desencadear erros.

Durante a execução do programa é possível desativar a conexão com um robô e ativar conexão com um novo robô diferente do utilizado inicialmente, isto porque é fechada a comunicação e estabelecida uma nova pelos métodos da aplicação. Quando se clica no botão de fechar a aplicação envia o comando de Parar(true), para o caso do robô estar a executar uma instrução e não ser necessário esperar, e posteriormente fechar a comunicação e de seguida é terminada a execução de toda a aplicação. O fecho total da aplicação só acontece pelo fecho da interface da aplicação, as restantes o programa continua a sua execução. A interface possui o aspeto da figura 23.

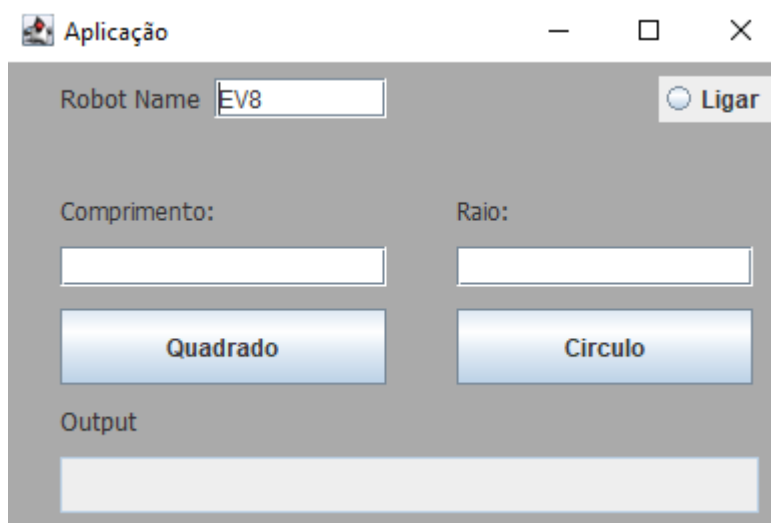


Figura 23 - Interface gráfica da aplicação

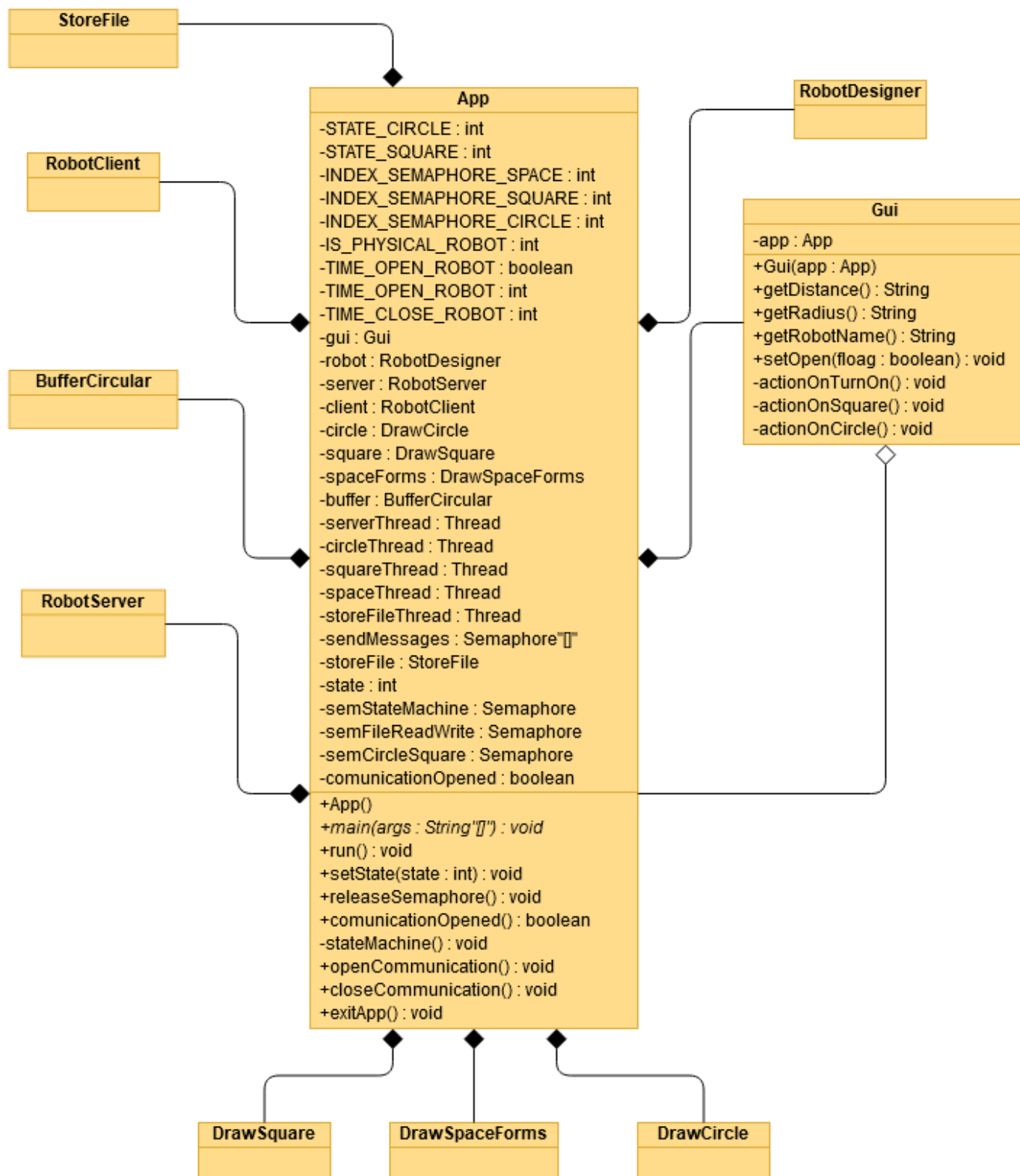


Figura 24 - Diagrama classes da aplicação

6.1 Funcionamento global da aplicação

A aplicação tem uma sequência funcional das instruções. A sequência é iniciada pela criação da instância da aplicação que por sua vez instância os comportamentos, o servidor, a classe de manipulação do ficheiro e a própria aplicação como Threads. A aplicação também instância classes que não são lançadas como Threads nomeadamente o buffer, o robô desenhador e também a gui. Todas as classes lançadas como Threads possuem um ciclo infinito no seu interior onde todas elas estão bloqueadas á espera passivamente que seja dada uma permissão para continuar a sua execução.

Antes de iniciar a comunicação com o robô não é possível realizar nenhuma ação, quando é estabelecida a comunicação fica assim possível de enviar comandos e seleccionar o ficheiro para guardar os comandos. Quando é enviado um comando é libertado uma permissão na máquina de estados da aplicação, por sua vez a aplicação liberta uma permissão no comportamento de espaçar formas. O espaçar formas quando acaba a sua execução liberta uma permissão para o comportamento pretendido seja ele um quadrado ou um circulo. Tanto o espaçar formas como a forma geometrica escreve comandos no buffer, por cada comando escrito é libertado uma permissão na leitura, desta forma o servidor consegue ler a informação no buffer interpreta-a e executa o comando correspondente no robô desenhador. Caso a aplicação esteja a gravar os comandos os comandos são escritos no ficheiro e só de seguida é que o servidor pode interpretar a informação. De seguida, o utilizador pode enviar mais um comando que segue sempre a lógica descrita anteriormente, ou mandar executar os comandos escritos no ficheiro. Para isso a classe lê os comandos do ficheiro e a própria classe manda executar os comandos no robô desenhador. Caso durante a execução de uma forma geométrica se seleccionar outra forma geométrica a segunda ficará em espera até a primeira acabar a sua execução. Por fim o utilizador pode fechar a aplicação, por sua vez a comunicação com o robô é fechada antes de a aplicação ser terminada.

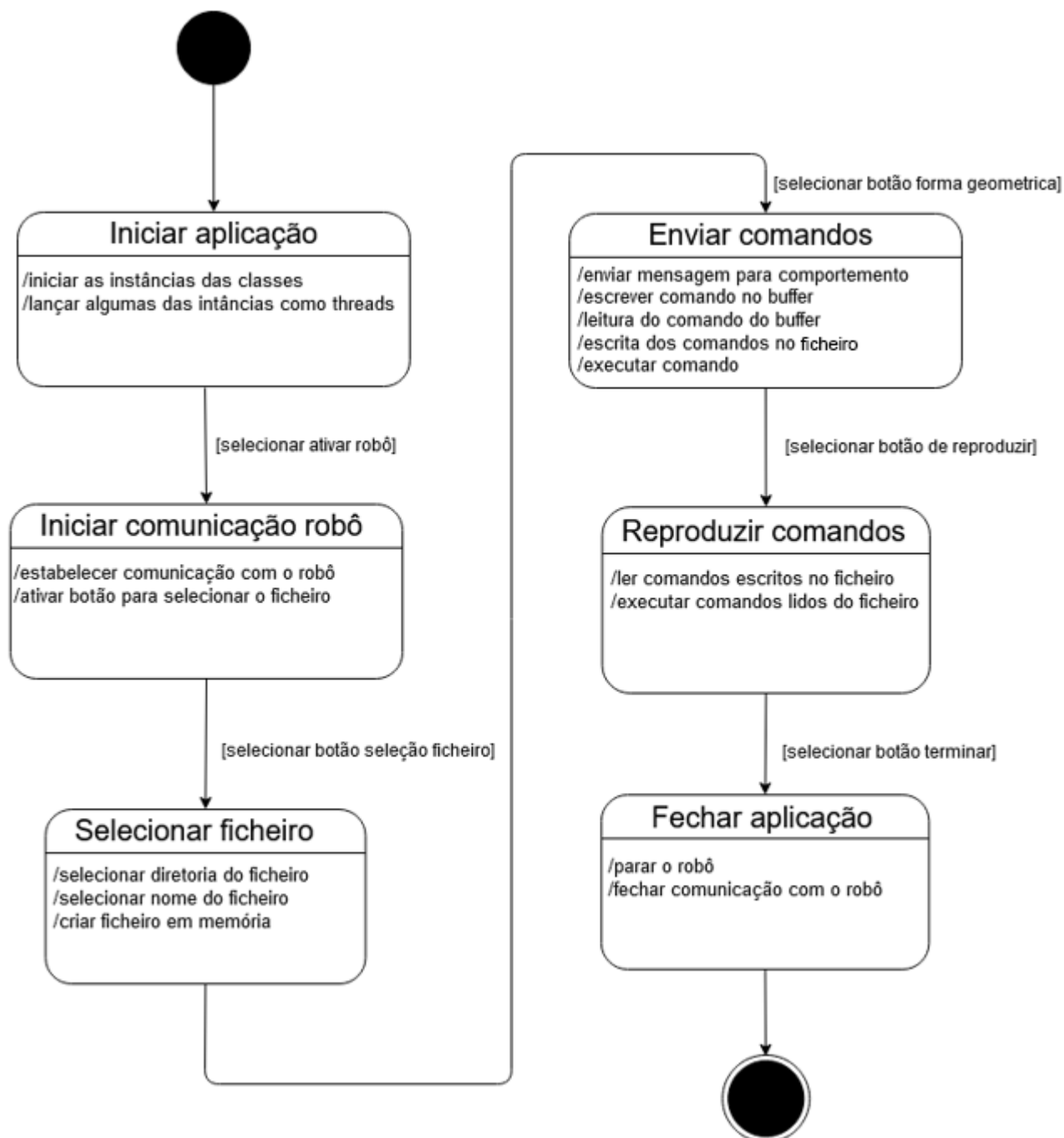


Figura 25 - Diagrama estados para a aplicação

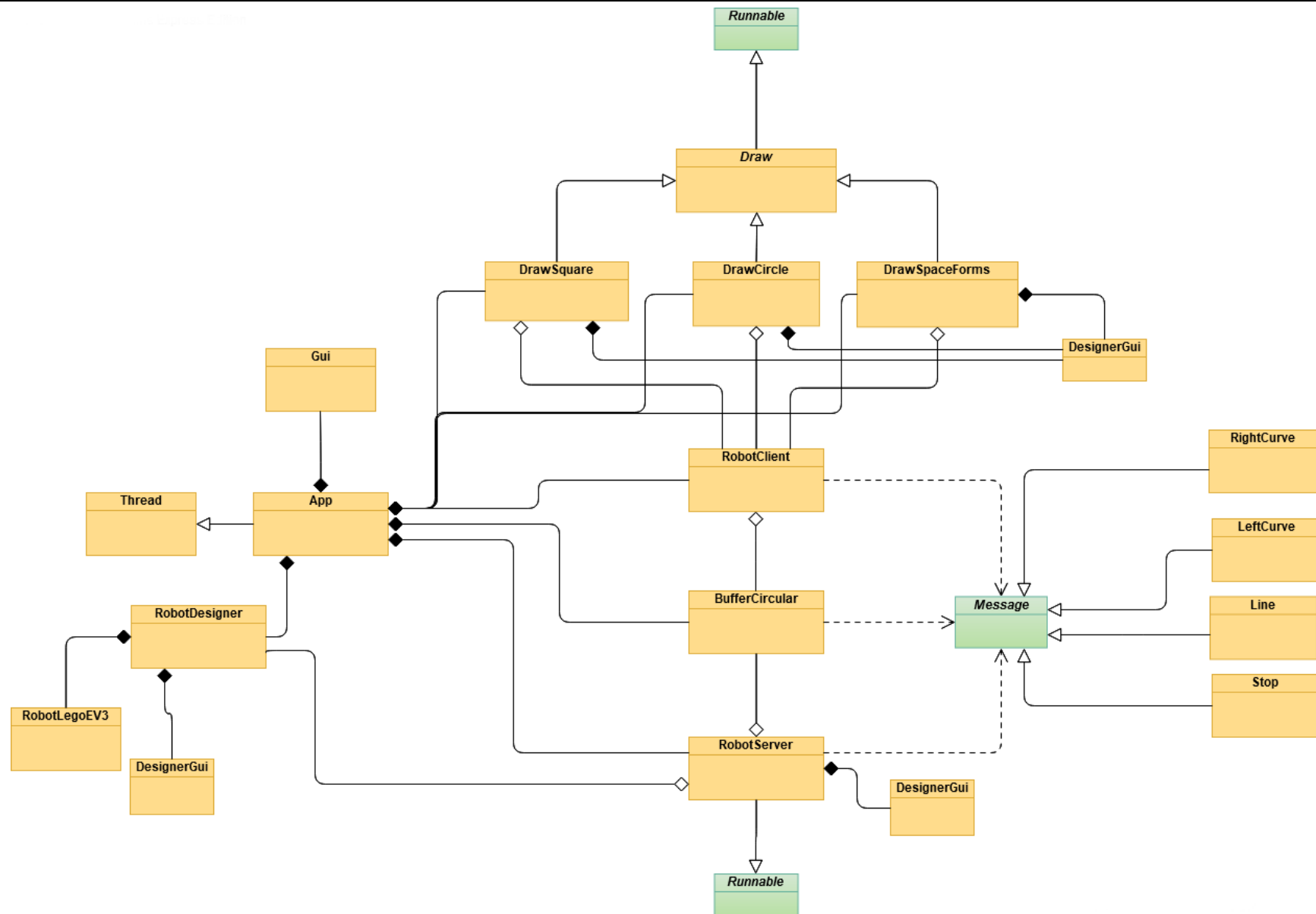


Figura 26 - Diagrama classes sem escrita no ficheiro

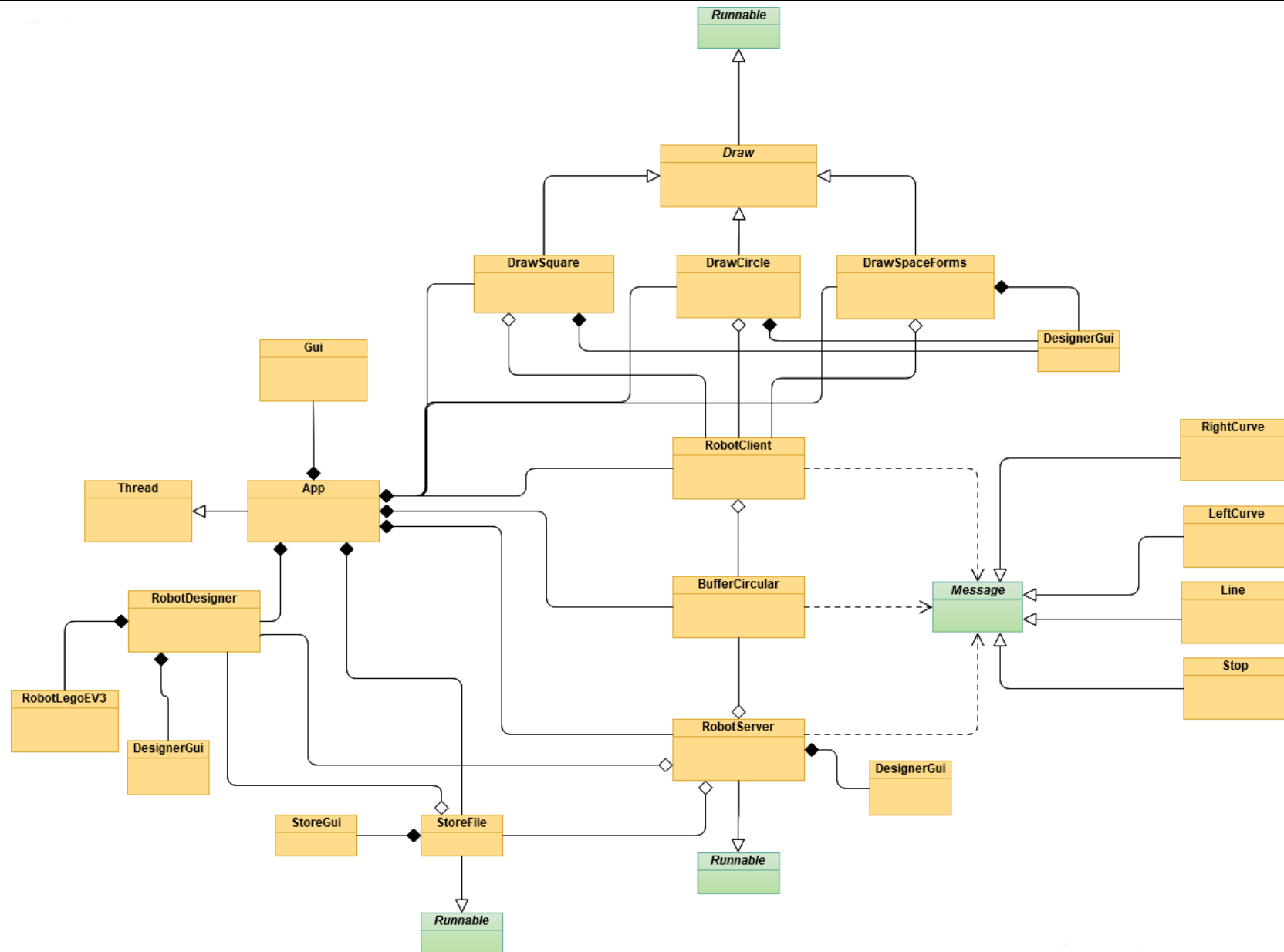


Figura 27 - Diagrama classes total

3. Conclusões

Em suma com a realização do presente trabalho prático o grupo adquiriu os seguintes conhecimentos:

- O que são tarefas leves, qual a sua funcionalidade e como são utilizadas num sistema operativo.
- O que é sincronia entre tarefas e qual a sua funcionalidade num programa.
- Como garantir que um recurso é acedido por exclusão mútua para prevenir erros na escrita ou leitura.
- Como garantir que um programa fica bloqueado á espera que seja dada uma permissão sem gastar CPU.
- Quais as diferentes formas de bloquear passivamente a execução de uma tarefa ou por semáforos ou monitores.
- Criar e utilizar ficheiros para gravar informações da execução da aplicação.
- Implementar uma aplicação no modelo produtor-consumidor.
- Utilizar um robô físico para visualizar a execução das instruções e confirmar o bom funcionamento da sincronia.

Nos resultados da implementação a aplicação ficou a funcionar no que diz respeito ao trabalho pretendido. Contudo existem melhorias que podiam ser implementadas nomeadamente na libertação de permissões, quando está mais do que um comportamento á espera. O grupo pensou noutra outra maneira de implementar o manipulador de ficheiros, onde a solução passaria por ser também um cliente, contudo a complexidade do código aumentou bastante e por isso não foi implementado.

4. Bibliografia

[1] Folhas de Fundamentos Sistemas Operativos, Jorge Pais, 2020/2021

<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html>

<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>

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

5. Anexo

```
import java.util.concurrent.Semaphore;

public class App extends Thread{

    public final int STATE_CIRCLE = 1;
    public final int STATE_SQUARE = 2;

    private final int INDEX_SEMAPHORE_SPACE = 0;
    private final int INDEX_SEMAPHORE_SQUARE = 1;
    private final int INDEX_SEMAPHORE_CIRCLE = 2;

    //Alterar este valor quando se pretende que utilize o robô físico
    private boolean IS_PHYSICAL_ROBOT = false;

    private int TIME_OPEN_ROBOT = 500;
    private int TIME_CLOSE_ROBOT = 100;
    private Gui gui;
    private RobotDesigner robot;
    private RobotServer server;
    private RobotClient client;
    private DrawCircle circle;
    private DrawSquare square;
    private DrawSpaceForms spaceForms;
    private BufferCircular buffer;

    private Thread serverThread;
    private Thread circleThread;
    private Thread squareThread;
    private Thread spaceThread;
    private Thread storeFileThread;
```

```
//semaforos para os 3 tipos de clientes
private Semaphore[] sendMessages;

private StoreFile storeFile;

private int state;
//semáforo para bloquear a execução da máquina de estados(libertado quando se clica
num dos botões)
private Semaphore semStateMachine;

//semáforo para não executar forma e executar comandos na leitura e escrita robô
private Semaphore semFileReadWrite;

//semáforo para não executar duas formas geometricas ao mesmo tempo
private Semaphore semCircleSquare;

//variável que dita se o canal de comunicação foi aberto com sucesso ou não
private boolean communicationOpened = false;

public App() {
    gui = new Gui(this);
    gui.setVisible();
    buffer = new BufferCircular();
    robot = new RobotDesigner(IS_PHYSICAL_ROBOT);

    semFileReadWrite = new Semaphore(1);
    storeFile = new StoreFile(robot,semFileReadWrite);
    server = new RobotServer(buffer, robot, storeFile);
    client = new RobotClient(buffer);
```

```
sendMessages = new Semaphore[3];
for(int i = 0; i < sendMessages.length;i++) {
    Semaphore s = new Semaphore(0);
    sendMessages[i] = s;
}
```

```
semCircleSquare = new Semaphore(1);
```

```
spaceForms = new
DrawSpaceForms(sendMessages[INDEX_SEMAPHORE_SPACE],sendMessages[INDEX_
SEMAPHORE_CIRCLE],sendMessages[INDEX_SEMAPHORE_SQUARE],client);
square = new
DrawSquare(sendMessages[INDEX_SEMAPHORE_SQUARE],client,semCircleSquare,sem
FileReadWrite);
circle = new
DrawCircle(sendMessages[INDEX_SEMAPHORE_CIRCLE],client,semCircleSquare,semFi
leReadWrite);
```

```
circleThread = new Thread(circle);
squareThread = new Thread(square);
spaceThread = new Thread(spaceForms);
```

```
serverThread = new Thread(server);
```

```
storeFileThread = new Thread(storeFile);
```

```
circleThread.start();
squareThread.start();
spaceThread.start();
```

```
        serverThread.start();

        storeFileThread.start();

        semStateMachine = new Semaphore(0);

    }

    public static void main(String[] args) {
        App app = new App();
        app.start();
    }

    @Override
    public void run() {
        for (;;) {
            stateMachine();
        }
    }

    public void setState(int state) {
        this.state = state;
    }

    //liberta uma permissão(utilizado na gui quando se clica nos botões)
    public void releaseSemaphore() {
        this.semStateMachine.release();
    }

    public boolean communicationOpened() {
```

```
        return this.comunicationOpened;
    }

    //não é necessário lançar permissões para o circulo ou quadrado(feito internamente)
    //o espaçar formas e que liberta permissões
    private void stateMachine() {
        try {
            semStateMachine.acquire();
            if (comunicationOpened) {
                semCircleSquare.acquire();
                semFileReadWrite.acquire();
                switch (state) {

                    case STATE_CIRCLE:

                        int radius = Integer.parseInt(gui.getRadius());
                        gui.setTextToOutput("Circulo: " + radius);

                        // espaçar formas
                        spaceForms.setCircleRadius(radius);
                        spaceForms.setTypeForm(spaceForms.CIRCLE);
                        sendMessages[INDEX_SEMAPHORE_SPACE].release();

                        // desenhar quadrado
                        circle.setDistance(radius);
                        break;

                    case STATE_SQUARE:

                        int distance = Integer.parseInt(gui.getDistance());
                        gui.setTextToOutput("Quadrado: " + distance);
```

```
        // espaçar formas
        spaceForms.setCircleRadius(distance);
        spaceForms.setTypeForm(spaceForms.SQUARE);
        sendMessages[INDEX_SEMAPHORE_SPACE].release();

        // desenhar quadrado
        square.setDistance(distance);
        break;
    }
}

} catch (InterruptedException e) {
    e.printStackTrace();
}

}

//função para abrir comunicação com o robô
public void openCommunication() {
    try {
        gui.setTextToOutput("Abrir: "+gui.getRobotName());
        boolean open = robot.OpenEV3(gui.getRobotName());
        Thread.sleep(TIME_OPEN_ROBOT);
        gui.setOpen(open);//não fica selecionado quando não consegue abrir
        communicationOpened = open;
        if(open)
            storeFile.setEnableSelectBtn();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

//função chamada quando o robô está conectado e se desconecta

```
public void closeCommunication() {  
    try {  
        if (comunicationOpened) {  
            comunicationOpened = false;  
            gui.setTextToOutput("Fechar");  
            Thread.sleep(TIME_CLOSE_ROBOT);  
            robot.CloseEV3();  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

//função chamada quando se termina a aplicação

```
public void exitApp() {  
    try {  
        if (comunicationOpened) {  
            robot.Parar(true);  
            Thread.sleep(TIME_CLOSE_ROBOT);  
            robot.CloseEV3();  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

```
import java.util.concurrent.Semaphore;

public class BufferCircular {

    private final int MAX_MENSSAGES = 16;
    private Message[] messages;
    private Semaphore semRead;
    //semaforo para garantir que a leitura e escrita não é feita em simultaneo
    private Semaphore readWrite;
    private int pointWrite = 0;
    private int pointRead = 0;
    private int numMessagesToRead = 0;
    private int numConsumersToRead = 0;

    public BufferCircular() {
        messages = new Message[MAX_MENSSAGES];
        readWrite = new Semaphore(1);
        semRead = new Semaphore(0);
    }

    //escreve a mensagem no buffer
    public void write(Message m) {
        try {
            readWrite.acquire();
            messages[pointWrite] = m;
            pointWrite = ++pointWrite % MAX_MENSSAGES;
            numMessagesToRead++;

            if(numConsumersToRead > 0) {
                numConsumersToRead--;
                semRead.release();
            }
        }
    }
}
```

```
        }
        readWrite.release();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

//lê a mensagem corrente do buffer
public Message read() {
    Message m = null;
    try {
        readWrite.acquire();
        if (numMessagesToRead == 0) {
            numConsumersToRead++;
            readWrite.release();
            semRead.acquire();
        }
        else
            readWrite.release();

        readWrite.acquire();
        m = messages[pointRead];
        pointRead = ++pointRead % MAX_MENSSAGES;
        numMessagesToRead--;
        readWrite.release();
    }
    catch (InterruptedException e1) {
        e1.printStackTrace();
    }

    return m;
}
```

```
}
```

```
//função para ler mensagens antigas, por isso não utiliza o semaforo de read
```

```
//por vezes tem de se ler o indice -2 outras vezes -4
```

```
//não incrementa o indice de leitura porque serve para ler mensagens antigas
```

```
public Message readLastMessage(int minusCurrentIndex) {
```

```
    try {
```

```
        readWrite.acquire();
```

```
        if (messages[convertIndexIntoCircular(minusCurrentIndex)] != null) {
```

```
            Message                                m                                =
```

```
messages[convertIndexIntoCircular(minusCurrentIndex)];
```

```
            return m;
```

```
        }
```

```
    } catch (InterruptedException e) {
```

```
        e.printStackTrace();
```

```
    } finally{
```

```
        readWrite.release();
```

```
    }
```

```
    return null;
```

```
}
```

```
//converte as mensagens para circular porque tem de andar para trás
```

```
private int convertIndexIntoCircular(int indexReducer) {
```

```
    if(pointWrite == 0 || pointWrite == 1|| (pointWrite==2 && indexReducer==4) ||  
(pointWrite==3 && indexReducer==4))
```

```
        return MAX_MENSSAGES-indexReducer+pointWrite;
```

```
    return (pointWrite-indexReducer)%MAX_MENSSAGES;
```

}

}

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class DesignerGui {

    private JFrame frame;
    private JTextArea receivedMessagesArea;

    public DesignerGui(String windowName, int posX, int posY) {
        createLayout(windowName, posX, posY);
    }

    private void createLayout(String windowName, int posX, int posY) {
        frame = new JFrame();
        frame.setBounds(posX, posY, 450, 300);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.getContentPane().setLayout(null);
        frame.setTitle(windowName);

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                frame.dispose();
            }
        });
    }
}
```

```
// adiciona uma textArea com scroll
JScrollPane sbrText = new JScrollPane();

sbrText.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

sbrText.setBounds(20, 20, 395, 220);
receivedMessagesArea = new JTextArea();
receivedMessagesArea.setEditable(false);
sbrText.setViewportView(receivedMessagesArea);
frame.getContentPane().add(sbrText);
}

public void setVisible() {
    frame.setVisible(true);
}

public void setTextToTextArea(String txt) {
    receivedMessagesArea.setText(txt);

    receivedMessagesArea.setCaretPosition(receivedMessagesArea.getDocument().getLength());
}

public String getTextFromTextArea() {
    return receivedMessagesArea.getText();
}
}
```

```
import java.util.concurrent.Semaphore;

public abstract class Draw implements Runnable{

    private Semaphore semSend;

    public Draw(Semaphore semSend) {
        this.semSend = semSend;
    }

    public abstract void draw();

    //método que fica a correr até se clicar numa forma geometrica
    //que liberta uma permissão e então pode desenhar
    @Override
    public void run() {
        for (;;) {
            try {
                semSend.acquire();
                draw();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
import java.util.concurrent.Semaphore;

public class DrawCircle extends Draw {

    private final int COMPLETE_ROTATION = 360;
    private int radius;
    private DesignerGui gui;
    private RobotClient client;
    private Semaphore semCircleSquare;
    private Semaphore semFileReadWrite;

    public DrawCircle(Semaphore sendMessage, RobotClient client, Semaphore
semCircleSquare, Semaphore semFileReadWrite) {
        super(sendMessage);
        this.client = client;
        this.semFileReadWrite = semFileReadWrite;
        this.semCircleSquare = semCircleSquare;

        gui = new DesignerGui("Desenhar circulo",520,100);
        gui.setVisible();
    }

    @Override
    public void draw() {
        Message curve = new LeftCurve(radius, COMPLETE_ROTATION);
        Message stop = new Stop();

        try {
            client.sendMessage(curve);
            gui.setTextToTextArea(gui.getTextFromTextArea()+"Circulo:
"+radius+"\n");
```

```
        Thread.sleep(curve.sleepTime());
        client.sendMessage(stop);
        gui.setTextToTextArea(gui.getTextFromTextArea()+"Parar\n\n");
        Thread.sleep(stop.sleepTime());

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // libertar permissão para o quadrado ou circulo
    semCircleSquare.release();

    // para libertar a permissão entre a aplicação e gravador de comandos
    semFileReadWrite.release();

}

//dar a distância que a forma geometrica deve ter
public void setDistance(int distance) {
    this.radius = distance;
}

}
```

```
import java.util.concurrent.Semaphore;

public class DrawSpaceForms extends Draw {

    public final int SQUARE = 1;
    public final int CIRCLE = 2;
    private DesignerGui gui;
    private RobotClient client;
    private int radius = 0;
    private int flag = 0;

    //para garantir que só correm após o espaçar formas
    private Semaphore semCircle;
    private Semaphore semSquare;

    public DrawSpaceForms(Semaphore sendMessage, Semaphore semCircle, Semaphore
semSquare, RobotClient client) {
        super(sendMessage);
        this.client = client;
        this.semCircle = semCircle;
        this.semSquare = semSquare;
        gui = new DesignerGui("Espaçar formas",1460,100);
        gui.setVisible();
    }

    @Override
    public void draw() {
        Message spaceForms;

        try {
```

```
        int space = 0;
        if (flag == SQUARE)
            space = client.spaceFormsSquare();

        else if (flag == CIRCLE)
            space = client.spaceFormsCircle(radius);

        spaceForms = new Line(space);
        client.sendMessage(spaceForms);
        gui.setTextToTextArea(gui.getTextFromTextArea() + "Reta: " + space +
"\n");

        Thread.sleep(spaceForms.sleepTime());

        Message stop = new Stop();
        client.sendMessage(stop);
        gui.setTextToTextArea(gui.getTextFromTextArea() + "Parar\n\n");
        Thread.sleep(stop.sleepTime());

        //libertar permissão quadrado ou circulo
        releaseForm(flag);

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

//que tipo de forma geometrica irá ser realizada
public void setTypeForm(int typeForm) {
    this.flag = typeForm;
```

```
}
```

```
//utilizado para saber o tamanho do próximo circulo para espaçar
```

```
//e incluir essa distância
```

```
public void setCircleRadius(int radius) {
```

```
    this.radius = radius;
```

```
}
```

```
// da release de uma das formas geometricas
```

```
private void releaseForm(int flag) {
```

```
    if (flag == SQUARE)
```

```
        semSquare.release();
```

```
    else if (flag == CIRCLE)
```

```
        semCircle.release();
```

```
}
```

```
}
```

```
import java.util.concurrent.Semaphore;

public class DrawSquare extends Draw {

    private final int ROTATION = 90;
    private int distance;
    private DesignerGui gui;
    private RobotClient client;
    private Semaphore semCircleSquare;
    private Semaphore semFileReadWrite;

    //o semaforo é para o draw e tem 0 permissões
    public DrawSquare(Semaphore sendMessage, RobotClient client, Semaphore
semCircleSquare, Semaphore semFileReadWrite) {
        super(sendMessage);
        this.client = client;
        this.semFileReadWrite = semFileReadWrite;
        this.semCircleSquare = semCircleSquare;
        gui = new DesignerGui("Desenhar quadrado",1000,100);
        gui.setVisible();
    }

    @Override
    public void draw() {
        try {
            Message reta = new Line(distance);
            Message stop = new Stop();
            //Para rodar deve ser o tamanho das rodas
            Message curve = new LeftCurve(10, ROTATION);
            //Message curve = new RightCurve(10, ROTATION);

            for (int i = 0; i < 4; i++) {
```



```

        client.sendMessage(reta);
        gui.setTextToTextArea(gui.getTextFromTextArea() + "Reta: " +
distance + "\n");

        Thread.sleep(reta.sleepTime());
        client.sendMessage(stop);
        gui.setTextToTextArea(gui.getTextFromTextArea() + "Parar\n");
        Thread.sleep(stop.sleepTime());
        client.sendMessage(curve);
        gui.setTextToTextArea(gui.getTextFromTextArea() + "Curva:
90º\n");

        Thread.sleep(curve.sleepTime());

    }

    client.sendMessage(stop);
    gui.setTextToTextArea(gui.getTextFromTextArea() + "Parar\n\n");
    Thread.sleep(stop.sleepTime());

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    semCircleSquare.release();
    semFileReadWrite.release();

}

//fazer set da distancia da reta
public void setDistance(int distance) {
    this.distance = distance;
}
}

```

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JRadioButton;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.ActionEvent;
import javax.swing.JTextField;
```

```
import javax.swing.JLabel;
import javax.swing.JOptionPane;
```

```
import java.awt.Color;
import java.awt.Font;
```

```
public class Gui {

    private App app;

    private JFrame frame;

    //botão para ligar o robo
    private JRadioButton turnOnBtn;

    //valores de distância e raio
    private JTextField distanceField;
    private JTextField radiusField;

    //botões das formas geometricas
    private JButton squareBtn;
    private JButton circleBtn;

    //textfield do outout
    private JTextField outputField;
    private JTextField robotName;

    public Gui(App app) {
        this.app = app;
        createLayout();
    }

    public void setVisible() {
        frame.setVisible(true);
    }
}
```

```
//escrever os comandos no campo do output
public void setTextToOutput(String txt) {
    outputField.setText(txt);
}

//vai buscar a distância em centímetros
public String getDistance() {
    if(distanceField.getText().matches("[0-9]+") &&
Integer.parseInt(distanceField.getText()) < 1001 && app.communicationOpened())
        return distanceField.getText();

    else if( distanceField.getText().matches("[0-9]+") &&
Integer.parseInt(distanceField.getText()) > 1000) {
        JOptionPane.showMessageDialog(frame,
            "O valor da distância não pode ser superior a 1000",
"Valores errados",
            JOptionPane.WARNING_MESSAGE);
        return null;
    }
    else if(!app.communicationOpened()) {
        JOptionPane.showMessageDialog(frame,
            "A comunicação com o robô não foi estabelecida", "Sem
comunicação",
            JOptionPane.WARNING_MESSAGE);
        return null;
    }
    else {
        JOptionPane.showMessageDialog(frame,
            "O valor da distância não é um número", "Valores
errados",
            JOptionPane.WARNING_MESSAGE);
        return null;
    }
}

//vai buscar a distância em graus
public String getRadius() {
    if(radiusField.getText().matches("[0-9]+") &&
Integer.parseInt(radiusField.getText()) < 1001 && app.communicationOpened())
        return radiusField.getText();

    else if(radiusField.getText().matches("[0-9]+") &&
Integer.parseInt(radiusField.getText()) > 1000) {
        JOptionPane.showMessageDialog(frame,
```

```
        "O valor da distância não pode ser superior a 1000",
"Valores errados",
        JOptionPane.WARNING_MESSAGE);
        return null;
    }
    else if(!app.comunicationOpened()) {
        JOptionPane.showMessageDialog(frame,
            "A comunicação com o robô não foi estabelecida", "Sem
comunicação",
            JOptionPane.WARNING_MESSAGE);
        return null;
    }

    else {
        JOptionPane.showMessageDialog(frame,
            "O valor da distância não é um número", "Valores
errados",
            JOptionPane.WARNING_MESSAGE);
        return null;
    }
}

public String getRobotName() {
    return robotName.getText();
}

public void setOpen(boolean flag) {
    turnOnBtn.setSelected(flag);
}

public void setButtonsActive(boolean flag) {
    squareBtn.setEnabled(flag);
    circleBtn.setEnabled(flag);
}

// cria o layout
private void createLayout() {
    frame = new JFrame();
    frame.setBounds(100, 100, 406, 275);
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.getContentPane().setLayout(null);
    frame.getContentPane().setBackground(new Color(170, 170, 170));
    frame.setTitle("Aplicação");

    frame.addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosing(WindowEvent e) {
            app.exitApp();
        }
    });
}
```

```
        frame.dispose();
        System.exit(0);
    }
});

squareBtn = new JButton("Quadrado");
squareBtn.setBounds(27, 123, 163, 38);
frame.getContentPane().add(squareBtn);

circleBtn = new JButton("Circulo");
circleBtn.setBounds(225, 123, 148, 38);
frame.getContentPane().add(circleBtn);

turnOnBtn = new JRadioButton("Ligar");
turnOnBtn.setBounds(326, 7, 58, 23);
frame.getContentPane().add(turnOnBtn);

JLabel outputTxt = new JLabel("Output");
outputTxt.setFont(new Font("Tahoma", Font.PLAIN, 13));
outputTxt.setBounds(27, 172, 46, 14);
frame.getContentPane().add(outputTxt);

outputField = new JTextField();
outputField.setBounds(27, 197, 349, 28);
outputField.setEditable(false);
frame.getContentPane().add(outputField);

distanceField = new JTextField();
distanceField.setBounds(27, 92, 163, 20);
frame.getContentPane().add(distanceField);

JLabel lblNewLabel = new JLabel("Comprimeto:");
lblNewLabel.setFont(new Font("Tahoma", Font.PLAIN, 12));
lblNewLabel.setBounds(27, 67, 86, 14);
frame.getContentPane().add(lblNewLabel);

radiusField = new JTextField();
radiusField.setBounds(225, 92, 148, 20);
frame.getContentPane().add(radiusField);

JLabel lblRaio = new JLabel("Raio:");
lblRaio.setFont(new Font("Tahoma", Font.PLAIN, 12));
lblRaio.setBounds(225, 67, 33, 14);
```

```

        frame.getContentPane().add(lblRaio);

        robotName = new JTextField();
        robotName.setBounds(104, 8, 86, 20);
        robotName.setText("EV8");
        frame.getContentPane().add(robotName);

        JLabel lblRobotName = new JLabel("Robot Name");
        lblRobotName.setFont(new Font("Tahoma", Font.PLAIN, 13));
        lblRobotName.setBounds(27, 11, 76, 14);
        frame.getContentPane().add(lblRobotName);

        //listener para quando se liga
        actionOnTurnOn();

        //action Listeners on click of forms
        actionOnSquare();
        actionOnCircle();
    }

    //listener para o click do radiobutton ligar
    private void actionOnTurnOn() {
        turnOnBtn.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if(turnOnBtn.isSelected())
                    app.openCommunication();

                else
                    app.closeCommunication();
            }
        });
    }

    //listener para o botão de desenhar quadrado
    private void actionOnSquare() {
        squareBtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                if(getDistance() != null) {
                    //tem de ser desativo aqui senão dá para clicar 2 vezes
                    antes que desativar o botão
                    app.setState(app.STATE_SQUARE);
                    app.releaseSemaphore();
                }
            }
        });
    }

```

```
    }

    //listener para o botão de desenhar círculo
    private void actionOnCircle() {
        circleBtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                if(getRadius() != null) {
                    app.setState(app.STATE_CIRCLE);
                    app.releaseSemaphore();
                }
            }
        });
    }
}
```

```
public class LeftCurve implements Message{

    public static final int LEFT_CURVE = 3;
    private final int COMPLETE_ROTATION = 360;
    private final int TIME_TO_CURVE = 1000;
    private int radius;
    private int angle;

    public LeftCurve(int radius, int angle) {
        this.radius = radius;
        this.angle = angle;
    }

    @Override
    public int getId() {
        return LEFT_CURVE;
    }

    @Override
    public int getDistance() {
        return 0;
    }

    @Override
    public int getRay() {
        return radius;
    }

    @Override
    public int getAngle() {
        return angle;
    }
}
```



```
// tempo que deve esperar por cada circulo
public int sleepTime() {
    if(radius == COMPLETE_ROTATION)
        return (2 * radius * (int) Math.PI) / 30 * 1000 + 200; // o 200 é para margem
de erro
    else
        return TIME_TO_CURVE;
}
}
```

```
public class Line implements Message {
```

```
    public static final int LINE = 1;
```

```
    // distancia que vai ser percorrida
```

```
    private int distance;
```

```
    public Line(int distance) {
```

```
        this.distance = distance;
```

```
    }
```

```
    @Override
```

```
    public int getId() {
```

```
        return LINE;
```

```
    }
```

```
    @Override
```

```
    public int getDistance() {
```

```
        return distance;
```

```
    }
```

```
    @Override
```

```
    public int getRay() {
```

```
        return 0;
```

```
    }
```

```
    @Override
```

```
    public int getAngle() {
```

```
        return 0;
```

```
    }
```

```
    @Override
```

```
    public int sleepTime() {  
        return (distance * 1000) / 30;  
    }  
  
}
```

```
public interface Message {  
  
    public int getId();  
  
    public int getDistance();  
  
    public int getRay();  
  
    public int getAngle();  
  
    public int sleepTime();  
  
}
```

```
public class RightCurve implements Message {

    public static final int RIGHT_CURVE = 2;
    private final int COMPLETE_ROTATION = 360;
    private final int TIME_TO_CURVE = 1000;
    private int radius;
    private int angle;

    public RightCurve(int raio, int angulo) {
        this.radius = raio;
        this.angle = angulo;
    }

    @Override
    public int getId() {
        return RIGHT_CURVE;
    }

    @Override
    public int getDistance() {
        return 0;
    }

    @Override
    public int getRay() {
        return radius;
    }

    @Override
    public int getAngle() {
        return angle;
    }
}
```

```
// tempo que deve esperar por cada circulo
public int sleepTime() {
    if (radius == COMPLETE_ROTATION)
        return (2 * radius * (int) Math.PI) / 30 * 1000 + 200; // o 200 é para margem
de erro
    else
        return TIME_TO_CURVE;
}
}
```

```
public class RobotClient {

    public final int STATE_DRAW_RECT = 1;
    public final int STATE_DRAW_CIRCLE = 2;

    private final int COMPLETE_ROTATION = 360;
    private final int INCREASE_SPACE = 10; // incremento por omissão espaço entre cada
forma
    private final int DEFAULT_SPACE = 0; // espaçamento por omissão (quando não existe
nenhuma forma anterior)

    private BufferCircular buffer;

    public RobotClient(BufferCircular buffer) {
        this.buffer = buffer;
    }
    //envia mensagens para o buffer
    public void sendMessage(Message m) {
        buffer.write(m);
    }

    // buscar o tamanho das formas anteriores para retornar o tamanho do espaçamento do
circulo
    public int spaceFormsCircle(int radius) {
        Message m = buffer.readLastMessage(2);
        Message m2 = buffer.readLastMessage(4);

        if (m != null) {
            if ((m.getId() == LeftCurve.LEFT_CURVE || m.getId() ==
RightCurve.RIGHT_CURVE) && m.getAngle() == COMPLETE_ROTATION)
                return m.getRay() + INCREASE_SPACE + radius;
        }
    }
}
```

```
    }

    if (m2 != null) {
        if (m2.getId() == Line.LINE)
            return m2.getDistance() + INCREASE_SPACE + radius;
    }
    return DEFAULT_SPACE;
}

//mesmo mas para o quadrado
public int spaceFormsSquare() {
    Message m = buffer.readLastMessage(2);
    Message m2 = buffer.readLastMessage(4);

    if (m != null) {
        if ((m.getId() == LeftCurve.LEFT_CURVE || m.getId() ==
RightCurve.RIGHT_CURVE) && m.getAngle() == COMPLETE_ROTATION)
            return m.getRay() + INCREASE_SPACE;// corresponde ao raio
mais mais distância segurança
    }

    if (m2 != null) {
        if (m2.getId() == Line.LINE)
            return m2.getDistance() + INCREASE_SPACE;// tamanho do lado
do quadrado mais distância segurança
    }

    return DEFAULT_SPACE;
}
```


}

}

```
public class RobotDesigner {

    private DesignerGui gui;
    private boolean isPhysical = false;
    private RobotLegoEV3 robot;

    public RobotDesigner(boolean isPhysical) {
        this.isPhysical = isPhysical;
        gui = new DesignerGui("Mensagens executadas",700,500);
        robot = new RobotLegoEV3();
        gui.setVisible();
    }

    public boolean OpenEV3(String robotName) {
        if(isPhysical) {
            setTextToGui("OpenEV3\n");
            return robot.OpenEV3(robotName);
        }
        else {
            setTextToGui("OpenEV3\n");
            return true;
        }
    }

    public void Reta(int distance) {
        if(isPhysical) {
            robot.Reta(distance);
            setTextToGui("Reta: " + distance+"\n");
        }
        else
            setTextToGui("Reta: " + distance+"\n");
    }
}
```

```
}
```

```
public void CurvarDireita(int raio, int angulo) {  
    if(isPhysical) {  
        setTextToGui("Curvar direita: " +raio + ", angulo: "+ angulo+"\n");  
        robot.CurvarDireita(raio, angulo);  
    }  
    else  
        setTextToGui("Curvar direita: " +raio + ", angulo: "+ angulo+"\n");  
}
```

```
public void CurvarEsquerda(int raio, int angulo) {  
    if(isPhysical) {  
        setTextToGui("Curvar esquerda: " +raio + ", angulo: "+ angulo+"\n");  
        robot.CurvarEsquerda(raio, angulo);  
    }  
    else  
        setTextToGui("Curvar esquerda: " +raio + ", angulo: "+ angulo+"\n");  
}
```

```
public void Parar(boolean flag) {  
    if(isPhysical) {  
        setTextToGui("Parar("+flag+")\n\n");  
        robot.Parar(false);  
    }  
    else  
        setTextToGui("Parar("+flag+")\n\n");  
}
```

```
public void CloseEV3() {  
    if(isPhysical) {  
        setTextToGui("Close");  
    }  
}
```

```
        robot.Parar(true);
    }
    else
        setTextToGui("Close");
}

private void setTextToGui(String txt) {
    gui.setTextToTextArea(gui.getTextFromTextArea()+txt);
}

}
```

```
import java.util.concurrent.Semaphore;

public class RobotServer implements Runnable{

    private BufferCircular buffer;
    private DesignerGui gui;
    private RobotDesigner robot;
    private StoreFile storeFile;

    //semáforo para garantir que são sempre escritos todos os comandos
    private Semaphore semWrite;

    //semáforo para garantir que só é executada as funções do servidor depois de ser escrito
    no ficheiro
    private Semaphore semFileWrite;

    public RobotServer(BufferCircular buffer, RobotDesigner robot, StoreFile storeFile) {
        this.buffer = buffer;
        gui = new DesignerGui("Servidor Robot",100,500);
        this.storeFile = storeFile;
        this.robot = robot;
        semWrite = new Semaphore(1);
        semFileWrite = new Semaphore(0);
        this.storeFile.setSemServer(semFileWrite);

        gui.setVisible();
    }

    // método bloqueante com semaforo que prossegue quando tem mensagens
    public void run() {
        for (;;) {
```

```
        Message m;
        if ((m = buffer.read()) != null)
            convertMessage(m);
    }
}

// função que interpreta a informação do buffer e age conforme a informação
private void convertMessage(Message m) {
    try {
        semWrite.acquire();
        int id = m.getId();

        //quando é para gravar formas
        if(storeFile.getState() == storeFile.STATE_RECORD) {

            storeFile.setContentToFile(id,m.getDistance(),m.getRay(),m.getAngle());
            semFileWrite.acquire();
        }

        if (id == Line.LINE) {
            robot.Reta(m.getDistance());
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Reta: " +
m.getDistance()+"\n");
        }

        else if (id == RightCurve.RIGHT_CURVE) {
            robot.CurvarDireita(m.getRay(), m.getAngle());
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Curvar
direita: " + m.getRay() + ", angulo:" + m.getAngle()+"\n");
        }
    }
}
```

```
        else if (id == LeftCurve.LEFT_CURVE) {
            robot.CurvarEsquerda(m.getRay(), m.getAngle());
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Curvar
esquerda: " + m.getRay() + ", angulo:" + m.getAngle()+ "º\n");
        }

        else if (id == Stop.STOP) {
            robot.Parar(false);
            gui.setTextToTextArea(gui.getTextFromTextArea() + "Parar\n");
        }
        semWrite.release();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

}

}
```

```
public class Stop implements Message{

    public static final int STOP = 4;
    private final int TIME_TO_STOP = 100;

    @Override
    public int getId() {
        return STOP;
    }

    @Override
    public int getDistance() {
        return 0;
    }

    @Override
    public int getRay() {
        return 0;
    }

    @Override
    public int getAngle() {
        return 0;
    }

    @Override
    public int sleepTime() {
        return TIME_TO_STOP;
    }

}
```

```
import java.awt.FileDialog;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.concurrent.Semaphore;

public class StoreFile implements Runnable {

    public final int STATE_SELECT_FILE = 0;
    public final int STATE_RECORD = 1;
    public final int STATE_STOP = 2;
    public final int STATE_PLAY = 3;
    private final String SEPARATE_SAME_MESSAGE = ";";
    private final String SEPARATE_DIFFERENT_MESSAGE = "#";

    private RobotDesigner robot;
    private File file;
    private StoreGui window;

    private int state;
    private Semaphore semaphore;
    private Semaphore semReadWrite;
    private Semaphore semServer;
    private BufferedReader reader;
    private BufferedWriter writer;
    private String content;
    private Semaphore semApp;
    private boolean fileChosen = false;
```

```
public StoreFile(RobotDesigner robot, Semaphore semApp) {
    this.robot = robot;
    this.semApp = semApp;
    semaphore = new Semaphore(0);
    semReadWrite = new Semaphore(1);
    window = new StoreGui(this, "Gravar comandos");
    window.setVisible();
}

@Override
public void run() {
    for (;;) {
        stateMachine();
    }
}

public void realeaseSemaphore() {
    semaphore.release();
}

public void setState(int state) {
    this.state = state;
}

public int getState() {
    return this.state;
}

// afetação no atributo para depois ser escrito ficheiro pela maquina de estados
public void setContentToFile(int id, int distance, int ray, int angle) {
    String result = "";
```

```
        switch (id) {
            case Line.LINE:
                result += id + SEPARATE_SAME_MESSAGE + distance +
SEPARATE_DIFFERENT_MESSAGE;
                break;
            case RightCurve.RIGHT_CURVE:
                result += id + SEPARATE_SAME_MESSAGE + ray +
SEPARATE_SAME_MESSAGE + angle + SEPARATE_DIFFERENT_MESSAGE;
                break;

            case LeftCurve.LEFT_CURVE:
                result += id + SEPARATE_SAME_MESSAGE + ray +
SEPARATE_SAME_MESSAGE + angle + SEPARATE_DIFFERENT_MESSAGE;
                break;

            case Stop.STOP:
                result += id + SEPARATE_DIFFERENT_MESSAGE;
                break;
        }
        content = result;
        semaphore.release();

    }

    // para fazer set do semaforo para poder quando acabar de executar libertar a
    // permissão
    public void setSemServer(Semaphore semServer) {
        this.semServer = semServer;
    }

    // diz se já foi seleccionado um ficheiro
    public boolean getFileChoosen() {
```

```
        return this.fileChosen;
    }

    // para a aplicação bdos comportamentos(clientes)
    public void setBehaviorSending(boolean flag) {
        window.setSendingMessage(flag);
    }

    // para a aplicação bloquear o botão quando está a enviar mensagens
    public void setEnableSelectBtn() {
        window.setEnableSelectBtn(true);
    }

    // para seleccionar o ficheiro
    private void getFile() {
        FileDialog fd = new FileDialog(window.getFrame(), "Open file",
FileDialog.LOAD);
        fd.setFile("*.txt");
        fd.setVisible(true);
        String filename = fd.getFile();
        String directory = fd.getDirectory();
        if (filename == null)
            System.out.println("You cancelled the choice");

        // para o caso de ter escolhido um ficheiro, se não existir cria 1
        else {
            this.file = new File(directory + getNameTxtFormat(filename));
            if (!file.exists()) {
                try {
                    file.createNewFile();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
        }
    }

    window.setTextToTextArea("Ficheiro selecionado: " +
getNameTxtFormat(filename) + "\n");
    window.onSuccessFileChosen();
    fileChosen = true;
    state = STATE_STOP;
}
}

// máquina de estados
private void stateMachine() {
    try {
        semaphore.acquire();
        switch (state) {
            case STATE_SELECT_FILE:
                getFile();
                break;

            case STATE_RECORD:
                writeToFile(content);
                window.setTextToTextArea(window.getTextFromTextArea() +
convertContentToText(content) + "\n");
                semServer.release();
                break;

            case STATE_STOP:
                break;

            case STATE_PLAY:
                String str = readFromFile();
```

```
        playCommands(str);
        break;
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}
}

// adiciona no ficheiro o conteudo passado como argumento
private void writeToFile(String content) {
    try {
        String previousMsg = readFromFile();// lê do ficheiro o que lá está

        semReadWrite.acquire();
        writer = new BufferedWriter(new FileWriter(file));
        writer.write(previousMsg + content);

        writer.close();
        semReadWrite.release();

    } catch (InterruptedException | IOException e1) {
        e1.printStackTrace();
    }
}

// lê o que foi escrito no ficheiro
private String readFromFile() {
    String result = "";
    try {
        semReadWrite.acquire();
        String txt = "";
```

```

        reader = new BufferedReader(new FileReader(file));
        while ((txt = reader.readLine()) != null) {
            result += txt;
        }
        reader.close();
        semReadWrite.release();
    } catch (InterruptedException | IOException e1) {
        e1.printStackTrace();
    }
    return result;
}

```

// a partir do conteúdo do ficheiro interpreta e manda executar

```

private void playCommands(String content) {
    if (content.contains(SEPARATE_DIFFERENT_MESSAGE)) {
        try {
            semApp.acquire();
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }
}

```

```

        String[] commands =
content.split(SEPARATE_DIFFERENT_MESSAGE);

```

```

        setTextToGui("REPRODUZIR:\n-----\n");
        for (int i = 0; i < commands.length; i++) {
            if (state != STATE_PLAY)
                break;

```

```

        String[] message;
        if (commands[i].contains(";"))

```

```

        message = commands[i].split(";");

        else {
            message = new String[1];
            message[0] = commands[i];
        }

        int id = Integer.parseInt(message[0]);

        selectCommand(message,id);

    }
    setTextToGui("-----\n");
    window.setEndPlay();
    semApp.release();
}
// quando o ficheiro não tem conteúdo no interior
else {
    setTextToGui("REPRODUZIR:\n-----\nSem Mensagens\n-----
-----\n");
    window.setEndPlay();
}
}

//faz executar o comando baseado no id
private void selectCommand(String[] message, int id) {
    switch(id)    {

        case Line.LINE:
            int distance = Integer.parseInt(message[1]);
            robot.Reta(distance);
            setTextToGui("Reta: " + distance + "\n");

```

```
    delay(new Line(distance).sleepTime());  
    break;
```

```
case RightCurve.RIGHT_CURVE:
```

```
    int radius = Integer.parseInt(message[1]);  
    int angle = Integer.parseInt(message[2]);  
    robot.CurvarDireita(radius, angle);  
    setTextToGui("Curvar direita: " + radius + ", " + angle + "\n");  
    delay(new RightCurve(radius, angle).sleepTime());  
    break;
```

```
case LeftCurve.LEFT_CURVE:
```

```
    int radius2 = Integer.parseInt(message[1]);  
    int angle2 = Integer.parseInt(message[2]);  
    robot.CurvarEsquerda(radius2, angle2);  
    setTextToGui("Curvar esquerda: " + radius2 + ", " + angle2 + "\n");  
    delay(new LeftCurve(radius2, angle2).sleepTime());  
    break;
```

```
case Stop.STOP:
```

```
    robot.Parar(false);  
    setTextToGui("Parar\n");  
    delay(new Stop().sleepTime());  
    break;
```

```
}
```

```
}
```

```
// converte os números em mensagens para afixar na gui
```

```
private String convertContentToText(String content) {
```

```
    String contentx = content.replace(SEPARATE_DIFFERENT_MESSAGE, "");
```

```
String[] separate = contentx.split(SEPARATE_SAME_MESSAGE);

if (Integer.parseInt(separate[0]) == Line.LINE) {
    return "Reta: " + separate[1];
}

else if (Integer.parseInt(separate[0]) == RightCurve.RIGHT_CURVE) {
    return "Curva direita: " + separate[1] + ", " + separate[2] + "°";
}

else if (Integer.parseInt(separate[0]) == LeftCurve.LEFT_CURVE) {
    return "Curva esquerda: " + separate[1] + ", " + separate[2] + "°";
}

else if (Integer.parseInt(separate[0]) == Stop.STOP) {
    return "Parar";
}

return null;
}

// para criar sempre um ficheiro formato txt
private String getNameTxtFormat(String fileName) {
    if (fileName.endsWith(".txt"))
        return fileName;

    else
        return fileName + ".txt";
}

private void delay(int timeMilliseconds) {
    try {
        Thread.sleep(timeMilliseconds);
    }
}
```

```
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    private void setTextToGui(String txt) {  
        window.setTextToTextArea(window.getTextFromTextArea()+txt);  
    }  
  
}
```

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.Color;
import java.awt.event.ActionEvent;

public class StoreGui {

    private StoreFile storeFile;
    private JFrame frame;
    private JTextArea receivedMessagesArea;

    private JButton playBtn;
    private JButton recordBtn;
    private JButton stopBtn;
    private JButton selectBtn;

    private boolean sendingMessage = false;

    public StoreGui(StoreFile storeFile, String windowName) {
        this.storeFile = storeFile;
        createLayout(windowName);
    }

    public void setVisible() {
        frame.setVisible(true);
    }
}
```

```
}

public void setTextToTextArea(String txt) {
    receivedMessagesArea.setText(txt);

    receivedMessagesArea.setCaretPosition(receivedMessagesArea.getDocument().getLength());
}

public String getTextFromTextArea() {
    return receivedMessagesArea.getText();
}

public JFrame getFrame() {
    return frame;
}

public void setEnablePlayBtn(boolean flag) {
    playBtn.setEnabled(flag);
}

public void setEnableSelectBtn(boolean flag) {
    selectBtn.setEnabled(flag);
}

//para fazer enable e disable do botão play dependendo se a correr ou não
public void setSendingMessage(boolean flag) {
    sendingMessage = flag;
    if(recordBtn.getBackground() != Color.ORANGE && !flag &&
storeFile.getFileChoosen())
        playBtn.setEnabled(true);
    if(flag)
```

```
        playBtn.setEnabled(false);
    }

    //quando seleciona com sucesso o ficheiro
    public void onSuccessFileChosen() {
        playBtn.setEnabled(true);
        recordBtn.setEnabled(true);
        selectBtn.setEnabled(false);
    }

    //quando acaba de executar os comandos
    public void setEndPlay() {
        playBtn.setEnabled(true);
        recordBtn.setEnabled(true);
        stopBtn.setEnabled(false);
        playBtn.setBackground(null);
        recordBtn.setBackground(null);
    }

    private void createLayout(String windowName) {
        frame = new JFrame();
        frame.setBounds(1300, 500, 450, 300);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.getContentPane().setLayout(null);
        frame.setTitle(windowName);

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                frame.dispose();
                storeFile.setState(storeFile.STATE_STOP);
            }
        })
    }
}
```

```
});

// adiciona uma textArea com scroll
JScrollPane sbrText = new JScrollPane();

sbrText.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

sbrText.setBounds(20, 20, 180, 220);
receivedMessagesArea = new JTextArea();
receivedMessagesArea.setEditable(false);
sbrText.setViewportView(receivedMessagesArea);
frame.getContentPane().add(sbrText);

playBtn = new JButton("Executar");
playBtn.setBounds(244, 15, 148, 36);
playBtn.setEnabled(false);
frame.getContentPane().add(playBtn);

recordBtn = new JButton("Gravar");
recordBtn.setBounds(244, 77, 148, 36);
recordBtn.setEnabled(false);
frame.getContentPane().add(recordBtn);

stopBtn = new JButton("Parar");
stopBtn.setBounds(244, 139, 148, 36);
stopBtn.setEnabled(false);
frame.getContentPane().add(stopBtn);

selectBtn = new JButton("Selecionar ficheiro");
selectBtn.setBounds(244, 204, 148, 36);
selectBtn.setEnabled(false);
frame.getContentPane().add(selectBtn);
```

```
        createActionOnPlay();
        createActionOnBtns(recordBtn, storeFile.STATE_RECORD);
        createActionOnStop();
        createActionOnBtns(selectBtn,storeFile.STATE_SELECT_FILE);
    }
```

```
private void setStateBtnsPlayRecord(boolean stop) {
    playBtn.setEnabled(stop);
    recordBtn.setEnabled(stop);
    stopBtn.setEnabled(!stop);
}
```

```
private void setColor(int state) {
    if(state == storeFile.STATE_PLAY) {
        playBtn.setBackground(Color.ORANGE);
        recordBtn.setBackground(null);
    }
    else if(state == storeFile.STATE_RECORD) {
        playBtn.setBackground(null);
        recordBtn.setBackground(Color.ORANGE);
    }
    else if(state == storeFile.STATE_STOP) {
        playBtn.setBackground(null);
        recordBtn.setBackground(null);
    }
}
```

```
private void createActionOnBtns(JButton btn,int state) {
    btn.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent arg0) {
            if(state != storeFile.STATE_SELECT_FILE)
                setStateBtnsPlayRecord(state == storeFile.STATE_STOP);
            setColor(state);
            storeFile.setState(state);
            if(state == storeFile.STATE_SELECT_FILE || state ==
storeFile.STATE_PLAY)

                storeFile.releaseSemaphore();

        }
    });
}

private void createActionOnStop() {
    stopBtn.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
            if (!sendMessage)
                playBtn.setEnabled(true);
            recordBtn.setEnabled(true);
            stopBtn.setEnabled(false);
            setColor(storeFile.STATE_STOP);
            storeFile.setState(storeFile.STATE_STOP);

        }
    });
}

private void createActionOnPlay() {
    playBtn.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
            playBtn.setEnabled(false);
            recordBtn.setEnabled(false);
```

```
        stopBtn.setEnabled(true);
        setColor(storeFile.STATE_PLAY);
        storeFile.setState(storeFile.STATE_PLAY);
        storeFile.releaseSemaphore();
    }
});
}
}
```