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

LEIM -Licenciatura Engenharia informática e multimédia

Fundamentos de Sistemas Operativos

Trabalho prático 1

Turma:

LEIM-31D-b

Trabalho realizado por:

Miguel Silvestre Nº45101

Miguel Távora Nº45102

Docente:

Carlos Carvalho

Data: 12/11/2020

Índice

1. INTRODUÇÃO 1
2. DESENVOLVIMENTO 3
2.1 ENQUADRAMENTO TEÓRICO 3
1.1 DEFINIÇÃO DE SISTEMA OPERATIVO 3
1.2 TIPOS DE SISTEMAS OPERATIVOS 3
1.3 CAMADAS DE CONCEÇÃO SISTEMA OPERATIVO 4
1.4 TECNOLOGIA UTILIZADA 4
1.5 DEFINIÇÃO DE PROCESSO 5
1.6 COMUNICAÇÃO ENTRE PROCESSOS COM MEMÓRIA PARTILHADA 6
2.2 IMPLEMENTAÇÃO TRABALHO 7
1.1 IMPLEMENTAÇÃO DA CLASSE CHATGUI 7
1.2 IMPLEMENTAÇÃO DA CLASSE MESSAGE 11
1.3.1 SINCRONIA NO ACESSO AO CANAL COMUNICAÇÃO 16
1.4 IMPLEMENTAÇÃO DA CLASSE CHAT 20
1.5 RESULTADOS OBTIDOS 26
3. CONCLUSÕES 29
4. BIBLIOGRAFIA 31
5. ANEXO 31

Índice ilustrações

Figura 1 - Hierarquia do sistema operativo 4
Figura 2 - Ciclo de vida de um processo 5
Figura 3 - Design da interface gráfica 7
Figura 4 - Método *set* do tipo de mensagem 8
Figura 5 - Método validação da mensagem 9
Figura 6 - Diagrama classes do ChatGUI 10
Figura 7 - Implementação classe Message 11
Figura 8 - Formato das mensagens 11
Figura 9 - Diagrama de classes da Message 12
Figura 10 - Método construtor onde instancia um FileChannel e MappedByteBuffer 13
Figura 11 - Método de sincronização dos índices das mensagens 14
Figura 12 - Método escrita no buffer 14
Figura 13 - Método leitura do buffer 15
Figura 14 - Método para limpar o buffer 16
Figura 15 - Método para fechar o FileChannel 16
Figura 16 - Método construtor da classe NumInstances 17
Figura 17 - Método *set* e *get* do número de instancias 18
Figura 18 - Diagrama de classes do canal comunicação 19
Figura 19 - Método run da classe Chat 20
Figura 20 - Classe RunProgram 20
Figura 21 - Diagrama atividades da aplicação 21
Figura 22 - Funcionamento do estado Comunicar entre processos 23
Figura 23 - Método para selecionar o ficheiro 24
Figura 24 - Diagrama de classes do chat 25
Figura 25 - Diagrama de classes da implementação completa 26
Figura 26 - Swimlanes obtidas 27

1. Introdução

O primeiro trabalho prático tem como objetivo principal a introdução aos processos que correm no sistema operativo. Atualmente as máquinas são todas multi-processo, quer isto dizer que cada máquina pode ter mais do que um processo a correr em simultâneo, onde todas elas competem pela posse do CPU. Quem faz a gestão entre qual será o precesso que deve aceder ao CPU num determinado instante temporal é o sistema operativo.

Para estudar os processos será utilizada uma aplicação que suporta um *chat* entre diferentes processos Java dentro da mesma máquina. O mecanismo de comunicação vai funcionar de tal maneira que após o envio de uma mensagem todas as outras aplicações de *chat* recebem essa mesma mensagem.

A aplicação será realizada na linguagem de programação Java. A comunicação entre os diferentes processos Java é feita à custa de memória partilha. Para realizar esta comunicação em forma de memória partilhada será utilizada a classe Java MappedByteBuffer. Esta classe é um buffer de bytes cujo conteúdo é uma região de memória de um ficheiro.

Para apresentar as mensagens recebidas e enviar mensagens para as diferentes processos será desenvolvida uma interface gráfica utilizando o editor gráfico *Window Builder*. O *Window Builder* por sua vez utiliza o *widget Swing* do Java, que possui um conjunto de classes que permite criar uma interface gráfica através de código Java. Por sua vez a interface terá uma zona para escrever as mensagens e também uma zona de exibição das mensagens recebidas.

2. Desenvolvimento

2.1 Enquadramento teórico

1.1 Definição de sistema operativo

Um sistema operativo é um programa que gere o *hardware*, permite a execução de outros programas e fornece serviços comuns a programas de computador. O sistema operativo é o programa mais utilizado em qualquer computador.

O sistema operativo tem o objetivo de facilitar e maximizar a gestão de recursos *hardware*. Desta forma é possível criar um nível de abstração ao utilizador designado por máquina virtual. Sem este nível de abstração o utilizador do computador necessitaria de programar ao nível da máquina, esta tarefa é complicada e consome uma grande quantidade de tempo.

1.2 Tipos de sistemas operativos

Sistemas operativos de mono-utilizador permitem aceder aos dados pessoais do utilizador, mas não permitem a existência de multiplos perfís. Foi através deste tipo de sistema operativo que surgiram os primeiros computadores pessoais.

Apesar da variedade de sistemas operativos existentes, os que serão abordados no presente trabalho são nomeadamente de multi-utilizador, multi-processo e multi-programa.

- Sistemas operativos multi-utilizador são os que permitem multiplos utilizadores terem acesso ao mesmo computador.
- O multi-processo fornece a capacidade de executar multiplas tarefas do programa por um único utilizador ao mesmo tempo no mesmo computador.
- Multi-programa fornece a capacidade de permitir diversas aplicações concorrentes

onde todas querem executar, no qual o sistema operativo é o responsável por gerir a sua efetividade e eficiência.

1.3 Camadas de conceção sistema operativo

O sistema operativo possui camadas onde é possível ter uma abordagem modular e hierarquica da sua complexidade. A abordagem ao sistema operativo será feita seguindo a seguinte hierarquia:

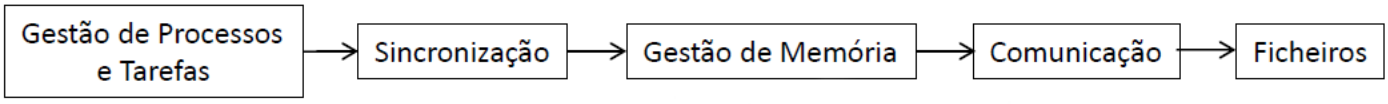

Figura 1 - Hierarquia do sistema operativo

Apesar de cada camada estar disjusta das restantes camadas, estas não estam inteiramente disjuntas quando fala de realização prática do sistema operativo.

1.4 Tecnologia utilizada

Para desenvolver a aplicação pretendida será utilizada a linguagem de programação Java. Desta forma o sistema operativo e as suas camadas serão exploradas utilizando a linguagem de programação. O Java acenta na *Java Virtual Machine (JVM)* desta forma o código fica portável para diversos *hardwares*, desde computadores a dispositivos móveis.

1.5 Definição de processo

Um processo é um programa em execução, a execução de um processo deve progredir de forma sequêncial. Um processo é definido como uma entidade que realiza um determinado trabalho implementado pelo sistema. Quando um programa é carregado para a memória ele passa a ser um processo.

Um processo possui um ciclo de vida que passa por diversos estados. Estes estados não são *standart* e podem divergir entre sistemas operativos.

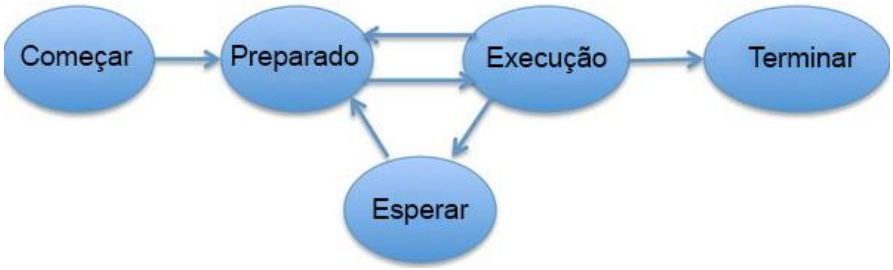

Figura 2 - Ciclo de vida de um processo

- Estado começar – é quando começou ou foi criado.
- Estado preparado – o processo está a espera que o sistema operativo lhe atribua o processador.
- Estado execução – quando o processo obtem o processador pelo *scheduler* e o processador executa as suas instruções.
- Estado espera – processo é movido para o estado de espera porque precisa de um recurso, como esperar o *input* do utilizador ou um ficheiro ficar disponível.
- Estado terminar – quando o programa termina a sua execução ou é terminado pelo sistema operativo é movido para o estado terminar, onde este espera ser removido da memória.

1.6 Comunicação entre processos com memória partilhada

A comunicação entre processos Java independentes na mesma máquina pode ser feito através do mecanismo de memória mapeada. Esta metodologia mapeia numa área de memória virtual o conteúdo de um ficheiro, onde a memória pode ser manipulada como um buffer comum.

A classe que suporta este conceito designa-se MappedByteBuffer, onde uma instância desta classe é feita através de um FileChannel.map e representa um buffer de acesso direto igual ao de um ficheiro. Qualquer processo Java pode alterar o conteúdo do ficheiro através da classe MappedByteBuffer.

2.2 Implementação trabalho

1.1 Implementação da classe ChatGUI

A classe ChatGui é a classe que cria a interface gráfica do *chat*. Através desta interface gráfica o utilizador realiza as ações pretendidas e obtém os resultados dessas mesmas ações.

Para a criação desta interface gráfica foi utilizado o *WindowBuilder*, esta ferramenta permite criar interfaces gráficas de forma simples sem perder muito tempo a escrever código. Através de operações de *drag-and-drop* esta ferramenta gera automaticamente o código que corresponde á componente selecionada.

O aspeto gráfico da interface é o que se segue:

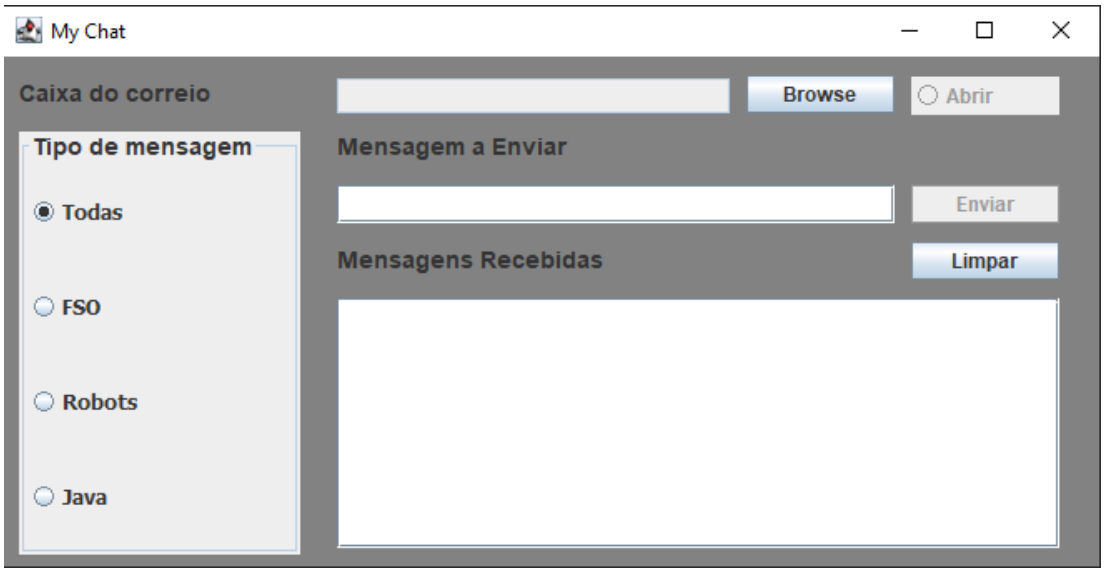

Figura 3 - Design da interface gráfica

Esta interface possui oito elementos principais para a interação com o utilizador. Essas componentes são nomeadamente: caixa de correio que corresponde ao nome do ficheiro que será utilizado para comunicar, botão de *browse* que abre um FileDialog para selecionar o ficheiro, o *radiobutton* que abre o canal de comunicação, caixa de texto para escrever a mensagem para enviar, botão enviar que envia a mensagem, o botão limpar que limpa todas as mensagens recebidas, o *textarea* que exibe todas as mensagens recebidas e o conjunto de *radiobuttons* que dita o tipo de mensagem que será enviada ou recebida.

Esta classe funciona de tal maneira que disponibiliza os métodos respetivos para as operações da interface, nomedamente fazer *set* do texto na caixa de texto, deteção de cliques nos botões entre outros. No tipo de mensagem só é possível ter um *radiobutton* selecionado no mesmo instante temporal e para saber qual dos botões está selecionado num dado instance, estes possuem *actionListeners* que fazem *set* do atributo typeMessage. Para a classe que gere a parte de negócio saber o seu valor existe um método *get* do atributo.

```
// faz set do tipo de mensagem conforme se clica nos radio buttons
private void setTypeMessageByRadioBtns() {
    allRadio.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            typeMessage = 0;

        }
    });

    fsoRadio.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            typeMessage = 1;

        }
    });

    robotsRadio.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            typeMessage = 2;

        }
    });

    javaRadio.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            typeMessage = 3;

        }
    });

}
```

Figura 4 - Método *set* do tipo de mensagem

Quando é enviada uma mensagem é feita uma validação ao conteúdo da mensagem, esta validação é feita através da classe ValidateData. A validação é nomeadamente para as mensagens enviadas não poderem ser nulas, não podem conter somente espaços, tem de ter pelo menos um caracter e só podem ter no máximo 256 caracteres. A classe ValidateData possui somente um método de validação podendo desta forma ficar na classe ChatGui, contudo desta maneira o código fica mais organizado e mais simples de posteriormente expandir.

```
public String validateMessage(String message) {
    if(message.trim().equals("") || message == null)
        return "A mensagem enviada não pode estar vazia";

    else if(message.length() < 1)
        return "A mensagem tem de ter pelo menos 1 caracteres";

    else if(message.length() > 256)
        return "A mensagem só pode conter no máximo 256 caracteres";

    return null;

}
```

Figura 5 - Método validação da mensagem

O diagrama de classes obtido para a implementação foi o que se segue:

Figura 6 - Diagrama classes do ChatGUI

A classe responsável por apresentar a interface gráfica é a classe ChatGUI esta classe estende de Jframe e utiliza diversas outras classes para criar botões entre outros elementos gráficos. Estas não foram exibidos no diagrama de classes pois aumentaria muito a complexidade do diagrama sem necessidade. Esta classe também utiliza outra classe designada por ValidateData para validar a mensagem para enviar.

1.2 Implementação da classe Message

A classe Message é a classe responsável por fazer o encapsulamento dos dados tanto de escrita como de leitura. Para realizar este encapsulamento é feito através dos argumentos do construtor da classe e *getters* para ler o seu valor.

```
//classe responsavel por armazenar os três campos da mensagem
public class Message {

  private int id;//contador que é incrementado cada vez que é enviada uma mensagem a começar em 0
  private int type;//valor que define para quem deve ser entrege a mensagem || 0 = todos, 1 = FSO, 2 = robots, 3 = Java
  private String text;//mensagem enviada

  public Message(int id, int type, String text) {
      this.id = id;
      this.type = type;
      this.text = text;
  }

  public int getId() {
      return this.id;
  }

  public int getType() {
      return this.type;
  }

  public String getText() {
      return this.text;
  }

}
```

Figura 7 - Implementação classe Message

Os argumentos do construtor desta classe corresponde á informação de uma única mensagem. Cada mensagem possui o seguinte formato:

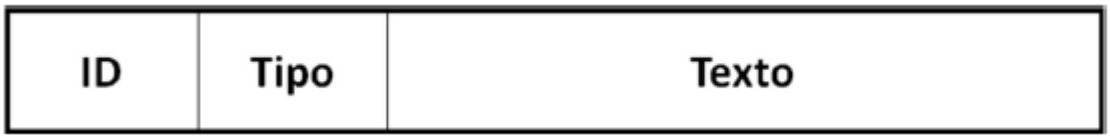

Figura 8 - Formato das mensagens

O campo ID corresponde ao índice da mensagem, este valor é do tipo inteiro não negativo iniciado a 0. O Tipo que permite filtrar as mensagens de recepção/envio, onde o seu valor é ditado pelo conjunto de *radiobuttons* do tipo de mensagem selecionado. Por fim o Texto é uma *string* que corresponde ao texto escrito pelo utilizador no campo de mensagem a enviar.

O diagrama de classes da classe Message obtido foi o seguinte:

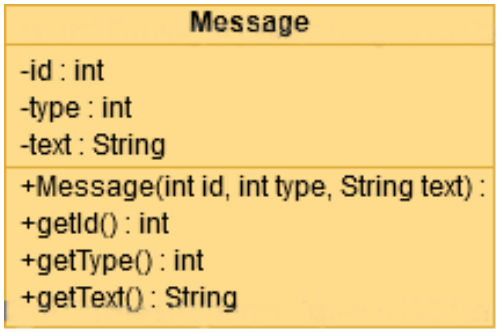

Figura 9 - Diagrama de classes da Message

1.3 Implementação da classe CommunicationChannel

A classe CommunicationChannel é responsável por estabelecer a comunicação entre os diferentes processos. Este mecanismo é feito através da leitura e escrita de dados num ficheiro. Como está a ser utilizada memória mapeada os dados são na verdade bytes e dessa forma são então escritos e lidos bytes.

Para criar a memória mapeada é necessário um FileChannel, para isso é criado uma instância de RandomAccessFile que permite aceder ao conteúdo de um ficheiro de forma aleatória e não sequêncial. Da classe RandomAccessFile utiliza-se o método getChannel que retorna um FileChannel. A partir deste FileChannel cria-se uma instância de MappedByteBuffer que permite meter inserir bytes no ficheiro, este ficheiro é passado como argumento da instância da classe RandomAccessFile.

```
@SuppressWarnings("resource")
public ComunicationChannel(String filePath, String fileName) {
    file = new File(filePath+fileName);
    instances = new NumInstances(filePath,fileName);
    try {
        channel = new RandomAccessFile(file, "rw").getChannel();
        buffer = channel.map(FileChannel.MapMode.READ_WRITE, 0, MAX_SIZE*8);//para bytes

    } catch (IOException e) {
        e.printStackTrace();
    }
    getNumInstances(true);
}
```

Figura 10 - Método construtor onde instancia um FileChannel e MappedByteBuffer

Quando se cria o MappedByteBuffer é necessário referir o tamanho ao qual corresponderá o buffer. No caso presente corresponde a dois inteiros, onde cada um corresponde a 4 bytes e também uma *string* de tamanho variável. Como uma *string* é um conjunto de chars onde cada char são 2 bytes então no máximo existe 256 caracteres vezes 2 bytes. No total dá o valor de 520 que é o valor da variável MAX_SIZE.

Visto que um processo quando se conecta a um *chat* já podem ter existido mensagens anteriores, a classe disponibiliza um método que assim que é criada uma instância de CommunicationChannel é feito o *set* do atributo messageId para o índice da mensagem corrente, caso não exista o índice fica a -1. Desta forma é garantido que todos os processos que participam no mesmo *chat* possuem sincronia entre os índices das mensagens.

```
//fazer set para o indice correto das mensagens, quando nao tem fica a -1
public synchronized void startBufferOnEmpty() throws IOException {
    FileLock f = channel.lock(0, channel.size(), false);

    // obter o primeiro caracter da mensagem, caso seja != 0 é porque já existe uma
    // mensagem previa
    int ix = buffer.getChar(4 + 4);
    if (ix != 0) {
        int id = buffer.getInt(0);
        System.out.println("id: " + id);
        messageId = id + 1;
    } else
        System.out.println("Sem mensagem");
    f.release();

}
```

Figura 11 - Método de sincronização dos índices das mensagens

Para realizar a leitura a e a escrita do conteúdo no ficheiro existem dois métodos nomeadamente o writeMessage() e o readMessage(Message m).

O método *write* recebe como argumento uma instância de mensagem ao qual vai buscar os seus valores e coloca esses valores no ficheiro através dos métodos putInt() e putChar(). Contudo é necessário garantir que a mensagem corrente não é corrompida pela mensagem anterior para isso é primeiro limpo os bytes do ficheiro. Quando se inicia a escrita é feito o *set* da posição no valor 0 através do método position.

```
// mete bytes no buffer respetivos da mensagem
public synchronized void writeMessage(Message m) throws IOException {
    FileLock f = channel.lock(0, channel.size(), false);
    clear();// limpa a mensagem para nao aparecerem caracteres de outras mensagens
    buffer.position(0);
    buffer.putInt(m.getId());
    buffer.putInt(m.getType());
    for (int i = 0; i < m.getText().length(); i++) {
        char c = m.getText().charAt(i);
        buffer.putChar(c);
    }
    f.release();
}
```

Figura 12 - Método escrita no buffer

O método *read* lê o conteúdo que está no buffer, a leitura é feita de tal maneira que se o atributo messageId for igual ao valor do id que está no ficheiro é porque existe mensagem, isto porque o messageId é sempre um valor superior ao valor corrente do ficheiro. Como a contagem começa a 0 e por *default* o valor inteiro também é 0, é validado se existe algum caracter no ficheiro, visto que não existem mensagens vazias. Desta forma é detetado se existe alguma mensagem nova. Caso exista é feito a leitura dos bytes através do método getInt e getChar. Da mesma maneira que o *write* também é preciso fazer *set* da posição do buffer para o valor 0.

```
//verifica se existe mensagem, caso exista retorna uma instancia de message
public synchronized Message readMessage() throws IOException {
    FileLock f = channel.lock(0, channel.size(), false);
    buffer.position(0);
    int newId = buffer.getInt(0);
    int c = buffer.getChar(4 + 4);// verificar se o inteiro corresponde ao char recebido não é null

    // mensagem no buffer tem o mesmo indice de escrita do proximo ou caso o
    // primeiro caracter por n poder ser null ser != null
    if (newId == messageId || (messageId == -1 && c != 0)) {
        int type = buffer.getInt(4);// '0

        String text = "";
        int index = 0;
        // loop para buscar os chars guardados no buffer
        while (true) {
            char cx = buffer.getChar(4 + 4 + index * 2);
            int ix = cx;
            index++;
            if (ix != 0)
                text += cx;
            else
                break;
        }

        Message m = new Message(newId, type, text);

        if (messageId != -1)
            messageId++;
        else
            messageId += 2;

        f.release();
        return m;
    }
    f.release();

    return null;

}
```

Figura 13 - Método leitura do buffer

1.3.1 Sincronia no acesso ao canal comunicação

Uma parte fundamental que existe em ambos os métodos leitura, escrita e também no método de iniciar o índice é o FileLock. Este mecanismo serve para garantir que nenhum processo está a ler ao mesmo tempo que outro método está a escrever, obtendo desta forma sincronia entre a informação. O FileLock assim que adquire um recurso bloqueia esse recurso para os restantes processos, quando outro processo tenta aceder a esse recurso fica na fila de espera até ser libertado pelo processo detentor do recurso. A aquisição do recurso é feito pelo método *lock* e a libertação é feita pelo *release*, sempre que existe um *lock* deve haver um *release* senão o recurso nunca é libertado e mais nenhum processo pode aceder ao recurso.

Sempre que é escrita uma nova mensagem, como o tamanho da mensagem pode variar é possível que no envio de uma mensagem esta apanhe bytes da mensagem anterior. Para evitar isso é feita uma limpeza no conteúdo do ficheiro pondo todo o seu conteúdo vazio com o método clear().

```
//limpa os bytes armazenados no ficheiro
public void clear() {
    for (int i = 0; i < MAX_SIZE * 8; i++) {
        buffer.put(i, (byte) 0);
    }
}
```

Figura 14 - Método para limpar o buffer

Sempre que um processo é terminado é importante fechar o FileChannel para isso foi criado o método closeChanel().

```
//fecha o FileChannel
public void closeChannel() {
    try {
        channel.close();
        getNumInstances(false);
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

Figura 15 - Método para fechar o FileChannel

Uma funcionalidade extra implementado pelo grupo é sempre que todos os processos que estão a realizar a comunicação sobre um dado ficheiro são terminados o conteúdo existente no ficheiro é apagado. Esta funcionalidade é feita através de uma classe designada NumInstances. A funcionalidade desta classe é que no momento da seleção do ficheiro utilizado para comunicar é criado um ficheiro txt na mesma diretoria e com o mesmo nome. Nesse ficheiro é escrito em formato de *string* o número de instâncias que existem para um dado ficheiro de comunicação. Desta forma a classe disponibiliza métodos de leitura e escrita em ficheiros e também um método de conversão para obter o mesmo nome do ficheiro selecionado para ser mapeado o conteúdo mas com extensão txt.

```
public NumInstances(String directory, String fileName) {
    this.directory = directory;
    this.fileName = getFileName(fileName)+".txt";

    File file = new File(directory+this.fileName);
    if (!file.exists()) {
        try {
            file.createNewFile();
            writeFile("0");//precisa ter o valor 0 senao da erro
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    this.file = new File(directory+this.fileName);
}
```

Figura 16 - Método construtor da classe NumInstances

O método que interage com esta classe para incrementar e decrementar na classe CommunicationChannel é o método getNumInstances(), onde existe um argumento booleano aumentar ou diminuir o número de instâncias. Existe também um *get* para obter o valor do número de instâncias da classe. Quando o número de instâncias é igual a 0 todo o conteúdo do ficheiro é limpo.

```
//limpa os bytes armazenados no ficheiro
public void clear() {
    for (int i = 0; i < MAX_SIZE * 8; i++) {
        buffer.put(i, (byte) 0);
    }
}

//obtem o número de instancias criadas do canal comunicação para depois limpar o ficheiro quando é 0
public int getNumInstances() {
    return this.numInstances;
}

private void getNumInstances(boolean increase) {
    int previous = instances.readFile();

    if (previous != -1) {
        if (increase)
            previous++;
        else
            previous--;
        instances.writeFile(""+previous);
        numInstances = previous;
    } else {
        previous = 1;
        instances.writeFile(""+previous);
        numInstances = previous;
    }

}
```

Figura 17 - Método *set* e *get* do número de instancias

O diagrama de classes do canal de comunicação é o que se segue:

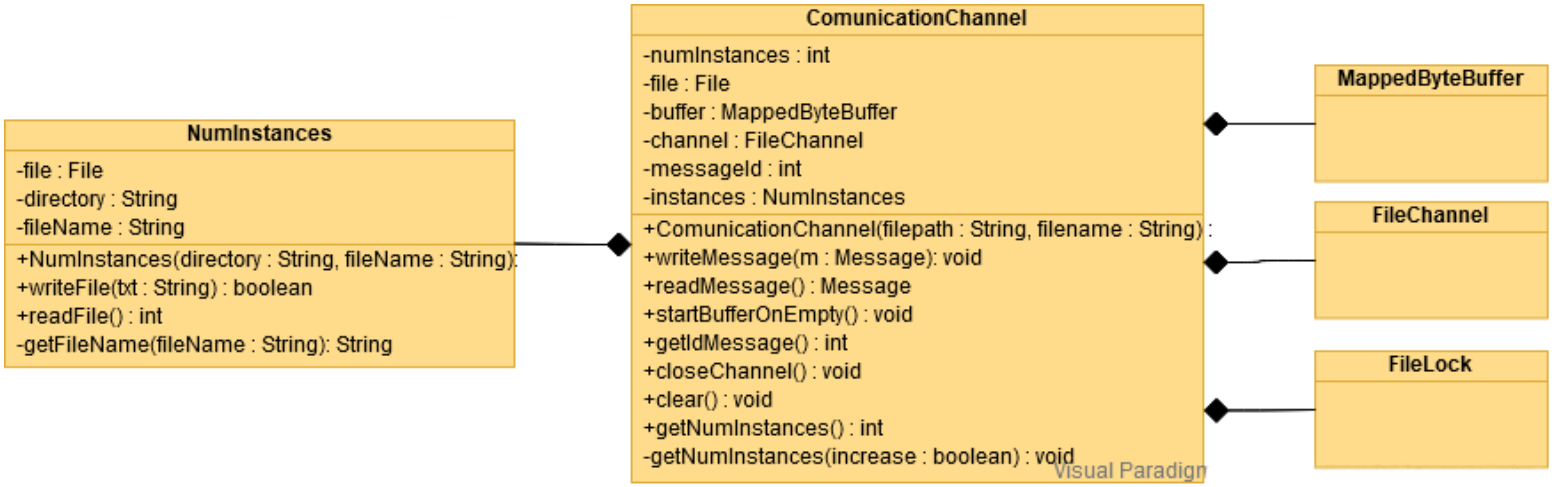

Figura 18 - Diagrama de classes do canal comunicação

A classe que realiza a comunicação entre os processos é a CommunicationChannel e para isso utiliza a classe FileChannel, MappedByteBuffer e também o FileLock para garantir a coerência da informação.

1.4 Implementação da classe Chat

A classe Chat é a classe principal responsável por realizar todo o modelo de negócio da aplicação. Como esta classe é responsável pela gestão da aplicação esta possui atributos de ChatGUI e também de CommunicationChannel.

O modelo funcional desta aplicação é feito através de não acabar a Thread do método main ficando num ciclo infinito até ao utilizador terminar o programa. Dentro deste ciclo infinito existe uma condição de paragem para acabar o ciclo infinito da máquina de estados responsável por executar a lógica do programa.

```
// metodo para correr o prop
public void run(){
    do {
        stateMachine();

    } while (executeProgram);

}
```

Figura 19 - Método run da classe Chat

Para correr o programa foi utilizado uma classe RunProgram que possui somente o método main. Isto foi feito para possuir um código mais estruturado e organizado, reduzindo assim a sua complexidade.

```
public class RunProgram {

    public static void main(String[] args) {
        Chat chat = new Chat();
        chat.run();
    }

}
```

Figura 20 - Classe RunProgram

Para a classe Chat conseguir todo o funcionamento da aplicação foi utilizada uma máquina de estados. Essa máquina de estados é que dita todo o procedimento que a aplicação segue.

A máquina de estados funciona como se apresenta no seguinte diagrama de atividades:

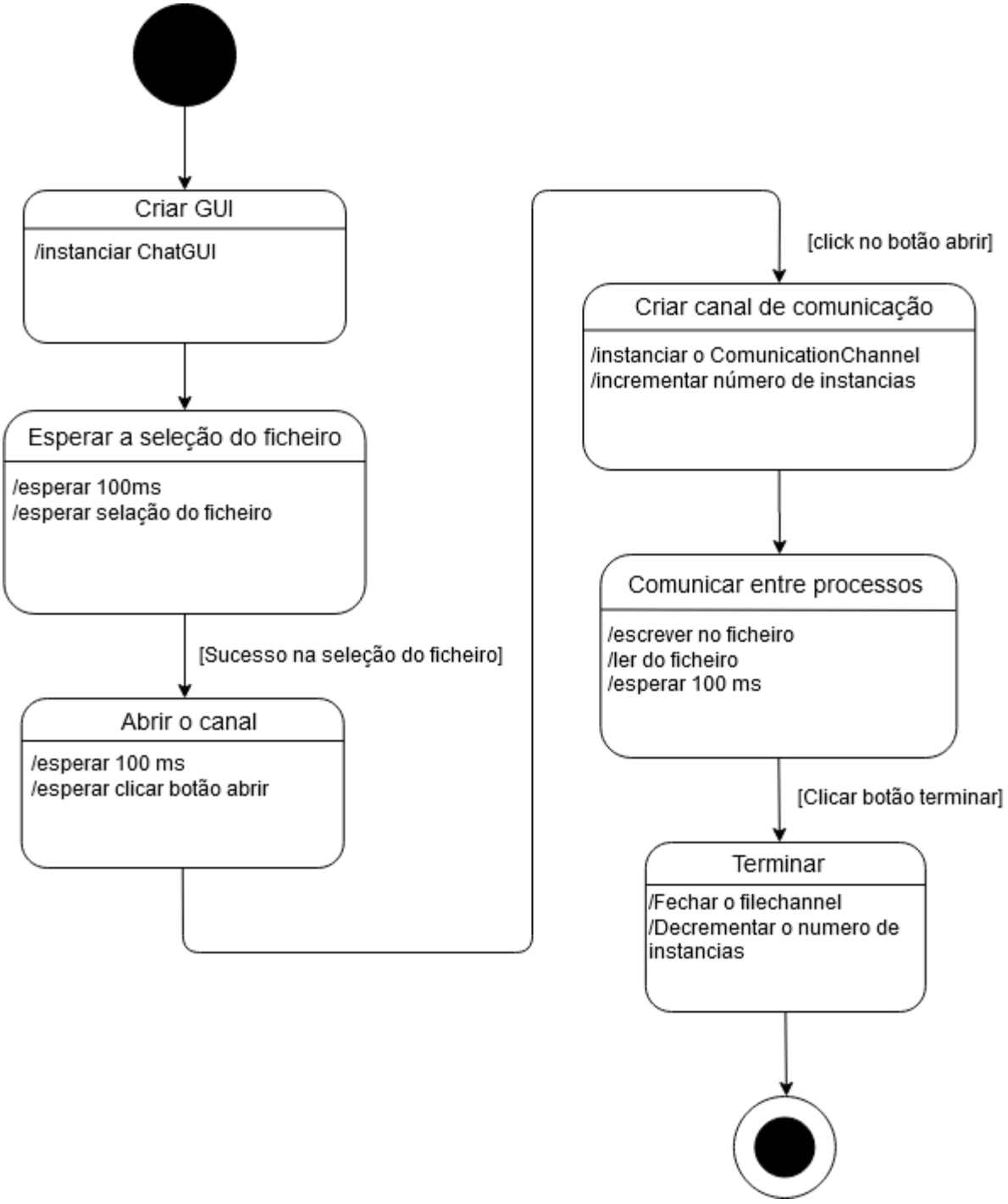

Figura 21 - Diagrama atividades da aplicação

A aplicação para funcionar inicialmente cria uma instância de ChatGUI. A partir dessa instância é criada a interface gráfica, onde o utilizador pode interagir. Como os botões de envio e de abrir o canal estão desativos antes da seleção do ficheiro, o utilizador só pode fazer *browse* para selecionar o ficheiro ou alterar o tipo de mensagens. Após criada a instância do ChatGUI a máquina de estados transita automáticamente para um estado de espera de 100 em 100 milisegundos onde espera que o utilizador selecione o ficheiro pela qual quer enviar as mensagens. Esta espera de 100 milisegundos é muito importante porque se o programa estiver sistematicamente a correr, todas as outras funcionalidades deixam de funcionar porque um processo fica com a posse total da máquina física congelando todas as outras até que esta liberte a posse do CPU. Quando o utilizador seleciona um ficheiro ou cria um novo o estado transita para um novo estado onde a aplicação fica também em espera, quando é clicado no botão abrir é aberto o canal de comunicação. Após o canal comunicação ser aberto é criada uma instância de CommunicationChannel e também é incrementado o número de instâncias existentes no ficheiro txt sobre um dado ficheiro dat ou com outra extensão. Por sua vez o estado de criação do canal transita automáticamente para o estado onde já é possível estabelecer comunicação entre os diversos processos. Esta comunicação é feita a partir de três estados da máquina de estados, são nomeadamente o estado de leitura do ficheiro, o estado de escrita e o estado de espera. Quando o utilizador clica no X para terminar a aplicação este transita para o estado terminar que fecha o canal de comunicação e decrementa o número de instâncias, quando o número de instâncias é zero o conteúdo do ficheiro é limpo. Qualquer estado a partir do estado [Esperar a seleção do ficheiro] até [Comunicar entre processos] pode transitar para o estado Terminar, contudo quando não é criado o canal comunicação este não é fechado.

O funcionamento do estado estado Comunicar entre processos no diagrama de atividades funciona da seguinte maneira:

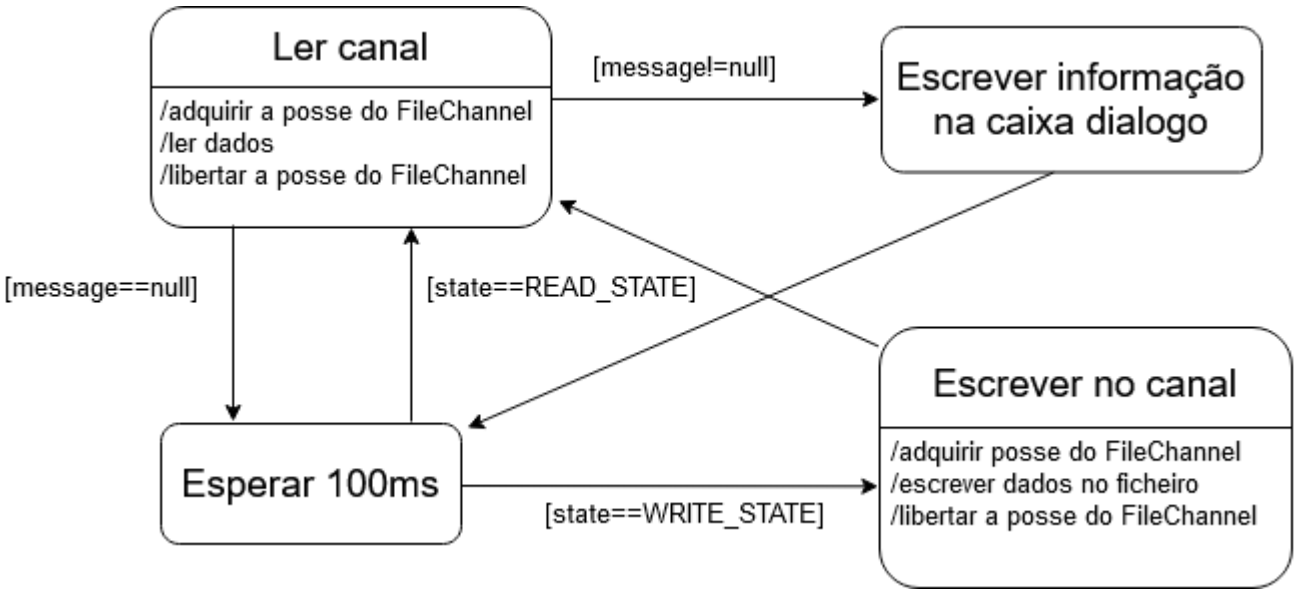

Figura 22 - Funcionamento do estado Comunicar entre processos

Dos seguintes estados a aplicação está maioritariamente no estado de espera para não subcarregar o CPU, o tempo de espera é sempre de 100 milisegundos. Quando passa 100 milisegundos caso não haja alterações no estado, como o envio de mensagens, a aplicação transita para o estado de leitura. Caso exista uma mensagem nova para ser lida o método *read* retorna um objeto do tipo Message, onde os seus dados são escritos na caixa diálogo designada por mensagens rebecidas na interface gráfica. Quando a aplicação acaba de ler o ficheiro ou escrever na caixa diálogo volta para o estado de espera. A transição para o estado de escrita só é feito quando se clica enviar e a mensagem é válida, a transição é sempre feita do estado de espera para o estado de escrita. Após escrever a mensagem no ficheiro é feita uma transição para o estado de leitura, pois existe uma nova mensagem para ser lida, e volta para a transição entre leitura e espera.

Uma parte importante na utilização da aplicação é a seleção do ficheiro, isto foi realizado na classe Chat através da classe FileDialog. O FileDialog permite criar e selecionar um ficheiro de forma automática e obter tanto o caminho como o nome do ficheiro.

```
// permite selecionar um ficheiro ou criar um dinâmicamente
public void browseMailMessages() {
    FileDialog fd = new FileDialog(gui, "Open file", FileDialog.LOAD);
    fd.setFile("*.dat");
    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 dat
    else {
        System.out.println(directory + filename);
        pathFile = directory;
        this.fileName = filename;
        gui.setMailText(filename);
        gui.enableOpenButton();
    }

}
```

Figura 23 - Método para selecionar o ficheiro

O diagrama de classes do Chat é o que se segue:

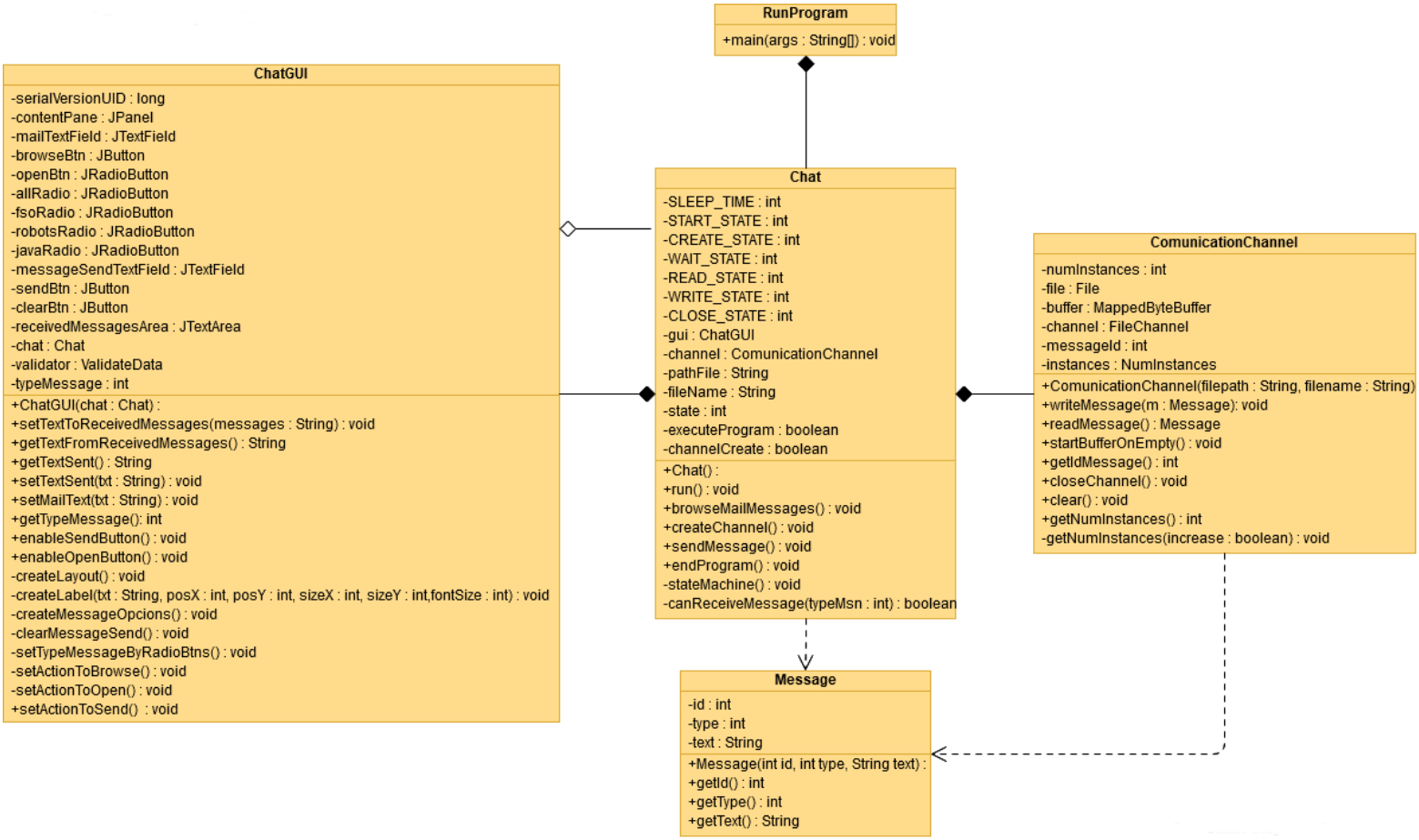

Figura 24 - Diagrama de classes do chat

O diagrama de classes possui três módulos principais a classe Chat, a classe CommunicationChannel e o ChatGUI. A classe Chat é responável por toda a lógica de negócio da aplicação e para isso precisa de uma interface gráfica que é a classe ChatGUI e de um canal de comunicação para comunicar entre processos que é a classe CommunicationChannel. O modo de transmitir informação entre a classe ComunicationChannel e o Chat é através de instâncias da classe Message.

1.5 Resultados obtidos

Perante a implementação feita pelo grupo o trabalho ficou todo funcional com mais do que dois processos a correr simultâneamente e também permite *chats* diferentes a correr simultaneamente. A parte extra de contar o número de instâncias para eliminar o conteúdo antigo do ficheiro também ficou funcional, contudo o acesso a este ficheiro não é feito através de exclusão mútua o que poderá gerar algum problema. Contudo como a leitura e escrita é só feita na altura da criação do canal de comunicação ou no fecho é pouco provável a ocorrencia de erros.

O resultado final do diagrama de classes simplificado foi o seguinte:

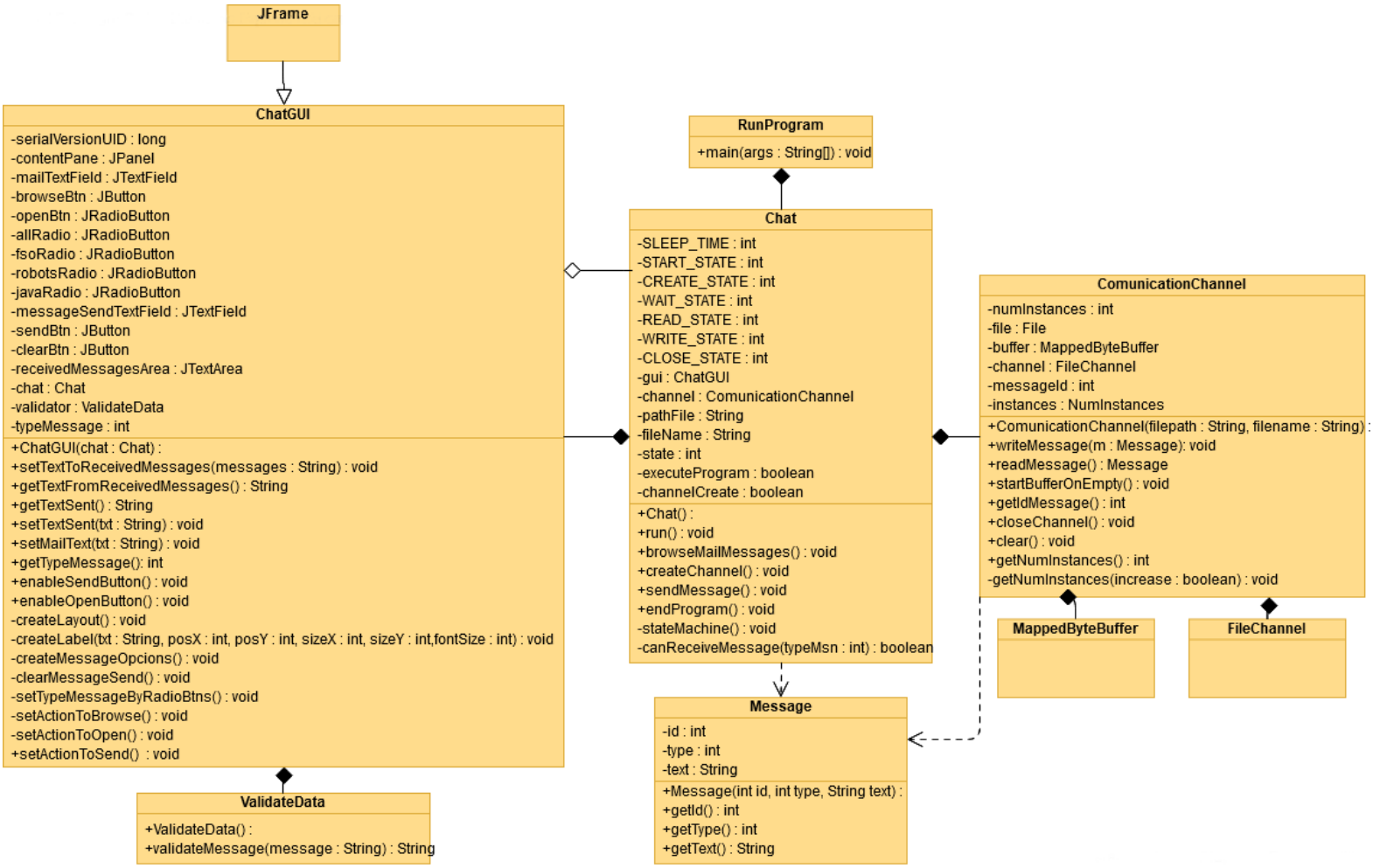

Figura 25 - Diagrama de classes da implementação completa

As swimlanes têm o objetivo de representar a comunicação entre objetos e a sua sincronização entre tarefas. As swimlanes obtidas para a implementação foram:

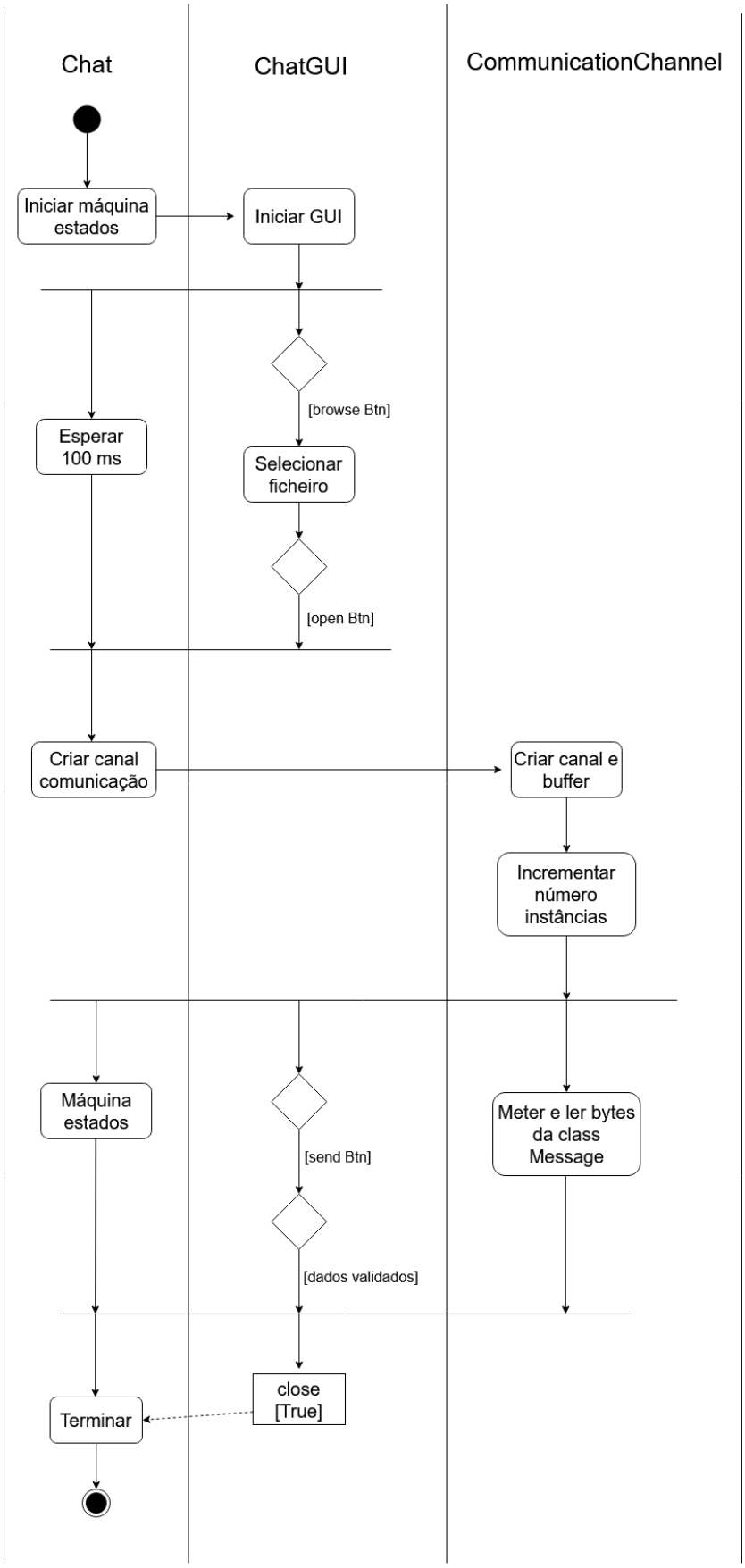

Figura 26 - Swimlanes obtidas

3. Conclusões

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

- Utilizar o *WindowBuilder* para construir interfaces gráficas de forma simples e automática.

- Qual é o funcionamento do sistema operativo e as diferentes arquiteturas possíveis do sistema operativo.

- Conceito sobre o que é um processo num sistema operativo e como um processo é visto pelos restantes processos.

- Implementação de um processo na linguagem programação Java.

- Utilizar um buffer para comunicar entre os diferentes processos, que no caso do Java é feito através de memória mapeada.

- Importância de estados de espera durante o decorrer de um processo.

- Qual a diferença entre um programa que está sempre a correr ou em *loop* e um *actionPerform.*

No futuro poderia ser implementados mecanismos que correm em paralelismo puro. Estes mecanismos são feitos à custa de *Threads* ou tarefas leves, onde cada tarefa está a correr num *core* do CPU havendo desta forma paralelismo puro dentro do mesmo processo.

4. Bibliografia

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

https://www.tutorialspoint.com/operating_system/os_processes.htm

https://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.html

https://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileLock.html

5. Anexo

```
public class Chat {

	private final int SLEEP_TIME = 100;//TODO depois ajustar
	private final int START_STATE = 0, CREATE_STATE= 1, WAIT_STATE = 2, READ_STATE = 
3, WRITE_STATE = 4, CLOSE_STATE = 5;

	private ChatGUI gui;
	private ComunicationChannel channel;
	private String pathFile;// caminho incluindo o proprio ficheiro selecionado pelo 
utilizador
	private String fileName;
	private int state = 0;// 0 = iniciar, 1 = espera, 2 = enviar, 3 = terminar
	private boolean executeProgram = true;// variavel para parar a execução do 
programa
	private boolean channelCreate = false;//variavel diz quando o canal comunicação 
foi criado

	public Chat() {

	}
	public void run(){
		do {
			stateMachine();

		} while (executeProgram);

	}
	private void stateMachine() {
		switch (state) {

		case START_STATE:
			gui = new ChatGUI(Chat.this);
			state = WAIT_STATE;
			break;

		case CREATE_STATE:
			try {
			channel = new ComunicationChannel(pathFile,fileName);
			channel.startBufferOnEmpty();
			gui.enableSendButton();
			JOptionPane.showMessageDialog(gui, "Canal comunicação criado");// TODO 
para debug
			channelCreate = true;
			state = READ_STATE;
		  } catch (Exception e) {
			e.printStackTrace();
		  }
			break;


		case WAIT_STATE:
			try {
				Thread.sleep(SLEEP_TIME);
				if(channelCreate && state == WAIT_STATE)
					state = READ_STATE;
```

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

                  break;


            case READ_STATE:
                  try {
                        Message message = channel.readMessage();
                        if (message != null) {
                              if (canReceiveMessage(message.getType())) {
                                    String previous =
gui.getTextFromReceivedMessages();
                                    gui.setTextToReceivedMessages(
                                                "Id: " + message.getId() + "
Mensagem: " + message.getText() + "\n" + previous);// TODO
                                    }
                        }
                  } catch (IOException e1) {
                        e1.printStackTrace();
                  }

                  state = WAIT_STATE;
                  break;

            case WRITE_STATE:
                  Message message = new Message(channel.getIdMessage(),
gui.getTypeMessage(),gui.getTextSent());
                  try {
                        channel.writeMessage(message);
                  } catch (IOException e) {
                        e.printStackTrace();
                  }
                  gui.setTextSent("");
                  state = READ_STATE;
                  break;

            case CLOSE_STATE:
                  System.out.println("Terminar");
                  if (channel != null)
                        channel.closeChannel();

                  if (channel != null) {
                        System.out.println(channel.getNumInstances());
                        if(channel.getNumInstances() == 0)
                              channel.clear();
                  }

                  executeProgram = false;
                  break;

            }

      }
      //validação para recepção das mensagens do mesmo tipo
      private boolean canReceiveMessage(int typeMsn) {
            return typeMsn == 0 || typeMsn == gui.getTypeMessage() ||
gui.getTypeMessage() == 0;
      }
```

```
	// procura numa diretoria se não for dentro do package communication n permite
	public void browseMailMessages() {
		FileDialog fd = new FileDialog(gui, "Open file", FileDialog.LOAD);
		fd.setFile("*.dat");
		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 dat
		else {
			System.out.println(directory + filename);
			pathFile = directory;
			this.fileName = filename;
			gui.setMailText(filename);
			gui.enableOpenButton();
		}


	}

    public void createChannel() {
      state = CREATE_STATE;
    }

	public void sendMessage() {
		state = WRITE_STATE;
	}

	public void endProgram() {
		state = CLOSE_STATE;
	}

}
```

```
public class ComunicationChannel {

	private int numInstances=0;//variavel que diz o número de instancias desta classe

	private final int MAX_SIZE = 520;// corresponde a ao tamanho de todos os ints mais
o tamanho maximo da string
	private File file;
	private MappedByteBuffer buffer;
	private FileChannel channel;
	//id que irá ser utilizado, sempre +1 superior ao currente, quando n tem mensagens
antigas é -1
	private int messageId = -1;
	private NumInstances instances;

	@SuppressWarnings("resource")
	public ComunicationChannel(String filePath, String fileName) {
		file = new File(filePath+fileName);
		instances = new NumInstances(filePath,fileName);
		try {
			channel = new RandomAccessFile(file, "rw").getChannel();
			buffer = channel.map(FileChannel.MapMode.READ_WRITE, 0,
MAX_SIZE*8);//para bytes
			buffer.clear();

		} catch (IOException e) {
			e.printStackTrace();
		}
		getNumInstances(true);
	}
	// mete bytes no buffer respetivos da mensagem
	public synchronized void writeMessage(Message m) throws IOException {
		FileLock f = channel.lock(0, channel.size(), false);
		clear();// limpa a mensagem para nao aparecerem caracteres de outras
mensagens
		buffer.position(0);
		buffer.putInt(m.getId());
		buffer.putInt(m.getType());
		for (int i = 0; i < m.getText().length(); i++) {
			char c = m.getText().charAt(i);
			buffer.putChar(c);
		}
		f.release();
	}


	//verifica se existe mensagem, caso exista retorna uma instancia de message
	public synchronized Message readMessage() throws IOException {
		FileLock f = channel.lock(0, channel.size(), false);
		buffer.position(0);
		int newId = buffer.getInt(0);
		int c = buffer.getChar(4 + 4);// verificar se o inteiro corresponde ao char
recebido não é null

		// mensagem no buffer tem o mesmo indice de escrita do proximo ou caso o
		// primeiro caracter por n poder ser null ser != null
		if (newId == messageId || (messageId == -1 && c != 0)) {
			int type = buffer.getInt(4);// '0
```

```
                String text = "";
                int index = 0;
                // loop para buscar os chars guardados no buffer
                while (true) {
                        char cx = buffer.getChar(4 + 4 + index * 2);
                        int ix = cx;
                        index++;
                        if (ix != 0)
                                text += cx;
                        else
                                break;
                }

                Message m = new Message(newId, type, text);

                if (messageId != -1)
                        messageId++;
                else
                        messageId += 2;

                f.release();
                return m;
        }
        f.release();

        return null;

    }
    //fazer set para o indice correto das mensagens, quando nao tem fica a -1
    public synchronized void startBufferOnEmpty() throws IOException {
        FileLock f = channel.lock(0, channel.size(), false);

        // obter o primeiro caracter da mensagem, caso seja != 0 é porque já existe
uma
        // mensagem previa
        int ix = buffer.getChar(4 + 4);
        if (ix != 0) {
                int id = buffer.getInt(0);
                System.out.println("id: " + id);
                messageId = id + 1;
        } else
                System.out.println("Sem mensagem");
        f.release();

    }
    //quando não tem mensagens previas fica a -1 então tem de se converter
    public int getIdMessage() {
        if(messageId == -1)
                return 0;
        else
                return this.messageId;
    }

    //fecha o FileChannel
    public void closeChannel() {
        try {
                channel.close();
                getNumInstances(false);
        } catch (IOException e) {
                e.printStackTrace();
```

```
			}

		}

		//limpa os bytes armazenados no ficheiro
		public void clear() {
			for (int i = 0; i < MAX_SIZE * 8; i++) {
				buffer.put(i, (byte) 0);
			}
		}

		//obtem o número de instancias criadas do canal comunicação para depois limpar o
ficheiro quando é 0
		public int getNumInstances() {
			return this.numInstances;
		}

		private void getNumInstances(boolean increase) {
			int previous = instances.readFile();

			if (previous != -1) {
				if (increase)
					previous++;
				else
					previous--;
				instances.writeFile(""+previous);
				numInstances = previous;
			} else {
				previous = 1;
				instances.writeFile(""+previous);
				numInstances = previous;
			}

		}

}
```

```
public class Message {

  private int id;//contador que é incrementado cada vez que é enviada uma mensagem a
começar em 0
  private int type;//valor que define para quem deve ser entrege a mensagem || 0 =
todos, 1 = FSO, 2 = robots, 3 = Java
  private String text;//mensagem enviada

  public Message(int id, int type, String text) {
      this.id = id;
      this.type = type;
      this.text = text;
  }

  public int getId() {
      return this.id;
  }

  public int getType() {
      return this.type;
  }

  public String getText() {
      return this.text;
  }

}
```

```
/escreve num ficheiro o numero de instancias que existem da classe, para resetar os
valores que existem no ficheiro
//quando so existe uma instancia do chat ele apaga o conteudo do ficheiro
public class NumInstances {
	private File file;
	private String directory;
	private String fileName;

	public NumInstances(String directory, String fileName) {
		this.directory = directory;
		this.fileName = getFileName(fileName)+".txt";

		File file = new File(directory+this.fileName);
		if (!file.exists()) {
			try {
				file.createNewFile();
				writeFile("0");//precisa ter o valor 0 senao da erro
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
		this.file = new File(directory+this.fileName);
	}

	//escreve no ficheiro o número de instâncias
	public boolean writeFile(String txt) {
		try {
			FileWriter myWriter = new FileWriter(directory+fileName);
			myWriter.write(txt);
			myWriter.close();
			return true;
		} catch (IOException e) {
			e.printStackTrace();
			return false;
		}

	}

	//lê do ficheiro o número de instâncias
	public int readFile() {
		try {
			Scanner myReader = new Scanner(this.file);
			int data = -1;
			data = Integer.parseInt(myReader.nextLine());
			myReader.close();
			return data;
		} catch (FileNotFoundException e) {
			e.printStackTrace();
			return -1;
		}
	}

	private String getFileName(String fileName) {
		if(fileName.contains(".")) {
			String y = fileName.replace(".", ",");
			String[] slipNames = y.split(",");
			return slipNames[0];
		}
		return fileName;
	}
}
```

```
public class ChatGUI extends JFrame {

	private static final long serialVersionUID = 1L;
	private JPanel contentPane;
	private JTextField mailTextField;
	private JButton browseBtn;
	private JRadioButton openBtn;
	private JRadioButton allRadio;
	private JRadioButton fsoRadio;
	private JRadioButton robotsRadio;
	private JRadioButton javaRadio;
	private JTextField messageSendTextField;
	private JButton sendBtn;
	private JButton clearBtn;
	private JTextArea receivedMessagesArea;

	private Chat chat;
	private ValidateData validator;
	private int typeMessage = 0;// 0 = todas, 1 = FSO, 2 = Robots, 3 =
Java

	public ChatGUI(Chat chat) {
		this.chat = chat;// para poder chamar as funções quando se clica nos butões
		validator = new ValidateData();

		//evento antes de terminar a janela
		addWindowListener(new WindowAdapter() {
			// TODO
			@Override
			public void windowClosing(WindowEvent e) {
				e.getWindow().dispose();//destroi a janela
				chat.endProgram();//Mete o state a 3 e termina a execução
			}
		});
		//cria as caracteristicas base da JFrame e do painel
		setBounds(100, 100, 675, 345);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		setTitle("My Chat");
		contentPane.setLayout(null);
		contentPane.setBackground(new Color(130, 130, 130));

		createLayout();

		setVisible(true);
	}

	//cria o layout do programa
	private void createLayout() {
		createLabel("Caixa do correio", 10, 11, 121, 23, 14);

		// textfield da caixa correio
		mailTextField = new JTextField();
		mailTextField.setBounds(201, 13, 236, 21);
		mailTextField.setEditable(false);
		contentPane.add(mailTextField);

		// butão de browse
		browseBtn = new JButton("Browse");
		browseBtn.setBounds(447, 11, 89, 23);
```

```
            contentPane.add(browseBtn);

            // butão abrir
            openBtn = new JRadioButton("Abrir");
            openBtn.setBounds(546, 12, 89, 23);
            openBtn.setEnabled(false);
            contentPane.add(openBtn);

            // cria os radio buttons do tipo mensagem
            createMessageOpcions();

            createLabel("Mensagem a Enviar", 201, 43, 146, 23, 14);

            // textfield do texto a enviar
            messageSendTextField = new JTextField();
            messageSendTextField.setBounds(201, 77, 335, 23);
            contentPane.add(messageSendTextField);

            // butão de enviar
            sendBtn = new JButton("Enviar");
            sendBtn.setBounds(546, 77, 89, 23);
            sendBtn.setEnabled(false);
            contentPane.add(sendBtn);

            // butão de limpar
            clearBtn = new JButton("Limpar");
            clearBtn.setBounds(546, 111, 89, 23);
            contentPane.add(clearBtn);

            createLabel("Mensagens Recebidas", 201, 111, 169, 23, 14);

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

      sbrText.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
            sbrText.setBounds(201, 145, 434, 150);
            receivedMessagesArea = new JTextArea();
            receivedMessagesArea.setEditable(false);
            sbrText.setViewportView(receivedMessagesArea);
            contentPane.add(sbrText);

            setTypeMessageByRadioBtns();// listeners dos radio btns
            clearMessageSend();// listener para limpar a mensagem a enviar
            setActionToBrowse();// listener do butão browse
            setActionToOpen();// listener do butão abrir
            setActionToSend();// listener para o butão enviar
      }
      // função para escrever texto no painel
      private void createLabel(String txt, int posX, int posY, int sizeX, int sizeY, int
fontSize) {
            JLabel label = new JLabel(txt);
            label.setFont(new Font("Arial", Font.BOLD, fontSize));
            label.setBounds(posX, posY, sizeX, sizeY);
            contentPane.add(label);
      }

      // função para criar os radio buttons
      private void createMessageOpcions() {
            // cria os butões
            allRadio = new JRadioButton("Todas", true);
            allRadio.setFont(new Font("Tahoma", Font.BOLD, 12));
```

```
		fsoRadio = new JRadioButton("FSO");
		fsoRadio.setFont(new Font("Tahoma", Font.BOLD, 12));
		robotsRadio = new JRadioButton("Robots");
		robotsRadio.setFont(new Font("Tahoma", Font.BOLD, 12));
		javaRadio = new JRadioButton("Java");
		javaRadio.setFont(new Font("Tahoma", Font.BOLD, 12));

		// cria um grupo de butões para só poder estar um selecionado de cada vez
		ButtonGroup bgroup = new ButtonGroup();
		bgroup.add(allRadio);
		bgroup.add(fsoRadio);
		bgroup.add(robotsRadio);
		bgroup.add(javaRadio);

		// cria um painel só para os butões
		JPanel radioPanel = new JPanel();
		radioPanel.setLayout(new GridLayout(4, 1));
		radioPanel.setBounds(10, 45, 169, 254);
		// adiciona uma borda ao painel
		radioPanel.setBorder(BorderFactory.createTitledBorder(
				"<HTML><BODY><FONT face=\"Arial\" size=\"4\"><B>Tipo de
mensagem</B></FONT></BODY></HTML>"));
		radioPanel.add(allRadio);
		radioPanel.add(fsoRadio);
		radioPanel.add(robotsRadio);
		radioPanel.add(javaRadio);

		contentPane.add(radioPanel);

	}

	// faz set de vazio para a mensagem a enviar
	private void clearMessageSend() {
		clearBtn.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				receivedMessagesArea.setText("");

			}
		});
	}

	// faz set do tipo de mensagem conforme se clica nos radio buttons
	private void setTypeMessageByRadioBtns() {
		allRadio.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				typeMessage = 0;

			}
		});

		fsoRadio.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				typeMessage = 1;

			}
		});

		robotsRadio.addActionListener(new ActionListener() {
```

```
                @Override
                public void actionPerformed(ActionEvent e) {
                        typeMessage = 2;

                }
        });

        javaRadio.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                        typeMessage = 3;

                }
        });

    }

    // chama o metodo de fazer browse da classe que instancia o objeto desta classe
    // para executar o seu propósito
    private void setActionToBrowse() {
        browseBtn.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                        chat.browseMailMessages();

                }
        });
    }

    // chama o metodo abrir mensagens da classe que instancia o objeto desta classe
    private void setActionToOpen() {
        openBtn.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                        chat.createChannel();
                        openBtn.setEnabled(false);
                }
        });
    }

    // chama o metodo enviar mensagens da classe que instancia o objeto desta classe
    private void setActionToSend() {
        sendBtn.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                        if (validator.validateMessage(messageSendTextField.getText())
== null)
                                chat.sendMessage();

                        else
                                JOptionPane.showMessageDialog(ChatGUI.this,

    validator.validateMessage(messageSendTextField.getText()), "Valores errados",
                                                JOptionPane.WARNING_MESSAGE);
                }
        });
    }

    public void setTextToReceivedMessages(String messages) {
        this.receivedMessagesArea.setText(messages);
    }
```

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

	public String getTextSent() {
		return this.messageSendTextField.getText();
	}

	public void setTextSent(String txt) {
		this.messageSendTextField.setText(txt);
	}

	public void setMailText(String txt) {
		this.mailTextField.setText(txt);
	}

	public int getTypeMessage() {
		return this.typeMessage;
	}

	public void enableSendButton() {
		this.sendBtn.setEnabled(true);
	}

	public void enableOpenButton() {
		this.openBtn.setEnabled(true);
	}

}
```

```
public class ValidateData {

	public String validateMessage(String message) {
		if(message.trim().equals("") || message == null)
			return "A mensagem enviada não pode estar vazia";

		else if(message.length() < 1)
			return "A mensagem tem de ter pelo menos 1 caracteres";

		else if(message.length() > 256)
			return "A mensagem só pode conter no máximo 256 caracteres";

		return null;

	}
}
```

```
public class RunProgram {

	public static void main(String[] args) {
		Chat chat = new Chat();
		chat.run();
	}

}
```