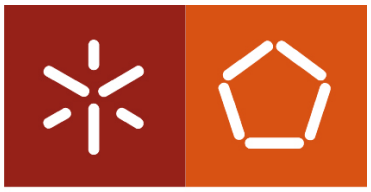


Agentes e Sistemas Multiagente

Fase 2

Mestrado Integrado em Engenharia Biomédica

Sistemas Inteligentes

(2º Semestre/Ano letivo 2018-2019)

Docentes: Cesar Analide e Filipe Gonçalves

Discentes:

A74407 Ana Duarte

A76542 Ângela Gonçalves

Braga, 5 de junho de 2019

RESUMO

A aplicabilidade de modelos informáticos baseados em agentes é um modo eficiente de se simulares situações reais complexas e de se otimizarem os resultados pretendidos. Com o presente trabalho pretende-se construir um sistema de gestão da compra e venda de produtos farmacêuticos que possibilite a visualização de métricas relativas ao sistema. Para isso, o trabalho terá uma introdução, uma descrição do problema, a descrição da implementação, a indicação dos resultados obtidos e conclusões e trabalhos futuros.

Pretende-se que o sistema de gestão a implementar utilize modelos de minimização de custos que, no caso do presente trabalho, estão associados ao cálculo da distância entre farmácias e o domicílio de um cidadão.

Com a elaboração da programação para implementar o sistema concluiu-se que a gestão de um conjunto de farmácias pode ser modelada e otimizada através de software de programação baseado em agentes. Constatou-se ainda, que este tipo de programação é facilitado pelo caráter intuitivo e pouco complexo das suas funções.

PALAVRAS-CHAVE

Agente; Sistema Multiagente; Farmácia; JADE; AUML; Performative; Behaviour.

ÍNDICE

1. INTRODUÇÃO

O presente projeto consiste na conceção e implementação de um Sistema Multiagente (SMA) que permita gerir a compra e a entrega de produtos ao domicílio em contexto farmacêutico. Este trabalho prático corresponde à 2ª fase do projeto, em que se pretende criar, com recurso à *framework* JADE, um SMA que simule e se aproxime da realidade de um sistema farmacêutico.

1.1 Diagramas UML

Na 1ª fase do projeto foram explicitados os conceitos que servem de base à implementação e feito um levantamento do estado da arte. Além disso, nesta fase foi ainda modelada a arquitetura de todo o sistema, através de diagramas UML. Estes diagramas serviram de ponto de partida para a 2ª fase do projeto. No entanto, após alguns ajustes, os diagramas sofreram pequenas alterações. Desta forma, os diagramas UML de Classes, de Sequência e de Estado considerados estão representados nas Figuras 1, 2 e 3, respetivamente.

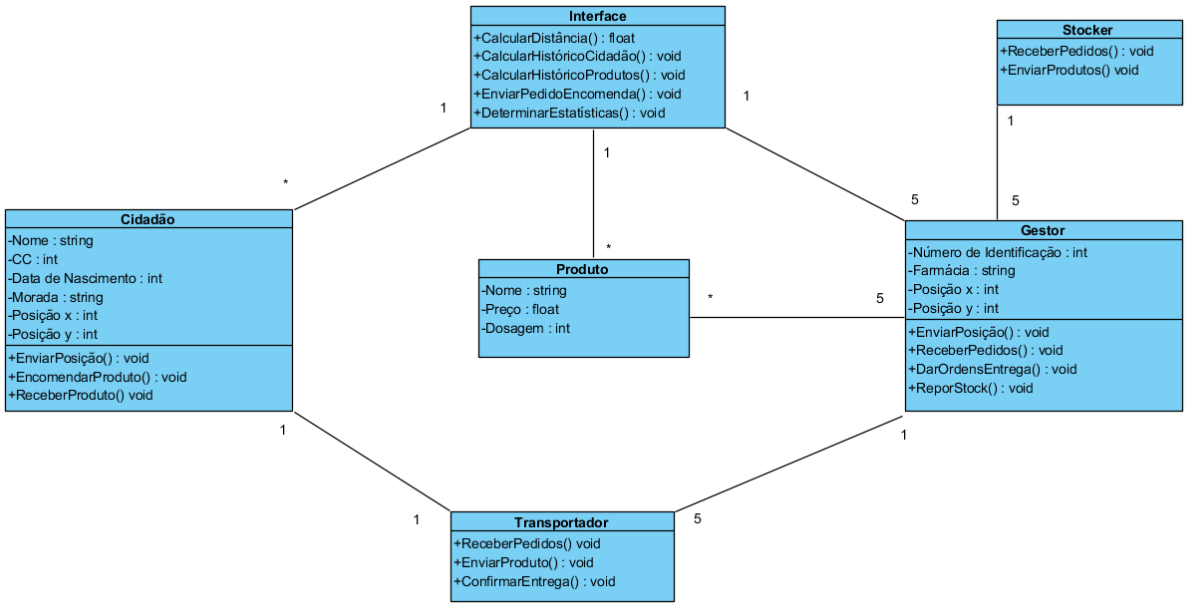


Figura 1 *- Diagrama de Classes usado como ponto de partida para a 2ª fase do projeto.*

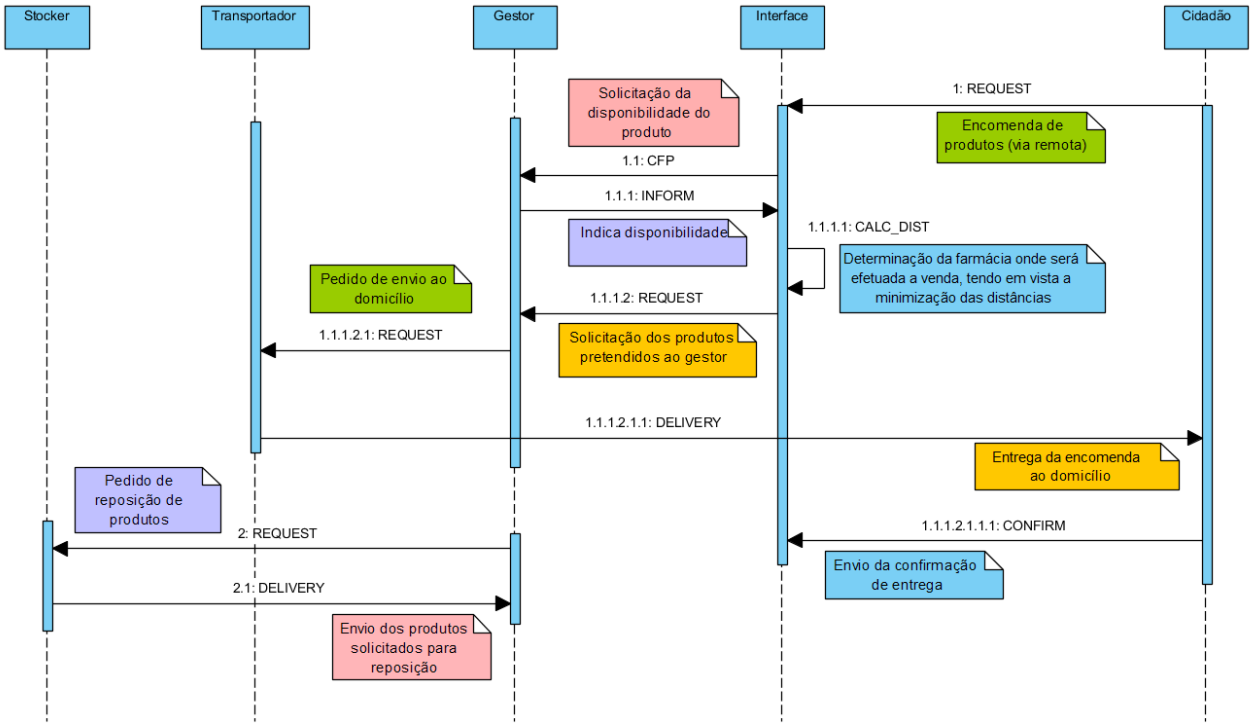


Figura 2 *- Diagrama de Sequência usado como ponto de partida para a 2ª fase do projeto.*

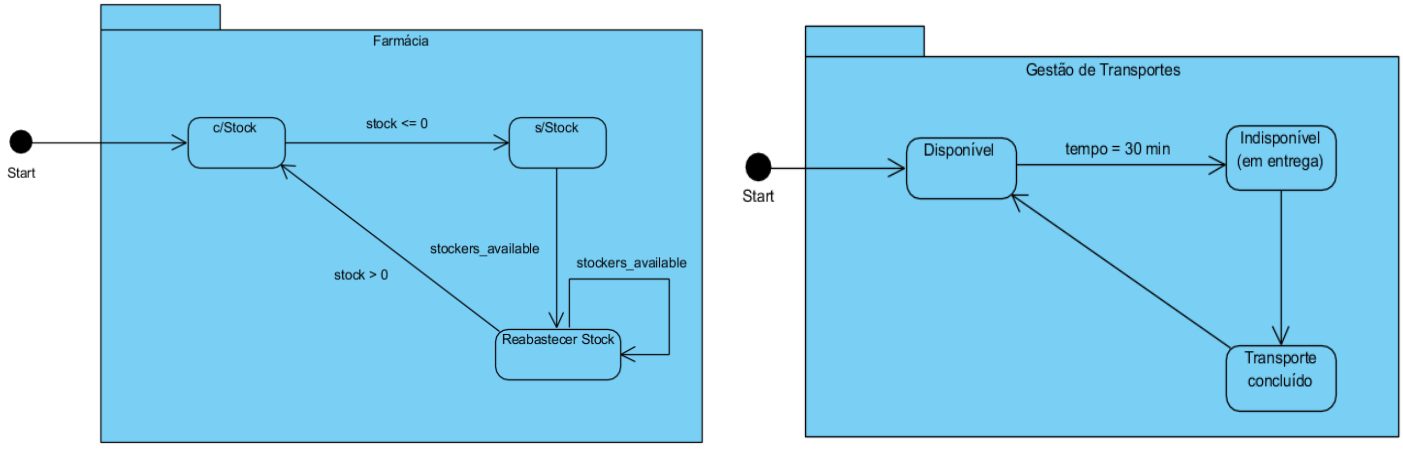


Figura 3 *- Diagrama de Estado usados como ponto de partida para a 2ª fase do projeto (à esquerda da farmácia e à direita da gestão de transportes).*

1.2 Framework JADE

Para a execução de modelações baseadas em agentes existem diversos *softwares* disponíveis que poderão ser utilizados. A *framework* mais utilizada é a *Java Agent Development Framework* (JADE), que é uma ferramenta que permite criar SMA interoperáveis e em

conformidade com as especificações FIPA, uma organização de padrões IEEE que promove a tecnologia baseada em agentes [1]. As especificações FIPA – *Foundation for Intelligent Physical Agents* – representam uma coleção de padrões que promovem a interoperabilidade entre agentes. A aplicação deste protocolo para a comunicação entre agentes designa-se por ACL – *Agent Communication Language* – e dispõe de um conjunto de parâmetros estruturados que permite a sua interação através de uma linguagem comum. Alguns dos parâmetros mais utilizados são [2]:

- **Performative:** parâmetro de caráter obrigatório que informa acerca do tipo de comunicação (e.g. REQUEST, INFORM, AGREE, REFUSE);
- **Sender:** informa que um agente está a enviar uma mensagem;
- **Receiver:** indica o agente que recebe a mensagem;
- **Content:** parâmetro que indica o conteúdo da mensagem;
- **Ontology:** identifica os termos e o vocabulário a utilizar para a comunicação.

Do ponto de vista funcional, o JADE fornece serviços para aplicações distribuídas *peer-to-peer*, em que cada agente é identificado por um nome global único (*Agent Identifier* – AID), pessoalizando, assim, cada agente. Esta plataforma permite que cada agente descubra de forma dinâmica outros agentes e que comunique e interaja com eles. Além disso, oferece ainda a possibilidade de registar/modificar serviços ou de procurar agentes que prestem um determinado serviço. [2][3]

1.3 JFreeChart

JFreeChart é uma biblioteca Java de gráficos gratuita para utilização na plataforma Java. Este software facilita a visualização informação ao exibi-la através de gráficos de qualidade. O JFreeChart contém uma API consistente e bem documentada, suporta uma variedade extensa de gráficos (por exemplo: barras, linhas, dispersão, séries temporais), tem um design flexível, é facilmente estendível e é utilizado tanto no lado das aplicações do cliente como no do servidor.

Para além disso, o *software* possibilita diversos tipos de saída, incluindo componentes Swing e JavaFX, arquivos de imagem como PNG e JPEG, e formatos de arquivos gráficos vetoriais como PDF, EPS e SVG. [4][5]

1.4 Estrutura do Relatório

Do ponto de vista estrutural, o presente relatório divide-se de acordo com:

- Capítulo 1 – Introdução;
- Capítulo 2 – Descrição do Trabalho;
- Capítulo 3 – Implementação;
- Capítulo 4 – Resultados;
- Capítulo 5 – Conclusões e Trabalho Futuro.

2. DESCRIÇÃO DO TRABALHO

Nesta secção é efetuada uma descrição do sistema a implementar e discriminam-se os requisitos tidos em conta para o seu funcionamento.

De uma forma sucinta e tal como indicado no Diagrama de Classes da Figura 1, o sistema pressupõe a existência dos elementos *Cidadão* (que encomendam produtos), *Interface* (que gere o sistema), *Gestor* (entidade responsável pela gestão de uma farmácia específica), *Stocker* (fornecedor), *Transportador* (envia os produtos da farmácia ao domicílio) e *Produto*.

Para o caso em estudo, considerou-se que os cidadãos encomendam os produtos que pretendem adquirir através de uma interação com a interface e que esta, por sua vez, é a responsável por decidir acerca de qual o gestor que fornecerá esses produtos. Neste caso, a interface vai encaminhar sempre o pedido do cliente para a farmácia que se situa mais próxima do seu domicílio, restringindo esta decisão, evidentemente, apenas a farmácias que disponham do produto em *stock*.

De uma forma simplificada, a interação entre os agentes faz-se de acordo com o esquema da Figura 4.

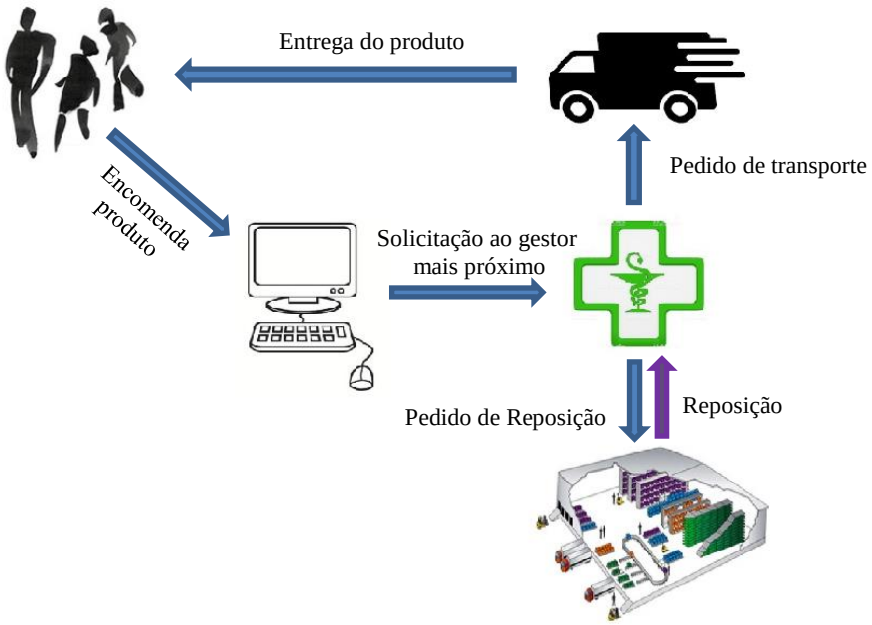


***Figura 4** - Esquema geral do Sistema de Gestão de Farmácias.*

Para a implementação do sistema proposto estabeleceram-se inicialmente um conjunto de pressupostos que definem as condições em que o sistema irá sem executado.

2.1 Pressupostos Iniciais

Para efeitos de modelação do sistema proposto foram considerados os seguintes pressupostos:

- Existência de 5 farmácias que cooperam umas com as outras e estão localizadas em pontos dispersos.
- Cada farmácia tem um transportador único e dedicado;
- A Interface é o intermediário entre o Cidadão e o Gestor;
- Apenas o Gestor de farmácia comunica com o Transportador e com o Stocker;
- Os custos estão diretamente relacionados com a distância percorrida pelo transportador numa entrega ao domicílio;
- O tempo de entrega dos produtos pelo transportador é de 10 segundos;
- O Stocker tem capacidade ilimitada de reposição de stocks em farmácia e demora 50 segundos a processar essa reposição.

3. IMPLEMENTAÇÃO

3.1 MainContainer.java

O *Main Container* é o ficheiro em que se geram os agentes, numa mesma plataforma de agentes, relativos à Interface, Cidadão, Gestor, Transportador e *Stocker*. Para a criação dos agentes, utiliza-se o comando `startAgentInPlatform`. No caso da Interface e do *Stocker* existe apenas um agente de cada um destes tipos, enquanto que, nos restantes casos, como existem múltiplos agentes, foi necessário recorrer-se a um ciclo para os criar.

3.2 Cidadão.java

O agente *Cidadão* é inicializado definindo-se, no *setup*, os atributos que o caracterizam e os comportamentos que irá realizar. Neste caso, considerou-se que cada cidadão exibe os comportamentos *EnviarPosicao(), EncomendarProduto1vez(), EncomendarProdutoigual(), EncomendarProdutonvez()* e *ReceberProduto().*

O comportamento *EnviarPosicao()* consiste em informar a interface das coordenadas geográficas da sua morada. Para tal, utiliza-se um *SimpleBehaviour*, enviando-se uma mensagem ACL do tipo INFORM, em que o *receiver* é a Interface.

Por sua vez, os comportamentos *EncomendarProdutoigual(), EncomendarProdutonvez()* e *EncomendarProduto1vez()* estão associados a uma identificação *random* (0, 1, 2, 3 ou 4), que atribui apenas um destes comportamentos a cada cidadão. Assim, permite-se que se criem cidadãos com perfis distintos, que tenham um comportamento de encomendar produtos apenas uma vez, ou encomendar sempre o mesmo produto de forma cíclica, ou encomendar produtos aleatórios várias vezes. O código construído para a seleção do perfil de Cidadão é o que se apresenta de seguida. Em qualquer um destes comportamentos, é enviado o pedido do produto para a Interface através de uma mensagem ACL do tipo REQUEST.

```
if (cliente == 0) {
    addBehaviour(new EncomendarProduto1vez());
}
if (cliente == 1 || cliente == 2) {
    this.addBehaviour(new EncomendarProdutoigual());
}
if (cliente > 2) {
    this.addBehaviour(new EncomendarProdutonvez());
}
```

Como exemplo de um destes tipos de comportamento, apresenta-se o que se associa a um cidadão que compra, de forma periódica, sempre o mesmo produto. Note-se que, neste caso, se trata de um *CyclicBehaviour* e que a seleção de qual o produto que o cidadão vai adquirir em todas as suas encomendas é efetuada de forma *random*.

```
int i = 0;
int id;
public void action() {
    if(i==0) {
        Random rand = new Random();
        id = rand.nextInt(100)+1;
    }
    i++;

    AID receiver2 = new AID();
    receiver2.setLocalName("Interface");
    ACLMessage msg2 = new ACLMessage(ACLMessage.REQUEST);
    msg2.addReceiver(receiver2);
    msg2.setContent(Integer.toString(id));
    myAgent.send(msg2);
```

Em relação ao *ReceberProduto()*, este comportamento consiste em aguardar a receção de uma mensagem do tipo INFORM (*CyclicBehaviour)*, que indica a entrega do produto por parte do transportador.

3.3 Gestor.java

De forma análoga à do *Cidadão*, no *Gestor*, começa-se por se definirem todos os atributos inerentes ao gestor da farmácia. No *setup*, é ainda necessário registar este agente nas "páginas amarelas" (*DFService*), configurando-se da seguinte forma:

```
// Registar agente
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setName(farmácia);
sd.setType("vendaProdutosFarmaceuticos");
dfd.addServices(sd);
try {
    DFService.register(this, dfd);
} catch (FIPAException fe) {
    fe.printStackTrace();
}
```

O *DFAgentDescription* corresponde à identificação única do agente e o tipo de serviço prestado é definido pelo *setType* como "vendaProdutosFarmaceuticos". Posteriormente, usa-se `DFService.register` para registar efetivamente o agente na *DFService*.

Adicionalmente, é preciso acrescentar-se, no final do código do agente, o comando `DFService.deregister` para que, caso o agente seja removido, se eliminem os seus dados das páginas amarelas.

Para o Gestor foi ainda criado um *HashMap* designado por *qtd_por_produto*, de forma a associar-se a cada um dos produtos existentes, a sua quantidade disponível nessa farmácia. A inicialização deste *HashMap* foi efetuada de forma aleatória, atribuindo-se um stock inicial a cada um dos produtos variável entre 0 e 9:

```
for(int i=1;i<101;i++) {
    int qtd = rand.nextInt(10);
    qtd_por_produto.put(i,qtd);
```

Quanto aos seus comportamentos, este agente terá três comportamentos distintos: *EnviarPosicao()*, *ReceberPedido()* e *ReporStock()*.

O primeiro comportamento é efetuado tal como o do Cidadão e consiste no envio das suas coordenadas geográficas para o agente Interface.

Por outro lado, o *ReceberPedidos()* está continuamente à espera (*CyclicBehaviour*) de mensagens do tipo CFP ou REQUEST ou de outro tipo. Caso as mensagens sejam do tipo CFP, são provenientes da Interface, que lhe solicitou uma "*Call for Proposal*", ou seja, solicitou que o gestor lhe indicasse se a farmácia tem em *stock* um determinado produto. O gestor, neste caso, cria uma resposta (*createReply*) do tipo INFORM, indicando se existe ou não disponibilidade do produto solicitado. De notar que ao conteúdo dessa mensagem é adicionada a expressão "disponivel" ou "indisponivel", o que permitirá à interface, numa fase posterior, saber que se trata de uma mensagem de indicação de disponibilidade de produtos.

```
//@Override
public void action() {
    ACLMessage msg = receive();
    if (msg != null && msg.getPerformative() == ACLMessage.CFP) {
        parts = msg.getContent().split(",");
        String conteudo = parts[0];
        String cidadao = parts[1];
        String pedido = parts[2];
        int stock = qtd_por_produto.get(Integer.parseInt(conteudo));
        ACLMessage resp = msg.createReply();
        resp.setPerformative(ACLMessage.INFORM);
        if(stock>0) {
            resp.setContent("disponivel" + "," + cidadao + "," + conteudo + "," + pedido);
            }
        else {
            resp.setContent("indisponivel" + "," + cidadao + "," + conteudo);
            }
        myAgent.send(resp);
        }
```

Caso a mensagem seja do tipo REQUEST significa que a Interface escolheu esse gestor para fornecer o produto ao cliente. Neste caso, o gestor terá de dar ordens ao transportador associado à sua farmácia para entregar o produto ao domicílio do cidadão. Para isso, terá de enviar uma mensagem também do tipo REQUEST ao transportador, indicando no seu conteúdo as coordenadas geográficas do domicílio do cliente. Após isso, o gestor deverá atualizar o stock do produto enviado, diminuindo a sua quantidade em 1 unidade.

```
AID receiver3 = new AID();
receiver3.setLocalName(numero);
ACLMessage msg2 = new ACLMessage(ACLMessage.REQUEST);
msg2.setContent(cidadao+","+xOrigem+","+yOrigem+","+produto);
msg2.addReceiver(receiver3);
myAgent.send(msg2);
int stock = qtd_por_produto.get(Integer.parseInt(produto))-1;
qtd por produto.replace(Integer.parseInt(produto),stock);
```

Importa ainda referir que se definiu que o Gestor1 está associado ao Transportador1, o Gestor 2 ao Transportador 2 e assim sucessivamente. Para estabelecer essa relação, usou-se o excerto de código:

```
String numero = myAgent.getLocalName().replaceAll("[a-z,A-Z]","");
numero = "Transportador"+numero;
```

No caso de o Gestor receber uma mensagem que não é nem do tipo CFP nem REQUEST, significa que a mensagem está a ser enviada pelo *Stocker* e, por isso, trata-se da receção dos produtos de reposição solicitados. Neste caso, o Gestor tem de atualizar o HashMap *qtd_por_produto*, incrementando ao stock que já existia a quantidade recebida.

```
int stock = qtd_por_produto.get(Integer.parseInt(produto));
qtd_por_produto.replace(Integer.parseInt(produto),stock+Integer.parseInt(quantidade));
```

Por último, o comportamento *ReporStock()*, consiste numa verificação periódica através de um *TickerBehaviour*, em que o Gestor averigua, para cada produto, o seu stock disponível. Caso o seu stock seja inferior a 3 unidades, envia uma mensagem do tipo REQUEST ao Stocker de forma a solicitar a reposição desses produtos.

3.4 Interface.java

A Interface é responsável pela criação dos produtos existentes para venda nas farmácias e serve de intermediário entre os cidadãos que necessitam de produtos e as farmácias que os disponibilizam. Este agente dispõe de *HashMaps* com coordenadas do cidadão, do gestor e histórico de compras do cidadão e histórico do número de vendas de cada produto.

Para a inicialização dos *HashMaps* relativos ao histórico do cidadão e do produto, é preciso efetuar um povoamento inicial, atribuindo a cada chave, o valor inicial 0.

```
protected void setup() {
    super.setup();
    for(int i=1;i<101;i++) {
        historicoProduto.put(i,0);
        }
    for(int j=1;j<11;j++) {
        historicoCidadao.put("Cidadão"+j,0);
```

O agente Interface possui dois comportamentos pré-definidos: o *CalcularDistancias()* e o *CalcularMetricas()*.

O comportamento *CalcularDistancias()* é o *CyclicBehaviour* que permite ao agente aguardar a chegada de mensagens dos cidadãos e dos gestores das farmácias. Para um correto funcionamento, importa notar que os *CyclicBehaviour* necessitam da inclusão de um *Block()* para evitar consumir recursos do computador e apenas ficar ativo quando chega uma mensagem. Este comportamento *CalcularDistancias()* está subdividido em três atuações distintas: uma que recebe as coordenadas dos cidadãos e dos gestores; outra, onde envia uma *Call for Proposal* (CFP) aos gestores e numa última atuação que consiste na solicitação de fornecimento de produto à farmácia mais próxima que tem o produto pretendido.

Para receber as coordenadas são colocadas as condições de a mensagem ser do tipo INFORM e de ter no seu conteúdo a expressão “Coordenadas:”. Para se conseguir identificar se as coordenadas foram enviadas pelo Cidadão ou pelo Gestor, recorreu-se ao extrato de código apresentado:

```
agentName = msg3.getSender().getLocalName();
String type = agentName.replaceAll("[0-9]","");

if (type.equals("Cidadão")) {
    String posicoes = msg3.getContent();
    coordenadasCidadao.put(agentName, posicoes);
    coordenadas_cidadao = posicoes.split(",");
    //System.out.println("Coordenadas CIDADAO: "+" "+coordenadasCidadao);
    }
else if(type.equals("Gestor")) {
    String posicoes2 = msg3.getContent();
    coordenadasGestor.put(agentName, posicoes2);
    //System.out.println("Coordenadas GESTOR: "+" "+coordenadasGestor);
    coordenadas_gestor = posicoes2.split(",");
```

Para processar os pedidos recebidos e escolher a farmácia é necessário, inicialmente, receber as mensagens do tipo REQUEST que contenham um produto que existe na base de dados dos produtos. De seguida, pesquisam-se na *DFService* os gestores que prestam serviços do tipo “vendaProdutosFarmaceuticos” e que se colocam numa lista “results”. Percorrendo-se essa lista, a Interface envia a cada um desses gestores uma mensagem do tipo CFP a solicitar que indiquem se têm ou não o produto pretendido em *stock*.

```
DFAgentDescription dfd = new DFAgentDescription();
ServiceDescription sd =new ServiceDescription();
sd.setType("vendaProdutosFarmaceuticos");
dfd.addServices(sd);
try {
    DFAgentDescription[] results = DFService.search(this.myAgent, dfd);
        if(results.length>0) {
            for (int i = 0; i < results.length; ++i) {
                DFAgentDescription dfd1 = results[i];
                AID receiver = dfd1.getName();
                ACLMessage msg2 = new ACLMessage(ACLMessage.CFP);
                msg2.setContent(conteudo+","+cidadao+","+numero_pedido);
                msg2.addReceiver(receiver);
                //System.out.println("Produto solicitado " +  " " + msg2.getContent());
                myAgent.send(msg2);
```

A última funcionalidade do comportamento *CalcularDistancia()* recebe mensagens do tipo INFORM, cujo conteúdo contenha a expressão "disponivel" (note-se que a palavra "indisponivel" é aceite por este "filtro"). Após se receber a totalidade das respostas dos gestores, são utilizadas as distâncias entre o cidadão e cada um dos gestores "disponíveis" para o cálculo da distância entre eles. Para cada nova distância calculada, compara-se esse resultado com o da distância mínima e, em caso de ser inferior, essa passa a ser a nova distância mínima. Assim, garante-se que o pedido possa ser enviado para a farmácia mais próxima (que tem disponibilidade do produto em stock).

```
if (parts[0].equals("disponivel")){
    String coord_gestor = coordenadasGestor.get(gestor).replaceFirst("Coordenadas:","");
    parts2=coord_gestor.split(",");
    xOrigem = Integer.parseInt(parts2[0]);
    yOrigem = Integer.parseInt(parts2[1]);
    int distance = (int) Math
            .sqrt(((Math.pow((xDestino - xOrigem), 2)) + (Math.pow((yDestino - yOrigem), 2))));
    if (distance < minDistancia) {
        minDistancia = distance;
        closestGestor = msg3.getSender();
        }
    }
```

Ainda neste passo e após se garantir o processamento de todos os gestores, atualizam-se os históricos relativos às compras de produtos e às compras dos cidadãos.

Relativamente ao comportamento *CalcularMetricas()*, que é um *TickerBehaviour* que é ativadado a cada 10 segundos, consideraram-se duas métricas distintas: o cálculo do histórico de vendas dos produtos e respetivo produto mais vendido e o cálculo do histórico de pedidos de cada um dos cidadãos e o respetivo cidadão que mais produtos encomendou.

O código para a métrica das vendas por produto é:

```
int max = 0;
int max2 = 0;
int prod = 0;
String prod2 = "";
for(int i=1;i<101;i++) {
    try {
        Thread.sleep(10);
    } catch (InterruptedException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
    int stock = historicoProduto.get(i);
    System.out.println("| Historico por produto - Produto" + i + " - " + stock + " |");
    if(stock>max) {
        max=stock;
        prod=i;
        }
    }
```

Para as encomendas pelos cidadãos construiu-se o seguinte código:

```
for(int j=1;j<11;j++) {
    try {
        Thread.sleep(10);
    } catch (InterruptedException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
    int stock2 = historicoCidadao.get("Cidadão"+j);
    System.out.println("| Historico por Cidadão - Cidadão" + j + " - " + stock2 + " |");
    if(stock2>max2) {
        max2=stock2;
        prod2="Cidadão"+j;
        }
    }
```

É ainda no *CalcularMetricas()* que se enviam os dados dos produtos necessários ao *gestor_dados.java*, que permite uma visualização gráfica das métricas relativas aos produtos mais vendidos.

3.5 Transportador.java

O agente Transportador recebe instruções do agente Gestor para proceder à entrega de um produto a um determinado Cidadão. Estas ações são executadas através de um *CyclicBehaviour*, que aguarda uma mensagem do tipo REQUEST proveniente do Gestor e, após a receção dessa mensagem, indica ao cidadão que o seu produto está a ser entrega. O tempo de transporte foi definido como 10 segundos.

```
AID receiver = new AID();
receiver.setLocalName(cidadao);
ACLMessage msg2 = new ACLMessage(ACLMessage.INFORM);
msg2.setContent(cidadao+","+produto);
msg2.addReceiver(receiver);
System.out.println("O produto " +  produto + " está a ser entregue ao " + cidadao);
try {
    Thread.sleep(10000);
} catch (InterruptedException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
myAgent.send(msg2);
```

3.6 Stocker.java

O Stocker tem um comportamento do tipo *Cyclic Behaviour*, em que aguarda a receção de mensagens do tipo REQUEST, que são compostas pelo produto pretendido e a quantidade necessária para a reposição do stock. Após isso, o Stocker informa acerca da entrega do produto

ao Gestor que o solicitou. Apresenta-se de seguida o código utilizado para a execução desta funcionalidade:

```
if (msg != null && msg.getPerformative() == ACLMessage.REQUEST) {
    parts = msg.getContent().split(",");
    String gestor = msg.getSender().getLocalName();
    String produto = parts[0];
    String quantidade = parts[1];

    AID receiver = new AID();
    receiver.setLocalName(gestor);
    ACLMessage msg2 = new ACLMessage(ACLMessage.INFORM);
    msg2.addReceiver(receiver);
    msg2.setContent(gestor + "," + produto + "," + quantidade);
    send(msg2);
```

3.7 Produto.java

A classe Produto não corresponde a um agente, mas sim à estrutura que os produtos apresentam. Assim, considerou-se que cada produto é definido pelo seu id, pelo seu nome, pelo preço e pela dosagem. Também é nesta classe que se definiram os métodos para se obter, de uma forma fácil, os parâmetros de cada produto, utilizando-se, para tal, os métodos *getid()*, *getnome()*, *getpreco()* e *getdosagem()*. Foi utilizado o seguinte código para a construção da classe Produto:

```
public class Produto {
    private int id;
    private String nome;
    private float preco;
    private int dosagem;

    Produto(int id, String nome, float preco, int dosagem){
        this.id = id;
        this.nome = nome;
        this.preco= preco;
        this.dosagem = dosagem;
    }
    public int getid() {
        return id;
    }
    public String getnome() {
        return nome;
    }
    public float getpreco() {
        return preco;
    }
    public int getdosagem() {
        return dosagem;
```

3.8 Gráficos - JFreeChart

De entre os gráficos do JFreeChart mais utilizados encontram-se o BarChart, LineChart, XYChart, 3D Chart/BarChart, BubbleChart, TimeSeries Chart e o PieChart, tendo sido este último o escolhido para a apresentação do produto mais vendido. Note-se que, num PieChart, o comprimento do arco de cada setor é proporcional à quantidade que ele representa.

Para a utilização do JFreeChart foi preciso previamente instalarem-se as bibliotecas externas no JADE necessárias. Para a construção do gráfico foi necessário, inicialmente, verificar o número de ocorrências de cada um dos produtos, utilizando-se, para isso, o seguinte código:

```
public void action() {
    ACLMessage msg = receive();
    if (msg != null && msg.getPerformative() == ACLMessage.INFORM) {
        parts = msg.getContent();
        int value = Integer.parseInt(parts);
        occurences[value]++;
```

De seguida, e através de um ciclo *for*, percorreram-se os primeiros 10 produtos, dos 100 que estavam registados na base de dados, para se encontrar os produtos mais vendidos de entre esses 10. Optou-se por esta simplificação para se proporcionar uma melhor visualização dos produtos e das respetivas quantidades vendidas. O código utilizado é apresentado de seguida:

```
for (int i=1;i<10;i++) {
    pie.setValue("Produto"+i, occurences[i]);
    System.out.println("Produto" + i + " vezes: " + occurences[i]);
    JFreeChart chart= ChartFactory.createPieChart(
            "Produtos Mais Vendidos", pie, true, true, false);
    try {
        ChartUtilities.saveChartAsJPEG(new File("C:/Users/Asus/Desktop/2ºsemestre/SI/grafico.jpeg"),
        chart,  500, 500
        );

    }catch(Exception e) {
        System.err.println("error"+e);
        }
```

Realça-se, do código, a indicação do formato de saída – JPEG – e do *path* para a sua gravação. Este gráfico tem a particularidade de ser atualizado em tempo real, ou dito de outro modo, à medida que o código está a ser executado, o ficheiro gerado em JPEG está a ser atualizado.

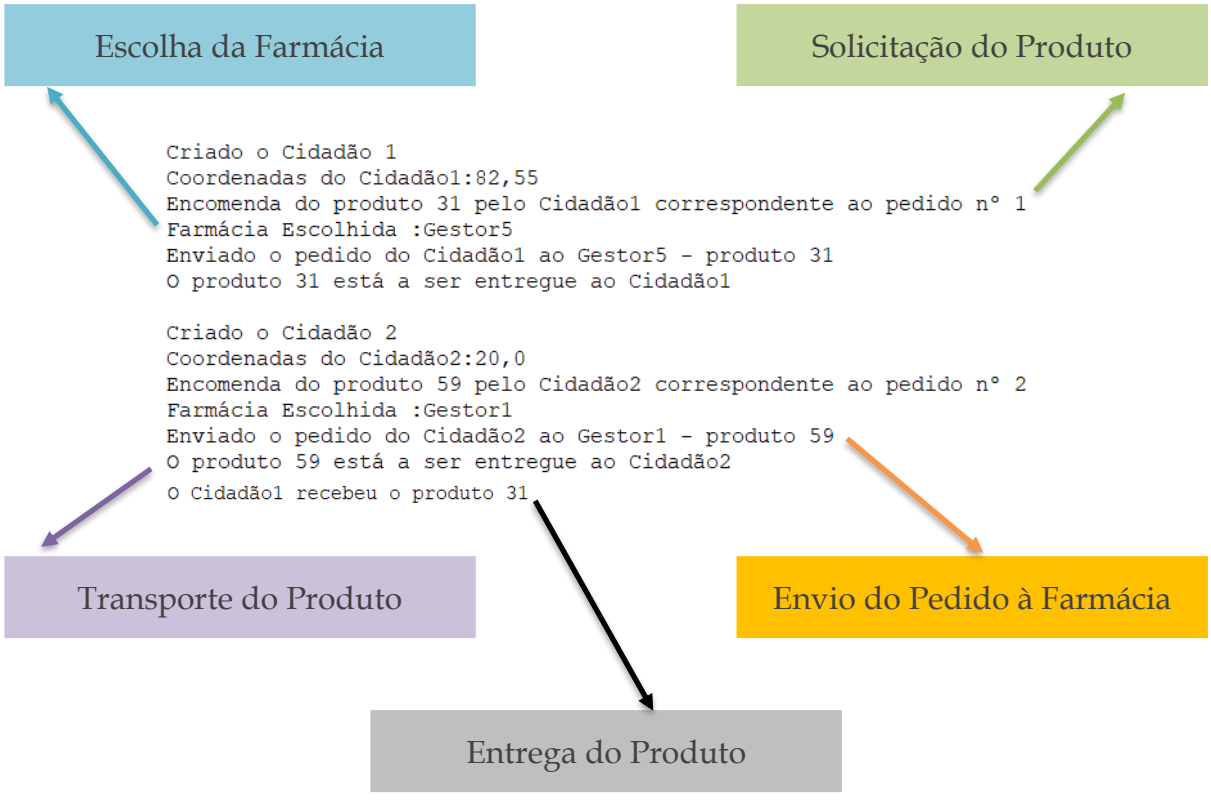
4. RESULTADOS

De forma a simular-se a viabilidade do sistema, executou-se o código desenvolvimento e obtiveram-se os resultados que se mostram nesta secção.

```
Foi inicializada a Interface
Foi inicializado o Stocker
Criado o Gestor 1
Criado o Gestor 2
Criado o Gestor 3
Criado o Gestor 4
Criado o Gestor 5
Coordenadas do Gestor3:64,45
Coordenadas do Gestor4:88,74
Coordenadas do Gestor5:77,71
Coordenadas do Gestor2:87,80
Coordenadas do Gestor1:48,10
Criado o Transportador 1
Criado o Transportador 2
Criado o Transportador 3
Criado o Transportador 4
Criado o Transportador 5
```

Inicialização dos Agentes

Envio das coordenadas



```
O Gestor3 está a encomendar ao Stocker o Produto3
```

Reposição de Produtos

```
4 unidades do produto 3 são entregues ao Gestor3 pelo Stocker
```

Solicitação ao Stocker

```
| Historico por produto - Produto2 - 1 |
| Historico por produto - Produto3 - 0 |
| Historico por produto - Produto4 - 1 |
| Historico por produto - Produto5 - 4 |
| Historico por produto - Produto6 - 6 |
| Historico por produto - Produto7 - 1 |
```

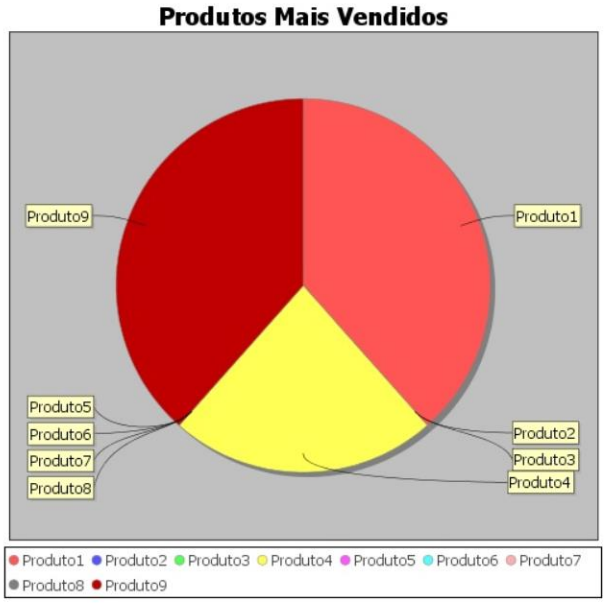
Vendas de cada produto

```
Produto mais vendido 6 com um total de 6
3 Produtos Mais Vendidos: [2, 9, 5]
```

Produtos mais vendidos

```
| Historico por Cidadão - Cidadão7 - 1 |
| Historico por Cidadão - Cidadão8 - 1 |
| Historico por Cidadão - Cidadão9 - 1 |
| Historico por Cidadão - Cidadão10 - 1 |
```

Compras produtos por cidadão



Da análise do gráfico observa-se que os produtos 1, 4 e 9 são os mais vendidos, não se registando a venda de qualquer um dos outros produtos.

Este tipo de representação visual proporciona um mais rápido entendimento da informação e pode ser utilizada como ferramenta de análise estatística.

5. CONCLUSÕES E TRABALHO FUTURO

No desenvolvimento do trabalho foram utilizados os métodos pré-definidos que a Framework JADE dispõe e constatou-se que este tipo de programação é simples e eficiente e que com algumas instruções se podem simular situações complexas representativas da realidade. Constatou-se também que os agentes apresentam características que permitem interpretar de uma forma rápida e fácil os seus resultados. Outra das conclusões que se retira da elaboração do trabalho é a sua adequação para situações de otimização dos lucros/custos.

Apesar de o trabalho se basear num problema que simula uma situação real apresenta ainda muitas simplificações e que, por isso, não permite uma visão mais alargada das potencialidades destes tipos de linguagem. Assim, a partir desta fase de desenvolvimento do trabalho poderão, em trabalhos futuros, serem implementadas funcionalidades mais complexas.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] J. Cucurull, R. Martí, G. Navarro-Arribas, S. Robles, B. Overeinder, and J. Borrell, "Agent mobility architecture based on IEEE-FIPA standards," Comput. Commun., vol. 32, no. 4, pp. 712–729, Mar. 2009.

[2] F. L. Bellifemine, G. Caire, and D. Greenwood, Developing multi-agent systems with JADE. John Wiley, 2004.

[3] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "JADE: A software framework for developing multi-agent applications. Lessons learned," Inf. Softw. Technol., vol. 50, no. 1–2, pp. 10–21, Jan. 2008.

[4] D. Gilbert, "The JFreeChart Class Library Version 0.9.18 Developer Guide," Object Refinery Limited, 2004.

[5] "http://www.jfree.org/jfreechart/," Consultado em 5 de junho de 2019.