Universidade Do Minho

Mestrado Integrado Em Engenharia Biomédica

Aplicações Distribuídas

TRABALHO PRÁTICO 2

Braga, Fevereiro 2013

Universidade Do Minho

Mestrado Integrado Em Engenharia Biomédica

Aplicações Distribuídas

TRABALHO PRÁTICO 2

Docente: António Sousa

Discentes: Hugo Gomes (58536)
Marta Moreira (54459)
Telma Veloso (58521)

Resumo

Pretendia-se desenvolver uma aplicação que implementasse uma farmácia com diversas filiais e que permitisse, simultaneamente, efectuar compras de forma presencial e *online*, através da definição de um programa cliente/servidor utilizando lógica de negócio (Java EE).

Neste sentido, desenvolveu-se código para uma aplicação, organizada em *sessions* e *entities*, que recorrem a uma biblioteca externa ao projecto onde estão as interfaces remotas. Foram criadas ainda as aplicações cliente conforme pedido no enunciado do problema.

Findo o desenvolvimento da aplicação, conclui-se que alguns dos métodos implementados apresentam, porém, uma ineficiência considerável, que poderia ser utltrapassada através da utilização de abordagens diferentes daquelas efectivamente empregues.

O presente relatório visa descrever a abordagem seguida até à solução definitiva sob a forma de uma sequência de decisões acerca da arquitectura global, entidades, sessões, *interfaces* e clientes.

Conteúdo

1 Abordagem ao Problema

Atendendo ao requisito de implementar lógica de negócio, foi utilizada a tecnologia de Java EE - Enterprise Edition. O primeiro passo consistiu na criação de uma aplicação empresarial, onde se escolheu o `GlassFish 3.1.2` como servidor e onde foi também criado o módulo `EJB` - Enterprise Java Bean. É neste módulo que estão armazenadas as entidades e as sessões necessárias ao funcionamento da aplicação. Este módulo está organizado em dois *packages* - *entities* e *sessions* - de modo a simplificar a organização e separar os diferentes elementos. Em primeiro lugar, devem ser criadas as entidades, que definem os objectos com os quais a aplicação lida e são vistas como tabelas preenchidas de informação. De seguida, são criadas as sessões que contêm a implementação dos métodos correspondentes aos objectivos da aplicação. Ao criar cada sessão foi ainda criada a sua respectiva interface remota que se localiza numa biblioteca externa ao projecto denominada `PharmaRemoteInterface`. Decidiu-se criar três sessões diferentes - `Loja`, `Estatisticas`, `GestaoFarma` - de modo a separar os métodos consoante o acesso que depois os cliente iriam ter. Para implementar os métodos nas sessões, recorreu-se ao comando *Add Business Method* que completou as interfaces com a invocação do respectivo método. Por fim, criaram-se as aplicações-cliente. Conforme referido, diferentes clientes foram criados de acordo com as sessões que iriam aceder. Deste modo, existem 4 clientes - `LeCli`, `GestaoCli`, `LojaCli` e `EstatisticasCli` - cada um apenas com acesso a uma interface. O cliente `LeCli` serve para ler os ficheiros `.csv` fornecidos aquando do primeiro trabalho e popular as tabelas/entidades com a informação existente nesses ficheiros. Por sua vez, o cliente `GestaoCli` pode adicionar e remover elementos existentes nas tabelas/entidades. Ambos os clientes referidos têm acesso à *interface* `GestaoFarma`, no entanto executam diferentes métodos pois foram criados com diferentes propósitos. Decidiu-se separar a leitura dos ficheiros `.csv` porque esta só deve acontecer uma vez. O cliente `LojaCli` corresponde ao cliente que pode ir à farmácia efectuar uma comprar presencial por intermédio de um vendedor ou que

pode efectuar um compra via *online*, onde se considera o vendedor representado pela *string* `'online'`. Por fim, o cliente `EstatisticasCli` é aquele que pode obter os valores de vendas diárias, semanais, mensais e anuais assim como a lista de medicamentos, utilizadores, funcionários e fornecedores que constituem os *tops*. Este cliente tem acesso à sessão `Estatisticas` pela *interface* `EstatisticasRemote`. De modo a ligar um cliente à aplicação pela respectiva *interface*, é necessário recorrer ao comando *Call Enterprise Bean*.

2 Entity Beans

De forma a assegurar a persistência da lógica de negócio, construíram-se cinco Entity Beans, respeitantes a cada um dos objectos criados no trabalho prático anterior. De um ponto de vista mais pragmático, os Entity Beans podem ser encarados, neste contexto, como a representação das tabelas que hipoteticamente existiriam caso se recorresse a um SGBD (Sistema de Gestão de Bases de Dados), contando que se definem igualmente uma chave primária e relações entre entidades, por exemplo. Assim, as entidades criadas foram:

1. `Medicamento`;
2. `Utilizador`;
3. `Fornecedor`;
4. `Funcionario`;
5. `Factura`.

No sentido de mimetizar este comportamento, recorreu-se a anotações para configurar cada uma das entidades, identificadas pela presença de um símbolo @. A título de exemplo, inclui-se parte do código de uma das entidades desenvolvidas, com o intuito de explicitar a pertinência das anotações:

```
@Entity
public class Funcionario implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    private String idfunc;
    private Integer nrVendido;
    private String nome;
    @OneToMany(targetEntity=entities.Factura.class, fetch= FetchType.LAZY, orphanRemoval=true)
    private Collection<Factura> facturas;

    public Funcionario() {
        super();
        facturas = new ArrayList<Factura>();
    }
```

A anotação `@Entity` serve o propósito de identificar a classe como um Entity Bean, sendo que a propriedade da entidade correspondente à chave primária é identificada com um @Id. Embora, neste caso específico, a identificação do funcionário seja explicitamente declarada na informação contida no ficheiro `.csv`, nos casos em que isso não acontece — como, por exemplo, na entidade `Factura` —, adicionou-se a anotação `@GeneratedValue`, que permite especificar um gerador capaz de gerar chaves de cada vez que um novo objecto é criado, conferindo-lhe, assim, uma identificação unívoca. Algumas das propriedades revelaram ser necessária a definição de relações entre determinados campos de entidades diferentes, como seja o exemplo acima desenvolvido, em que, para um mesmo funcionário, existem múltiplas facturas. Assim, criou-se uma colecção destes objectos, associada à anotação de relação `@OneToMany`, que, por sua vez, possui uma entidade alvo — `Factura`, neste contexto — e a um *fethcing* do tipo `LAZY`, de forma a garantir que,

aquando da selecção de um funcionário, não é disparado o carregamento da totalidade das facturas que lhe estão associadas através da hierarquia de relacionamento. Naturalmente, na entidade respeitante à factura, definiu-se uma relação `@ManyToOne` para com a entidade `Funcionário`. Adicionalmente, definiu-se um parâmetro `OrphanRemoval = true`, que assegura a remoção das facturas associadas a um dado funcionário, caso este seja eliminado.

A restante estrutura das entidades foi definida de modo a incluir os métodos responsáveis pela escrita (`set`) e leitura (`get`) da informação armazenada. Nos casos em que existe declaração de relações entre entidades, definiram-se ainda os métodos que permitem adicionar (`add`) informação referente à entidade correspondentea nível hierárquico. Por exemplo, retomando a demonstração acima realizada, acrescentou-se o método `addFactura` aos demais, tendo em vista a associação de uma factura ao respectivo funcionário, aquando da sua emissão.

```
public java.lang.String getId() {
    return idfunc;
}

public void setId(java.lang.String idfunc) {
    this.idfunc = idfunc;
}

public Integer getNrVendido() {
    return nrVendido;
}

public void setNrVendido(Integer nrVendido) {
    this.nrVendido = nrVendido;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public Collection getFactura() {
    return facturas;
}

public void setFactura(Collection facturas) {
    this.facturas = facturas;
}
```

```
public void addFactura(Factura receipt) {
    getFactura().add(receipt);
}
```

3 Session Beans

Uma vez implementadas as entidades, o próximo passo lógico seguido foi o de construir os Session Beans capazes de executar as tarefas de negócio no lugar do cliente, ocultando-lhe, desta forma, a complexidade da aplicação e garantindo-lhe, em simultâneo, o acesso à mesma. Optou-se, neste ponto, por utilizar apenas Stateless Session Beans, na medida em que não se punha a necessidade de manter o estado específico do cliente por mais tempo do que aquele correspondente à duração da invocação de métodos. As sessões criadas dizem então respeito às tarefas que se pretendiam passíveis de realizar sobre a aplicação, que neste caso passavam por tarefas de gestão — como sejam o povoamento das entidades através da leitura da informação contida nos ficheiros `.csv` e a inserção e remoção de instâncias de entidade —, de obtenção de estatísticas e de compra — quer *online*, quer directamente nas dependência físicas da farmácia —, correspondendo respectivamente a:

1. GestaoFarma;
2. Estatisticas;
3. Loja.

3.1 `GestaoFarma`

Esta sessão implementa, consoante mencionado anteriormente, métodos que permitem inserir e remover entidades do tipo `Medicamento`, `Funcionario`, `Fornecedor`, `Utilizador` e `Factura`, bem como o método que permite efectivar uma encomenda e os métodos responsáveis por povoar as entidades, aquando da inicialização da aplicação, a partir da leitura dos ficheiros `.csv` fornecidos aquando da realização do trabalho prático anterior. Abaixo, apresentam-se exemplos de cada um dos métodos referidos.

1. Inserção de um novo medicamento.

```
@Override
    public boolean addMedicamento(String idmed, String nome_comercial, String laboratorio,
    Double pvp, Integer stock, Integer vezesVendido) {
        Medicamento med = new Medicamento();
        med.setIdmed(idmed);
        med.setNome_comercial(nome_comercial);
        med.setLaboratorio(laboratorio);
        med.setPvp(pvp);
        med.setStock(stock);
        med.setVezesVendido(vezesVendido);
        em.persist(med);
        return true;
    }
```

2. Remoção de um medicamento previamente existente.

```
@Override
```

```
    public boolean rmMedicamento(String idmed) {
        Medicamento med = em.find(Medicamento.class, idmed);
        em.getTransaction().begin();
        em.remove(med);
        em.getTransaction().commit();
        return true;
    }
```

3. Encomenda de um medicamento.

```
@Override
    public void encomenda(String idmed, String idforn) {
        Medicamento ordered = em.find(Medicamento.class, idmed);
        if (emAlerta().contains(ordered)) {
            ordered.setStock(ordered.getStock() + 3);
            incrQtFornecida(idforn);
        }
    }
```

4. Leitura dos ficheiros `.csv` para a camada de persistência.

```
@Override
    public void readMedicamentos() {
        try {

            BufferedReader br = new BufferedReader(new FileReader(MED_FILE));
            String line;

            br.readLine();
            br.readLine();//ignorar as duas primeiras linhas

            while ((line = br.readLine()) != null) {
                String[] values = line.split(SEPARADOR2);
                String idmed = values[0];
                String nome_comercial = values[1];
                String laboratorio = values[7];

                String tmp = values[8].replace(",", ".");
                Double pvp = Double.parseDouble(tmp);
                Integer stock = (int) (Math.random() * 25) + 1;
                Integer vezesVendido = 0;

                addMedicamento(idmed, nome_comercial, laboratorio, pvp, stock, vezesVendido);
            }
        } catch (IOException ex) {
```

```
            Logger.getLogger(GestaoFarma.class.getName()).log(Level.SEVERE, null, ex);
        }

    }
```

A inserção é efectuada inicializando uma entidade e procedendo, de seguida, à definição de cada uma das suas propriedades através da aplicação de `sets`. Por fim, recorrendo ao Entity Manager `em` — inicializado na sessão sob a anotação `@Persistence Context` —, é aplicado um `persist`, que garante a persistência da entidade construída na base de dados da aplicação. Por sua vez, a remoção implica que se procure a entidade alvo, por intermédio de um `find` direccionado à chave primária da 'tabela', que é subsequentemente removida através de métodos do Entity Manager que culminam num `commit` da operação. O método que possibilita a encomenda de medicamentos sempre que o respectivo *stock* se encontre abaixo do limite estabelecido (3 unidades) assenta também sobre uma procura, procedendo depois a uma verificação relativamente à presença ou não do medicamento especificado na lista de medicamentos em alerta. Por último, a persistência da informação contida nos ficheiros `.csv` foi efectuada de forma semelhante àquela utilizada no trabalho prático anterior, com a particularidade de os Hash Maps serem substituídos por entidades, isto é, cada linha do ficheiro é guardada na forma de uma linha de 'tabela' e assim armazenada na camada de persistência. Releve-se que os ficheiros `.csv` foram alojados na raiz da pasta `PharmaApp-ejb`, sendo necessário fornecer à aplicação o caminho completo para os mesmos de forma a contornar erros de 'File not Found'.

3.2 Estatisticas

Os métodos relacionados com a obtenção de estatísticas de vendas — nomeadamente vendas diárias, semanais, mensais e anuais — e dos *tops* de vendas de medicamentos, utilizadores e fornecedores foram definidos numa sessão dedicada. Os trechos de código originalmente construídos para a obtenção dos *tops* são, aqui, acompanhados da correcção de uma incongruência que se detectou após a entrega do Projecto, que se deveu à não familiarização com a temática das *queries*, no contexto da plataforma J2EE.

1. Obtenção das vendas diárias.

```
@Override
    public double vendasDiarias(int d, int m, int a) {
        double soma=0;
        List<Factura> queryDia = em.createQuery("SELECT fact FROM Factura fact WHERE
         cal.get(Calendar.DAY_Of_MONTH) = :dia, cal.get(Calendar.MONTH) = :mes,
         cal.get(Calendar.YEAR) = :ano").setParameter("dia",d).setParameter("mes", m)
         .setParameter("ano", a).getResultList();
        for(int i=0; i < queryDia.size(); i++) {
            soma = soma + queryDia.get(i).getDespesa();
        }
        return soma;
    }
```

2. Obtenção do *top* 100 de medicamentos inicialmente idealizada.

```
@Override
    public List<Order> top100m() {

        CriteriaBuilder cb = em.getCriteriaBuilder();

        CriteriaQuery<Medicamento> qtop100 = cb.createQuery(Medicamento.class);
        Root<Medicamento> med = qtop100.from(Medicamento.class);
        qtop100.select(med);
        qtop100.orderBy(cb.desc(med.get("vezesVendido")));

        List<Order> medOrdem = qtop100.getOrderList().subList(0, 99);

        return medOrdem;
    }
```

3. Obtenção do *top* 100 de medicamentos, após correcção.

```
@Override
    public List<Medicamento> top100m() {

        CriteriaBuilder cb = em.getCriteriaBuilder();

        CriteriaQuery<Medicamento> qtop100 = cb.createQuery(Medicamento.class);
        Root<Medicamento> med = qtop100.from(Medicamento.class);
        qtop100.select(med);
        qtop100.orderBy(cb.desc(med.get("vezesVendido")));

        Query qmed100 = em.createQuery(qtop100);
        List<Medicamento> qtop100m = qmed100.getResultList();

        List<Medicamento> top100m = qtop100m.subList(0, 99);

        return top100m;
    }
```

As vendas diárias são, então, obtidas por intermédio da criação de uma *query*, na qual se procede à aplicação de uma restrição que permite seleccionar apenas as facturas cuja data corresponda aos dias, mês e ano passados como parâmetros, neste caso específico. De seguida, a lista resultante desta selecção é percorrida iterativamente no sentido de obter a soma dos montantes da totalidade das facturas, que é finalmente retornada. No que concerne a obtenção dos *tops*, recorreu-se à *interface* CriteriaQuery, que define funcionalidades específicas para *queries* de alto nível, necessárias à construção de listas ordenadas, das quais se pretende extrair o número de objectos correspondente à extensão dos *tops*. Assim, numa primeira fase, é criada uma CriteriaQuery à qual é posteriormente aplicado um método de ordenação `orderBy`, que posiciona os medicamentos por ordem decrescente relativamente ao número de vendas, e da qual é derivada

uma sub-lista que abarca os índices correspondentes ao número de objectos pretendidos. Releve-se que a ideia inicialmente implementada não foi correctamente interpretada, ao nível da estrutura final que retorna o *top*, pelo que, após a percepção da falha, se passou à sua correção, conforme exemplificado acima.

3.3 `Loja`

A sessão relativa à loja implementa as funcionalidades que permitem efectuar compras de medicamentos e, à semelhança do trabalho prático anterior, divide-se em dois métodos — `compra` e `compraOnline` —, complementados por quatro métodos auxiliares — `incrVezesVendido`, `incrNrVendido`, `incrQtComprada` e `actualizaStock`. A construção dos métodos referidos teve como base um raciocínio semelhante àquele adoptado nos métodas da sessão `GestaoFarma`, recorrendo a `sets` e à noção de persistência, sendo que, a cada compra, são chamados os métodos de incrementação e actualização — que permitem manter a dinâmica e a coerência do sistema farmacêutico e se regem pela chamada ao método `find` do Entity Manager, com base na chave primária das entidades implicadas.

1. Compra de medicamentos *online*.

```
@Override
    public void compraOnline(String idmed, String idutil) {

            incrVezesVendido(idmed);
            incrQtComprada(idutil);
            actualizaStock(idmed);

            Medicamento medVendido = em.find(Medicamento.class, idmed);
            Double despesa = medVendido.getPvp();
            Factura receipt = new Factura(idutil, "online", idmed, despesa);
            receipt.setCliente(idutil);
            receipt.setVendedor("online");
            receipt.setMedicamento(idmed);
            receipt.setDespesa(despesa);
            em.persist(receipt);
    }
```

2. Compra de medicamentos presencial.

```
@Override
    public void compra(String idmed, String idfunc, String idutil) {

            incrVezesVendido(idmed);
            incrNrVendido(idfunc);
            incrQtComprada(idutil);
            actualizaStock(idmed);

            Medicamento medVendido = em.find(Medicamento.class, idmed);
```

```
            Double despesa = medVendido.getPvp();
            Factura receipt = new Factura(idutil, idfunc, idmed, despesa);
            receipt.setId(idutil);
            receipt.setVendedor(idfunc);
            receipt.setMedicamento(idmed);
            receipt.setDespesa(despesa);
            em.persist(receipt);
    }
```

3. Métodos Auxiliares.

```
public void incrVezesVendido(String idmed) {
        Medicamento sold = em.find(Medicamento.class, idmed);
        sold.setVezesVendido(sold.getVezesVendido() + 1);
    }

    public void incrNrVendido(String idfunc) {
        Funcionario cashier = em.find(Funcionario.class,idfunc);
        cashier.setNrVendido(cashier.getNrVendido() + 1);

    }

    public void incrQtFornecida(String idforn) {
        Fornecedor suplier = em.find(Fornecedor.class,idforn);
        suplier.setQtFornecida(suplier.getQtFornecida() + 1);
    }

    public void incrQtComprada(String idutil) {
        Utilizador client = em.find(Utilizador.class,idutil);
        client.setQtComprada(client.getQtComprada() + 1);
    }

    public void actualizaStock(String idmed) {
        Medicamento sold = em.find(Medicamento.class,idmed);
        sold.setStock(sold.getStock() - 1);
    }
```

4 Clientes

A construção de um novo cliente implica a criação de um novo projecto, que poderá ou não encontrar-se associado à aplicação principal, consoante a versão do servidor `GlassFish` utilizada. Pese embora, no caso específico da utilização do `GlassFish 3.1`, se torne desnecessário efectuar o *deploy* e correr a aplicação do cliente como parte integrante da aplicação empresarial, e apesar de a versão aplicada no âmbito do presente trabalho ser a `3.1.2`, os clientes gerados foram opcionalmente associados à aplicação empresarial. Os referidos clientes criam uma dinâmica de interacção através do acesso directo aos Enterprise Beans que correm na camada de negócio. Neste sentido, para que esta aplicação fosse capaz de referenciar essas estruturas, a biblioteca que contém as *interfaces* remotas foi adicionada ao caminho do projecto, processo viabilizado através da classe `main` e da implementação do seguinte trecho de código — a título de exemplo, referente ao cliente `GestaoCli` —, que permite chamar o Enterprise Bean pretendido:

```
@EJB
private static GestaoFarmaRemote gestaoFarma;
```

Especificamente, os clientes criados neste contexto obedecem a diferentes perfis, concordantes com as funcionalidades da aplicação às quais estes têm acesso e direccionadas à ideia de que um cliente não possa interferir nos métodos definidos em *interfaces* que estejam fora da esfera de acção que lhe é destinada, sendo:

1. `EstatisticasCli`: cliente que obtém os *tops* enunciados em 3.2, bem como as vendas diárias, semanais, mensais e anuais;
2. `GestaoCli`: cliente com a capacidade de adicionar e remover entidades da camada de persistência, e listar medicamentos em alerta, efectuando, neste seguimento, a sua encomenda;
3. `LojaCli`: cliente que pode efectuar a compra — *online* e presencialmente;
4. `LerCli`: cliente que procede à leitura dos ficheiros `.csv` e popula a camada de peristência, por intermédio da criação das entidades que deste derivam.

5 Errata

Por lapso, no Session Bean `GestaoFarma`, consta um excerto de código respeitante à inicialização dos Hash Maps utilizados no trabalho prático anterior. Este trecho não deve, portanto, ser tido em consideração.

```
public static HashMap<String, Fornecedor> fornecedores;
public static HashMap<String, Funcionario> funcionarios;
public static HashMap<String, Medicamento> medicamentos;
public static HashMap<String, Utilizador> utilizadores;
public static HashMap<String, Factura> livroFacturas;
```

6 Conclusão

Quanto à arquitectura de Java EE, conclui-se que se torna necessário ter, em primeiro lugar,as `entities` pois são o ponto de partida para a aplicação. De seguida devem ser construídas as `sessions` com as respectivas interfaces e, por fim, as aplicações-cliente. Considera-se que esta tecnologia simplifica o trabalho relativo à persistência dos dados, no entanto requere um maior conhecimento sobre as funcionalidades da tecnologia usada e ainda um melhor compreensão sobre o funcionamento de Bases de Dados. Não foi possível popular as tabelas devido a erros de SQL que não fomos capazes de resolver, pelo que não foi possível testar a implementação com sucesso.

No cômputo geral, considera-se que o presente trabalho contribuiu para a consolidação dos conhecimentos adquiridos no decorrer das aulas teórico-práticas, representando um instrumento inegável de aprendizagem no que diz respeito à linguagem de programação `Java`, com especial atenção para as funcionalidades de `Java EE`.