Universidade do Minho
Escola de Engenharia

Programação Orientada aos Objetos
LEI - 2022/2023

Délio Alves - a94557
Martim Ribeiro - a96113
Gustavo Barros - a100656

14 de maio de 2023

Índice

1 Introdução

Este trabalho consiste na implementação de um sistema de *marketplace* que permite a compra e venda de artigos novos e usados de vários tipos. Neste sistema, existem utilizadores registados, que tanto podem adicionar artigos para venda como optar pela compra de artigos disponíveis. Os artigos podem ser de vários tipos, e o sistema oferece várias funcionalidades, como consultar estatísticas e devolver produtos.

2 Diagrama de classes

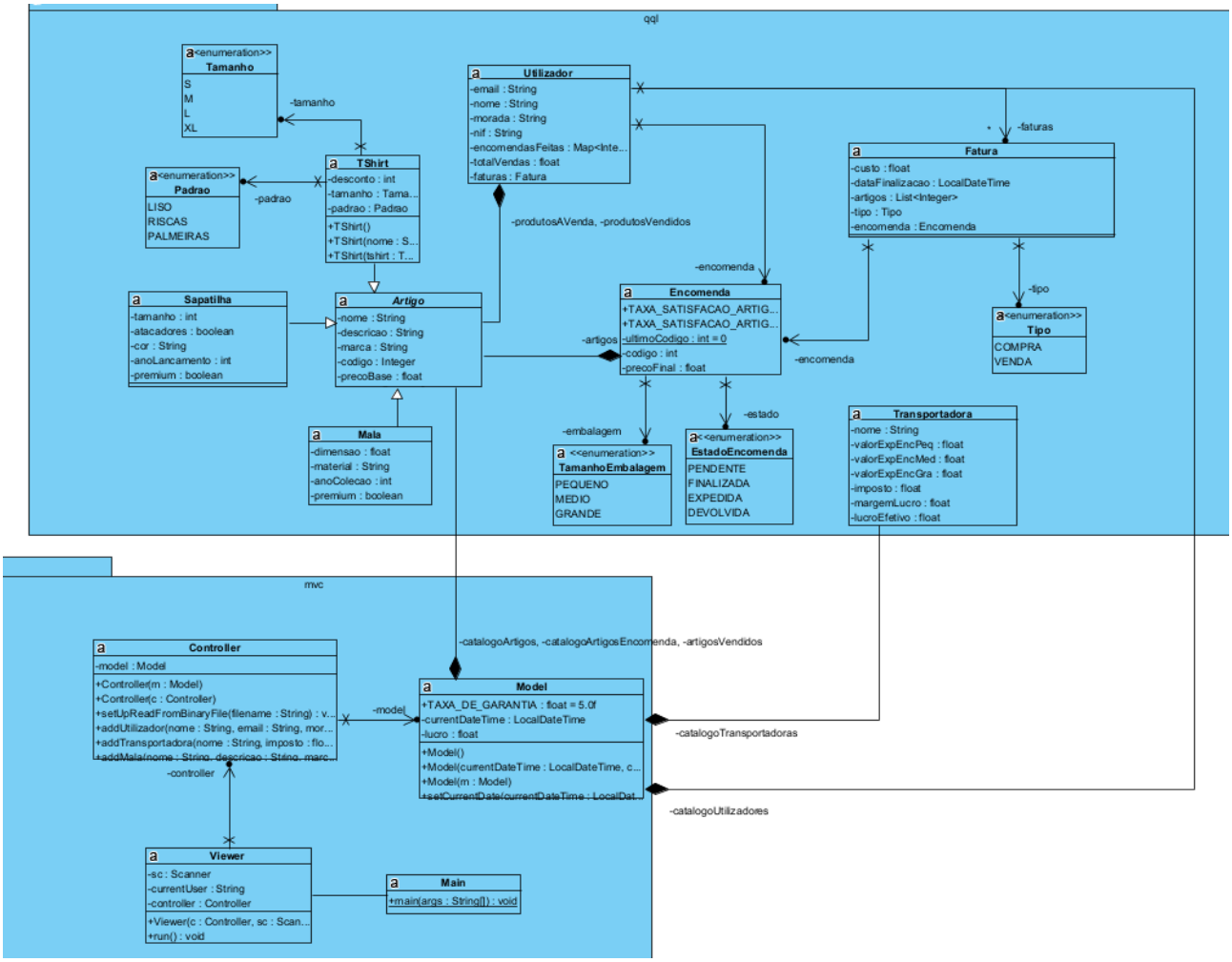

Figure 1: Diagrama de classes

3 Arquitetura de classes

3.1 Artigo

```
public abstract class Artigo implements Serializable {
    private String nome;
    private String descricao;
    private String marca;
    private Integer codigo;
    private float precoBase;
    private float correcaoPreco;
    private float custosExpedicao;
    private float precoFinal;
    private boolean novo;
    private int numDonos;
    private String condicao;
    private String transportadora;
    private int stock;
    private String vendedor;
```

Figure 2: Variáveis de instância de classe *Artigo*

A classe é composta pelas informações em comum entre os diferentes artigos.

- **nome:** uma String que representa o nome do artigo.
- **descricao:** uma String que representa o nome do artigo.
- **marca:** uma String que indica a marca do artigo.
- **codigo:** um Integer que representa o código do artigo.
- **precoBase:** um float que indica o preço base do artigo.
- **correcaoPreco:** um float que representa a correção de preço do artigo.
- **custosExpedicao:** um float que representa os custos de expedição do artigo.
- **precoFinal:** um float que armazena o preço final calculado do artigo.
- **novo:** um booleano que indica se o artigo é novo ou não.
- **numDonos:** um inteiro que representa o número de donos do artigo.
- **condicao:** uma String que descreve a condição atual do artigo.
- **transportadora:** uma String que indica a transportadora associada ao artigo.
- **stock:** um inteiro que representa a quantidade de itens em estoque.
- **vendedor:** Indica o Vendedor do artigo.

Essa classe serve como uma estrutura básica para representar um artigo genérico e pode ser estendida por outras classes mais específicas que definem comportamentos adicionais ou especializados.

3.1.1 Sapatilha

```
public class Sapatilha extends Artigo implements Serializable {
    private int tamanho;
    private boolean atacadores;
    private String cor;
    private int anoLancamento;
    private boolean premium;
```

Figure 3: Variáveis de instância de classe *Sapatilha*

- **tamanho:** um inteiro que representa o tamanho da sapatilha.
- **atacadores:** um booleano que indica se a sapatilha possui atacadores ou não.
- **cor:** uma String que representa a cor da sapatilha.
- **anoLancamento:** um inteiro que representa o ano de lançamento da sapatilha.
- **premium:** um booleano que indica se a sapatilha é premium ou não.

A classe *Sapatilha* representa um tipo específico de artigo, uma sapatilha, que possui características adicionais, como tamanho, presença de atacadores, cor, ano de lançamento e se é premium ou não. A classe herda as variáveis e métodos da classe pai *Artigo* e adiciona as especificidades relacionadas a sapatilhas.

3.1.2 Mala

```
public class Mala extends Artigo implements Serializable {
    private float dimensao;
    private String material;
    private int anoColecao;
    private boolean premium;
    private float valorizacao;
```

Figure 4: Variáveis de instância de classe *Mala*

- **dimensao:** um float que representa a dimensão da mala.
- **material:** uma String que representa o material da mala.
- **anoColecao:** um inteiro que representa o ano da coleção da mala.
- **premium:** um booleano que indica se a mala é premium ou não.
- **valorizacao:** um float que representa a valorização da mala.

A classe *Mala* representa um tipo específico de artigo, uma mala, que possui características adicionais, como dimensão, material, ano da coleção, se é premium e sua valorização. A classe herda as variáveis e métodos da classe pai *Artigo* e adiciona as especificidades relacionadas a malas.

3.1.3 Tshirt

```
public class TShirt extends Artigo implements Serializable {
    public enum Tamanho {
        S,
        M,
        L,
        XL
    }

    public enum Padrao {
        LISO,
        RISCAS,
        PALMEIRAS
    }

    private Tamanho tamanho;
    private Padrao padrao;
    private int desconto;
```

Figure 5: Variáveis de instância de classe *Tshirt*

- **tamanho:** um enum do tipo *Tamanho* que representa o tamanho da T-shirt (S, M, L, XL).
- **padrao:** um enum do tipo *Padrao* que indica o padrão da T-shirt (LISO, RISCAS, PALMEIRAS).
- **desconto:** um inteiro que representa o desconto aplicado à T-shirt.

A classe *TShirt* é uma subclasse da classe Artigo. Ela representa uma T-shirt e adiciona características específicas, como tamanho, padrão e desconto. A classe *TShirt* herda as propriedades e métodos da classe Artigo, permitindo gerenciar e processar informações sobre T-shirts.

3.2 Encomenda

```java
public class Encomenda implements Serializable {

    public enum TamanhoEmbalagem {
        PEQUENO,
        MEDIO,
        GRANDE
    }

    public enum EstadoEncomenda {
        PENDENTE,
        FINALIZADA,
        EXPEDIDA,
        DEVOLVIDA
    }

    public final float TAXA_SATISFACAO_ARTIGO_NOVO = 0.5f;
    public final float TAXA_SATISFACAO_ARTIGO_USADO = 0.25f;

    private static int ultimoCodigo = 0;

    private int codigo;
    private List<Integer> artigos;
    private TamanhoEmbalagem embalagem;
    private float precoFinal;
    private EstadoEncomenda estado;
    private LocalDateTime dataCriacao;
    private LocalDateTime dataFinalizacao;
    private LocalDateTime dataPrevistaEntrega;
```

Figure 6: Variáveis de instância de classe *Encomenda*

- **codigo:** o código único da encomenda.
- **artigos:** um mapa que associa os códigos dos artigos na encomenda com suas quantidades.
- **embalagem:** um enum do tipo *TamanhoEmbalagem* que representa o tamanho da embalagem (PEQUENO, MEDIO, GRANDE).
- **precoFinal:** o preço final da encomenda.
- **estado:** um enum do tipo *EstadoEncomenda* que representa o estado da ecnomenda (PENDENTE, FINALIZADA, EXPEDIDA, DEVOLVIDA).
- **dataCriacao:** a data e hora de criação da encomenda.
- **dataFinalizacao:** a data e hora em que a encomenda foi finalizada.
- **dataPrevistaEntrega:** a data e hora prevista para a entrega da encomenda.

Estas variáveis são responsáveis por armazenar informações relevantes sobre uma encomenda, como os artigos presentes, o tamanho de embalagem, o preço final, o estado e datas importantes.

A classe *Encomenda* possui métodos para adicionar e remover artigos da encomenda, calcular o tamanho da embalagem com base na quantidade de artigos, definir o preço final da encomenda, definir o estado da encomenda, definir a data de finalização e a data prevista de entrega, e realizar a devolução da encomenda se estiver no estado expedida e dentro do prazo de 48 horas.

3.3 Utilizador

```
public class Utilizador implements Serializable {

    private String email;
    private String nome;
    private String morada;
    private String nif;
    private Map<Integer, Artigo> produtosAVenda;
    private Map<Integer, Artigo> produtosVendidos;
    private Map<Integer, Encomenda> encomendasFeitas;
    private List<Fatura> faturas;
    private Encomenda encomenda;
    private float totalVendas;
```

Figure 7: Variáveis de instância de classe *Utilizador*

- **email:** o endereço de e-mail do utilizador.
- **nome:** o nome do utilizador.
- **morada:** a morada do utilizador.
- **nif:** o número de identificação fiscal do utilizador.
- **produtosAVenda:** um *Map* que associa os códigos dos artigos que o utilizador tem à venda com os próprios artigos.
- **produtosVendidos:** um *Map* que associa os códigos dos artigos que o utilizador já vendeu com os próprios artigos.
- **encomendasFeitas:** um *Map* que associa os códigos das encomendas feitas pelo utilizador com as próprias encomendas.
- **faturas:** uma lista de faturas relacionadas com as vendas do utilizador.
- **encomenda:** um objeto da classe *Encomenda* que contém informações sobre a encomenda.
- **totalVendas:** o valor total das vendas realizadas pelo utilizador.

Essas variáveis armazenam informações relevantes sobre um utilizador, incluindo dados pessoais, os produtos que ele tem à venda, os produtos que ele já vendeu, as encomendas feitas, as faturas geradas e o valor total das vendas. Essas informações permitem o gerenciamento das atividades do utilizador, como a adição/remoção de artigos, o processamento de encomendas e o cálculo das vendas realizadas.

3.4 Fatura

```
public class Fatura implements Serializable {

    public enum Tipo {
        COMPRA,
        VENDA
    }

    private float custo;
    private Tipo tipo;
    private Encomenda encomenda;
    private LocalDateTime dataFinalizacao;
    private List<Integer> artigos;
```

Figure 8: Variáveis de instância de classe *Fatura*

- **custo:** um float que representa o custo da fatura.
- **tipo:** um enum *Tipo* que representa o tipo da fatura (COMPRA ou VENDA).
- **encomenda:** um objeto do tipo *Encomenda* que representa a encomenda relacionada à fatura.
- **dataFinalizacao:** que representa a data de finalização da fatura.
- **artigos:** uma lista de inteiros que representa os identificadores dos artigos relacionados à fatura.

Essas variáveis armazenam informações importantes sobre a fatura, como o valor total, o tipo de transação, a encomenda associada (se houver), a data de finalização e os códigos dos artigos envolvidos. Essas informações são úteis para registar as transações ocorridas no sistema.

3.5 Transportadora

```
public class Transportadora implements Serializable {
    private String nome;
    private float valorExpEncPeq;
    private float valorExpEncMed;
    private float valorExpEncGra;
    private float imposto;
    private float margemLucro;
    private float lucroEfetivo;
    private String formula;
```

Figure 9: Variáveis de instância de classe *Transportadora*

- **nome:** o nome da transportadora.
- **valorExpEncPeq:** o valor de expedição para encomendas de tamanho pequeno.
- **valorExpEncMed:** o valor de expedição para encomendas de tamanho médio.
- **valorExpEncGra:** o valor de expedição para encomendas de tamanho grande.
- **imposto:** a taxa de imposto aplicada às expedições.
- **margemLucro:** a margem de lucro adicionada ao cálculo do preço de expedição.
- **lucroEfetivo:** o lucro efetivo da transportadora.
- **formula:** a fórmula utilizada para calcular o preço de expedição.

Essas variáveis armazenam informações importantes sobre a transportadora, como o nome, os valores de expedição para cada tamanho de embalagem, o imposto aplicado, a margem de lucro, o lucro efetivo e a fórmula usada para calcular o preço de expedição.

A classe também possui métodos para definir e obter os valores das variáveis, bem como um método para calcular o preço de expedição com base no tamanho da embalagem usando a fórmula especificada.

4 MVC (Model-View-Controller)

O nosso grupo decidiu seguir um modelo *Model-View-Controller* no trabalho, dados os vários aspetos positivos que o mesmo traz. Ao dividir o código nestas 3 componentes, passa a existir uma clara estruturação do projeto, o que facilita a manuntenção e futura evolução do programa.

4.1 Viewer

O código do Viewer é responsável pela interação com o cliente, ou seja, exibe menus e recebe *inputs* do utilizador. Consoante os inputs, estes são enviados para o Controller, que comunica com o Model de modo a retornar a informação necessária.

4.2 Controller

A classe Controller é responsável por coordenadar as interações entre o Viewer e o Model.

4.3 Model

A classe Model representa o modelo principal do sistema e contém uma série de métodos que lidam com operações relacionadas a utilizadores, artigos, encomendas, faturas e transportadoras.

- *Métodos relacionados a encomendas.* Esses métodos lidam com a adição, remoção e exibição de artigos em encomendas, verificação do estado das encomendas, cálculo de preços finais, finalização de compras e devolução de encomendas.
- *Métodos relacionados a utilizadores.* Esses métodos envolvem verificação de existência de utilizadores, obtenção de informações sobre vendas de um determinado utilizador, identificação do maior vendedor, obtenção de valores de vendas de um vendedor, verificação se um utilizador possui faturas, exibição de faturas e obtenção de uma lista dos principais vendedores durante um período específico.
- *Métodos relacionados a transportadoras.* Esses métodos envolvem verificação de existência de transportadoras, identificação da transportadora com o maior lucro, obtenção de valores de lucro de uma transportadora, alteração de valores e fórmulas relacionadas a transportadoras, e obtenção de informações sobre os valores e fórmulas utilizados por uma transportadora.
- *Outros métodos:* **avancar** e **startSimulation**. Esses métodos permitem avançar o tempo do sistema em um determinado número de horas e iniciar uma simulação que verifica e atualiza o estado das encomendas expedidas.

Em geral, o código é organizado de forma a separar as diferentes funcionalidades do sistema em métodos individuais. Usamos várias estruturas de dados, como mapas e listas, para armazenar e manipular informações relevantes. Além disso, os métodos são responsáveis por interagir com outras classes e objetos do sistema para realizar operações específicas.

4.4 Descrição e Funcionamento da Aplicação

Inicalmente, é perguntado ao utilizador como pretende carregar a informação para o programa.

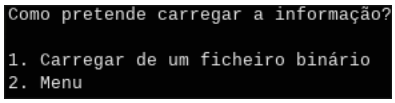

Figure 10: Menu inicial

De seguida, caso o utilizador escolha carregar a informação por menu, segue-se o seguinte menu. De notar que é também é possível aceder a este menu caso a informação seja carregada por ficheiro.

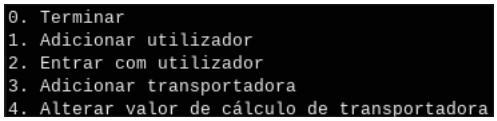

Figure 11: Adicionar informação

Caso o utilizador inicie sessão ou crie conta, é apresentado com um menu onde pode realizar várias operações.

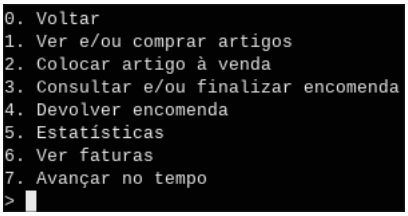

Figure 12: Sessão iniciada

5 Funcionalidades do programa

5.1 Requisitos base de gestão das entidades

O nosso programa cumpre a implementação de todos os requisitos base referidos no enunciado: criar utilizadores (vendedores/compradores), artigos, transportadoras e fazer encomendas.

Assim, é possível, através do menu principal do nosso projeto, adicionar utilizadores e transportadoras. Tendo sessão iniciada com um utilizador, este tem a possibilidade de adicionar um artigo ao sistema e colocá-lo à venda, encomendar 1 ou mais artigos que estejam disponíveis (e posteriormente devolvê-los), bem como ver as suas faturas e estatísticas da Vintage. Pode, ainda, avançar no tempo.

5.2 Efetuar estatísticas sobre o estado do programa

Novamente, o nosso programa cumpre a implementação dos requisitos pedidos. É possível tanto avançar no tempo, como consultar estatísticas sobre o programa.

No que toca ao avançar do tempo, este é feito em horas, pelo que o utilizador indica o número que horas que quer simular. Após isto, são realizadas todas as tarefas necessárias: as encomendas são despachadas e as faturas são emitidas.

Quanto às estatísticas, foram todas implementadas, e são acessíveis através do menu de sessão iniciada.

1.1. Qual é o vendedor que mais facturou desde sempre

O programa chama o método *getMaiorVendedor()* para obter o nome do vendedor que mais faturou. Em seguida, é chamado o método *getMaiorVendedorValor()* para obter o valor total das vendas desse vendedor.

O programa verifica se o valor retornado é igual a 0.0, o que indica que não houve vendas na Vintage. Nesse caso, é exibida uma mensagem a informar ao usuário que ainda não houve vendas e, portanto, não é possível calcular o vendedor que mais faturou.

Se o valor retornado for diferente de 0.0, significa que houve vendas na Vintage. O programa então exibe o nome do vendedor que mais faturou e o valor total das vendas realizadas pelo vendedor.

1.2. Qual é o vendedor que mais facturou num período

O programa pede ao utilizador as datas inicial e final desejadas no formato "yyyy-mm-dd hh". Em seguida, verifica se as datas fornecidas são válidas.

Se as datas forem válidas, o programa chama os métodos *getMaiorVendedorPeriodo()* e *getMaiorVendedorValorPeriodo()* para obter o nome e o valor total das vendas do vendedor que mais faturou durante o período especificado.

Se o valor retornado for igual a 0.0, indica que não houve vendas durante o período, e o programa exibe uma mensagem apropriada.

Caso contrário, o programa exibe o nome do vendedor e o valor total das vendas realizadas por ele durante o período selecionado.

2. Qual o transportador com maior volume de facturação

Primeiro lugar, o programa chama o método *getMaiorTransportadora()* para obter o nome da transportadora que mais faturou. Em seguida, o programa chama o método *getMaiorTransportadoraValor()* passa-lhe o nome da transportadora como argumento, para obter o valor total das vendas realizadas por essa transportadora.

Se o valor retornado for igual a 0.0, significa que ainda não houve vendas registradas na Vintage, e o programa exibe uma mensagem a informar que não é possível calcular a transportadora que mais faturou.

Caso contrário, o programa exibe o nome da transportadora e o valor total das vendas realizadas por ela.

3. Listar as encomendas emitidas por um vendedor

Primeiro, o programa solicita que o usuário digite o nome do vendedor pretendido. Em seguida, o programa chama o método *checkUtilizadorExiste()* passando o nome do vendedor como argumento para verificar se o vendedor existe no sistema.

Se o resultado da verificação for falso, ou seja, se não existir um vendedor com o nome indicado, o programa exibe uma mensagem informando que não existe um vendedor com o nome indicado.

Caso contrário, o programa chama o método *getVendasUtilizador()* passando o nome do vendedor como argumento para obter a lista de vendas desse vendedor. Se a lista não estiver vazia, o programa exibe o resultado, ou seja, a lista de vendas do vendedor. Se a lista estiver vazia, o programa exibe uma mensagem informando que o vendedor indicado ainda não tem nenhuma venda.

4. Fornecer uma ordenação dos maiores compradores/vendedores do sistema durante um período a determinar

O programa pede ao utilizador que indique a data inicial e a data final pretendida, com o seguinte o formato "yyyy-mm-dd hh". Em seguida, ele lê as datas digitadas pelo usuário.

Após isso, é feita uma verificação para garantir que a data final seja posterior à data inicial. Se as datas forem inválidas, ou seja, se a data final não for posterior à data inicial, o programa exibe a mensagem "As datas dadas são inválidas".

Caso as datas sejam válidas, o programa chama o método *getListaMaioresVendedores()* passando as datas iniciais e finais como argumentos. Este método retorna um *Map* que contém os nomes dos vendedores/compradores como chaves e o número de vendas como valores.

Em seguida, o programa percorre o *Map* resultante com um loop. Para cada entrada no *Map*, que consiste no nome do vendedor/comprador e no número de vendas, o programa exibe o resultado na forma "nome: número de vendas".

5. Determinar quanto dinheiro ganhou o Vintage no seu funcionamento

O programa chama o método *getLucroVintage()* para obter o lucro total da Vintage. Esse método retorna um valor numérico representando o lucro.

5.3 Alterar os transportadores e prever a noção de Premium

Ao contrário dos anteriores, não foram implementados todos os requisitos pedidos. Foi implementada a existência de produtos *premium*, mas o mesmo não aconteceu para as transportadoras. No entanto, os requisitos identificados em 2.5 (do enunciado) foram satisfeitos: é possível alterar os valores de cálculo das transportadoras.

5.4 Automatizar a simulação

O grupo não conseguiu implementar os requisitos pedidos nesta secção a tempo.

6 Conclusão

Neste trabalho, fomos introduzidos ao paradigma da programação orientada aos objetos e, sobretudo, aos novos padrões de desenvolvimento de aplicações neste paradigma.

Assim, aprofundamos os conceitos de encapsulamento, modularidade e abstração de dados, importantes para garantir uma boa evolução do nosso programa, permitindo acrescentar novas funcionalidades, caso necessário, sem grandes complicações.

Por fim, apesar de não termos conseguido implementar a totalidade dos requisitos pedidos, consideramos que o trabalho está positivo e o que o esforço empenhado compensou pelo conhecimento adquirido.