



Universidade do Minho

Licenciatura em Engenharia Informática

UC de Programação Orientada a Objetos

Ano Letivo de 2022/2023

Trabalho Prático

(Grupo 88)

Rodrigo Monteiro, Miguel Gramoso

e-mail: {a100706,a100835}@alunos.uminho.pt

Abstract. No âmbito da unidade curricular de Programação Orientada a Objetos, este trabalho constrói um sistema de *marketplace* chamado Vintage, onde utilizadores podem comprar artigos, — e, portanto, fazer encomendas — e vender artigos, sendo que estes podem ser novos, usados e/ou premium e de vários tipos (T-Shirts, Malas e Sapatilhas). Cada artigo está associado a uma transportadora, que pode ser especializada em artigos premium, e, por isso, encomendas podem ser enviadas em diferentes *packages*, podendo estes ser enviados e chegarem ao destino em datas diferentes.

Para além disso, é implementado um sistema que permite: avançar no tempo, de modo a simular de forma mais rápida uma situação real; guardar o estado do programa, serializando o *model*; mudar de idioma, através da utilização de *resource bundles*; ler dados de ficheiros *csv* fazendo o *parsing* necessário; e realizar uma simulação através de instruções de um ficheiro.

Keywords: Java · Object Oriented Programming · OOP · Design Patterns

Índice

<u>1</u>	<u>Arquitetura de classes</u>	<u>3</u>
1.1	Artigos	3
1.1.1	Sapatilhas	6
1.1.2	T-Shirt	6
1.1.3	Malas	7
1.2	Utilizadores	7
1.3	Encomendas	7
1.4	Transportadoras	9
1.5	Catálogos	10
1.5.1	Estatísticas	11
1.6	User Manager	12
1.7	Article Factory	13
1.8	App Clock	14
1.9	State	14
1.10	Model	15
1.11	Views	16
1.12	Controllors	17
<u>2</u>	<u>Funcionalidades</u>	<u>18</u>
2.1	Menu inicial	18
2.2	Menu principal	18
2.3	Modo de Utilizador	20
<u>3</u>	<u>Diagrama de classes</u>	<u>22</u>
3.1	Fields	22
3.2	Dependencies	23
3.3	Dependencies (com inner classes)	23
<u>4</u>	<u>Conclusão</u>	<u>24</u>

1 Arquitetura de classes

1.1 Artigos

A vintage especializa-se em artigos de vários tipos, e cada um destes pode ter vários comportamentos. No entanto, todos os artigos partilham um conjunto de características e atributos, por isso foi tomada a decisão de utilizar uma classe abstrata *Article* que possui esse conjunto, juntamente com alguns métodos abstratos que as subclasses têm de implementar, e um *builder* genérico.

Sempre que um artigo é criado, este é depois adicionado ao catálogo de artigos, e este catálogo é responsável por associar um identificador único ao artigo. Este identificador é um UUID, um valor de 128 bits, criado aleatoriamente utilizando a biblioteca `java.io.UUID`.

Um artigo também possui:

- uma descrição;
- uma marca;
- um preço base, isto é, o preço sem nenhuma aplicação de desconto ou acréscimo;
- uma correção de preço, que corresponde à aplicação de possíveis descontos e acréscimos ao preço base;

Além disso, também possui uma classe chamada `UsedBehaviour`. Este nome foi escolhido visto que é uma classe que caracteriza o comportamento relativo ao uso de um artigo. Portanto, foi utilizado um *strategy pattern*, ou seja, um artigo pode *trocar* entre diversos tipos de comportamento, neste caso, entre `RegularUsed` e `NotUsed` (ambos implementam a interface `UsedBehaviour`). Como existem apenas duas estratégias diferentes, e uma é relativa a “não possuir comportamento de *usado*”, pode-se argumentar que é utilizado um *Null Object Pattern*.

A mesma lógica é utilizada para o comportamento premium de um artigo: `RegularPremium` e `NotPremium` implementam a interface `PremiumBehaviour`.

Deste modo, não é necessário criar uma extensa hierarquia de classes para traduzir o problema de um artigo ser premium ou usado (por exemplo, evita-se a criação de `BagPremium`, `BagUsed`, `BagPremiumUsed`, etc.)

Por fim, um artigo também possui uma transportadora, e o UUID do utilizador que vende o artigo.

```
private UUID id;
private String description;
private String brand;
private double basePrice;
private double priceCorrection;
private UsedBehaviour usedBehaviour;
private PremiumBehaviour premiumBehaviour;
private Carrier carrier;
private UUID sellerId;
```

Para além dos *getters*, *setters* e outros métodos necessários, a classe abstrata artigo contém três métodos que são importante destacar, visto que são abstratos:

```
protected abstract String articleType();
public abstract void updatePriceCorrection (int currentYear);
public abstract void updatePremiumPriceCorrection (int currentYear);
```

Cada um destes métodos tem de ser implementado pelas subclasses. Isto é, todas as subclasses têm de implementar um método que retorna o seu tipo em formato de String (“bag”, “shoes”, etc.), um método que calcula a `priceCorrection` do artigo, tendo em conta o comportamento de uso e de premium, e um método que atualiza a `priceCorrection` tendo em conta apenas o comportamento premium, visto que apenas este muda com o passar do tempo (ano da coleção).

```

public interface UsedBehaviour extends Serializable {
    double getUsedPriceCorrection (double basePrice);
    int getNumberOfOwners();
    double getStateOfUse();
}

public class RegularUsed implements UsedBehaviour {
    private int numberOfOwners;
    private double stateOfUse;

    @Override
    public double getUsedPriceCorrection(double basePrice) {
        return ((basePrice/this.numberOfOwners) * this.stateOfUse);
    }
    // ...
}

```

Caso um artigo seja usado, irá possuir a classe `RegularUsed`, e terá modificar a sua `priceCorrection` utilizando o método `getUsedPriceCorrection` que recebe o preço base e com base no número de donos e no estado de uso, devolve um valor a remover. Foi utilizada uma fórmula semelhante à exemplificada no enunciado, porém poderia ser mais complexa ou ainda utilizar outra lógica, por exemplo envolvendo o ano da coleção.

```

public class NotUsed implements UsedBehaviour {
    @Override
    public double getUsedPriceCorrection(double basePrice) {
        return 0;
    }
    //...
}

```

Caso o artigo não seja usado, o valor a remover será zero.

Uma lógica semelhante (mesmo padrão) é utilizada em relação ao comportamento premium:

```

public class RegularPremium implements PremiumBehaviour {
    private double markup;
    private int lastUpdated;
    // ...

    @Override
    public double getPremiumPriceCorrection (double basePrice, int currentYear) {
        double priceCorrection = basePrice * ((currentYear - this.lastUpdated) * this.markup);
        this.lastUpdated = currentYear;
        return priceCorrection;
    }
    // ...
}

```

Cada artigo premium tem uma percentagem de aumento do preço por ano, e o último ano em que o preço foi atualizado, de forma a que o preço seja sempre atualizado corretamente, sem adição de valores já adicionados.

Ao ser criado, o artigo começa sempre com o valor de `lastUpdated` igual ao seu `collectionYear`.

Assim, exemplificando o uso da fórmula, se um artigo tiver 2020 como ano de coleção e o ano atual for 2023, a fórmula fica a seguinte: $basePrice * ((2023 - 2020) * markup)$, e se passar um ano, o preço é atualizado da seguinte maneira: $basePrice * ((2024 - 2023) * markup)$.

Por último, a classe abstrata `Article` possui uma *protected static abstract inner class*, isto é, pode ser usada apenas pelas subclasses e não precisa de uma instância da classe externa para ser instanciada (mas mesmo assim não pode ser instanciada, visto que é abstrata, tal como a classe externa).

```
protected static abstract class ArticleBuilder<B extends ArticleBuilder<B>> {
    protected String description;
    // ...
    public B setDescription(String description) {
        this.description = description;
        return self();
    }
    // ...
    protected abstract Article buildWithoutPriceCorrection();
    protected abstract Article buildWithPriceCorrection();

    @SuppressWarnings("unchecked")
    protected final B self() {
        return (B) this;
    }
}
```

Assim, foi utilizado um *builder pattern* para instanciar qualquer tipo de artigo. No entanto, é um *builder* em geral mais complexo, visto que envolve construir efetivamente qualquer tipo de artigo, envolve uma hierarquia. A restrição `ArticleBuilder<B extends ArticleBuilder` garante que a classe `B` é uma subclasse de `ArticleBuilder`, isto é, o *builder* da subclasse tem de estender `ArticleBuilder` — este *builder* é genérico. Os métodos devolvem sempre o tipo `B`. O método `self()` retorna uma instância da classe em que está com cast para `B`, visto que o compilador não sabe qual é o tipo específico da subclasse `ArticleBuilder` que está a ser utilizado em tempo de compilação. O `@SuppressWarnings("unchecked")` é usado para suprimir o aviso do compilador de que o tipo de objeto retornado pode não ser compatível com o tipo genérico da classe. Isso ocorre porque, em tempo de compilação, o compilador não pode verificar se o tipo retornado é, de facto, uma subclasse, mas em tempo de execução isso é garantido devido à restrição imposta. Uma forma de não utilizar `@SuppressWarnings("unchecked")` seria tornando o método `self()` abstrato, e depois este seria *overridden* pelas subclasses, o que seria mais *type-safe*, no entanto, nesse caso a subclasse teria a possibilidade de retornar algo diferente de *this*.

Continuação da lógica utilizada para o *builder* agora na classe `Bag`:

```
protected static abstract class GenericBagBuilder<B extends GenericBagBuilder<B>>
extends Article.ArticleBuilder<B> {
    protected String material;
    // ...
    public B setMaterial(String material) {
        this.material = material;
        return self();
    }
    // ...
}
```

Aqui `GenericBagBuilder`, também uma classe abstrata, tal como o `ArticleBuilder` possui a restrição que garante que `B` é uma subclasse de `GenericBagBuilder`, e este *extends* `ArticleBuilder`.

```
public static class BagBuilder extends GenericBagBuilder<BagBuilder> {
    public Bag buildWithoutPriceCorrection() {
        return new Bag(this.description, ...);
    }
}
```

Por fim, o tipo B é definido como `BagBuilder` que é subclasse de `GenericBagBuilder` e por isso também subclasse de `ArticleBuilder`. Deste modo, todos os setters, de ambos os builders genéricos, “recebem e retornam” o mesmo tipo. Quando é chamado o `buildWithoutPriceCorrection`, este retorna então o objecto de acordo com os *setters* utilizados. (O `BagBuilder` possui os *setters* das superclasses, que retornam B, ou seja `BagBuilder`).

```
new Bag.BagBuilder()
    .setDescription(description)
    // ...
    .buildWithoutPriceCorrection();
```

Por conseguinte, tendo em conta toda a classe `Article`, foi possível gerar uma arquitetura através de hierarquia e *design patterns*, e, por isso, polimorfismo (todos os tipos de artigo são um `Article`) e abstracção (builder), que fornece flexibilidade para adição de outros tipos de artigos futuramente.

1.1.1 Sapatilhas

A classe sapatilhas possui um tamanho numérico, uma indicação se possuem atacadores (valor booleano), uma cor, ano de lançamento da coleção, e um possível desconto, caso as sapatilhas tenham um tamanho superior a 45 e sejam novas.

```
private int size;
private boolean hasLaces;
private String color;
private int collectionYear;
private double discount;
```

Assim, sempre que o objecto é criado é chamado o seguinte método:

```
private void applyDiscount() {
    if (this.size > 45 && !this.isUsed()) {
        this.setPriceCorrection(this.getPriceCorrection() - this.calculateSizeDiscount());
    }
}
```

1.1.2 T-Shirt

As T-Shirt possuem um tamanho (S, M, L, XL) um padrão (liso, riscas, palmeiras), e um desconto fixo de 50%. Este desconto é aplicado caso a T-Shirt tenha o padrão riscas ou palmeiras e seja usada. Para definir o tamanho e os padrões foram utilizados *enums*:

```
public enum Size { S, M, L, XL }
public enum Pattern { SOLID, STRIPES, PALM_TREES }

private Size size;
private Pattern pattern;
private static final double FIXED_DISCOUNT = 0.5;
```

O método `updatePremiumPriceCorrection`, apesar de poder ser chamado, é vazio, pois sapatilhas não têm comportamento premium definido.

1.1.3 Malas

As malas possuem informação sobre a dimensão (3 dimensões), tendo sempre um desconto inversamente proporcional à dimensão, o material, e o ano da coleção. Para além disso, podem ter o comportamento `RegularPremium` tal como as sapatilhas (mas poderia ter sido criada uma outra estratégia/ comportamento), isto é, o seu preço aumenta X% por ano.

```
private int[] dimensions;
private String material;
private int collectionYear;
private double discount;
```

O cálculo do desconto é feito da seguinte maneira: $((x \times y \times z) \times discount \times basePrice)/100.0$

1.2 Utilizadores

A cada utilizador é atribuído um identificador único, um UUID, tal como os artigos, quando este é adicionado ao catálogo correspondente. Para além disso, a classe `User` possui informação acerca do email, que o utilizador usa para efetuar login (e por isso não podem haver dois utilizadores com o mesmo email). Também possui informação acerca da sua morada através da classe `Residence` que guarda o país, a cidade, a rua e o código postal do utilizador.

A classe `User` também guarda dados acerca do dinheiro que o utilizador ganhou e gastou tanto no total como em dias específicos, através da utilização de um `Map`, cujas chaves são um `Long` correspondente ao *epoch* da data/dia, e os valores são o dinheiro gasto/ ganho respetivo dia.

```
private UUID id;
private String email, name;
private Residence residence;
private double moneyEarned, moneySpent;
private Map<Long, Double> moneySpentByDay, moneyEarnedByDay;
```

Utilizar o *epoch* em vez de `LocalDate` permite, de forma mais fácil, descobrir quanto dinheiro um utilizador ganhou ou gastou num período de tempo, visto que, assim, é possível iterar entre duas datas, ou seja, entre dois `Longs`, e verificar se o `Map` contém a chave a cada iteração, se sim adiciona-se o valor correspondente a essa chave à variável que acumula o valor total relativo ao intervalo de tempo.

```
double acc = 0.0;
long infEpoch = infDate.toEpochDay();
long supEpoch = supDate.toEpochDay();

for (long epoch = infEpoch; epoch <= supEpoch; epoch++)
    if (map.containsKey(epoch))
        acc += map.get(epoch);
```

1.3 Encomendas

As encomendas também possuem um identificador único, UUID, juntamente com um UUID do utilizador que comprou/ encomendou o conjunto de artigos. Possui um de cinco possíveis estados, e este pode mudar ao longo do tempo: *pending*, *finalized*, *sent*, *arrived*, *returned*.

Pending indica que esta classe está a servir como um “carrinho de compras”, isto é, a encomenda ainda não foi finalizada, e o utilizador ainda pode escolher adicionar ou remover artigos. *Finalized* indica que o utilizador já

pagou pela encomenda (neste estado, é ainda possível cancelar a encomenda — não existe estado *canceled*, pois aí a classe passa a ser desnecessária).

Quando a encomenda é marcada como finalizada, a transportadora é responsável por decidir as datas de envio e de quando chega ao destino (também para efeitos de simulação).

Sent indica que a encomenda já foi enviada, e neste estado não já não é possível cancelar a encomenda; também não é possível devolver a encomenda enquanto esta não chegar ao destino.

Arrived indica que a encomenda já chegou ao destino. Neste estado, o utilizador tem 48h, desde que a encomenda chegou, para devolver a encomenda, ficando marcada como *returned*, e nesse caso o sistema faz a gestão necessária de artigos como será visto posteriormente, caso contrário, não é possível devolver a encomenda.

```
public enum State { PENDING, FINALIZED, SENT, ARRIVED, RETURNED }
private UUID id, buyerId;
private Map<UUID, Package> packages;
private double finalPrice;
private State state;
private LocalDateTime creationDate, finalizedDate, arrivedDate;
private int numberOfArticles;
```

Um dos aspetos importantes acerca das encomendas é que possuem uma lista de *packages*. Tal é necessário visto que artigos têm diferentes transportadoras associadas, e devido a isso os artigos são agregados em diferentes conjuntos/*packages*, sendo cada *package* correspondente a uma transportadora, a qual é responsável por o transportar e entregar. Enquanto a encomenda não é finalizada, a classe trata da gestão de artigos por *packages* (reagindo à adição e remoção de artigos), assim, quando é finalizada, a gestão de artigos da encomenda já está pronta.

Um *package* também possui um estado que pode ser: *not sent*, *sent*, *arrived* ou *returned*. Como cada um, mesmo relativos à mesma encomenda, podem possuir diferentes datas de envio e chegada, uma encomenda só é marcada como *sent* se todos os *packages* dessa encomenda estejam marcados como *sent*, e a encomenda só fica marcada como *arrived* caso todos os *packages* estejam marcados como *arrived*.

```
public enum Status {NOT_SENT, SENT, ARRIVED, RETURNED}
private static final double NEW_ARTICLE_FEE = 0.5;
private static final double USED_ARTICLE_FEE = 0.25;
private double vintageProfit;
private Carrier carrier;
private ArrayList<Article> articles;
private Status status;
private int numberOfArticles;
private LocalDateTime sentDate, arrivedDate;
```

Como é possível verificar, cada artigo que seja novo tem uma taxa de satisfação de serviço de 0.5 €, e caso seja usado tem uma taxa de 0.25 €. Esta é a única forma pela qual a Vintage ganha dinheiro neste programa, e o ganho da Vintage relativo a cada *package* é guardado numa variável *vintageProfit* e depois “recolhido” pela classe encomenda.

A encomenda calcula o seu valor final/total recolhendo o preço relativo a todos os seus *packages* — a classe *Package* calcula o seu preço recorrendo aos preços corrigidos dos artigos, à taxa de satisfação de serviço de cada artigo e ao *shipping cost* calculado pela respetiva transportadora.

Por fim, a classe encomenda tem um método que permite retornar informações sobre os seus artigos retornando um *array* de *ArticleInfo*, um *record* que possui o ID do utilizador que compra e vende o artigo, e o artigo em *String*. Este método é útil para disponibilizar apenas a informação necessária à “*front-end*” do programa, e para não se retornar cópias completas de todos os artigos da encomenda.

1.4 Transportadoras

Uma transportadora tanto pode ser “normal” como premium, por isso foi tomada a decisão de criar uma classe abstrata `Carrier` com duas subclasses: `RegularCarrier` e `PremiumCarrier`, sendo que `PremiumCarrier` são transportadoras especializadas em transportar artigos premium e por isso têm uma fórmula diferente para o cálculo do *shipping cost* em relação às transportadoras normais.

A classe abstrata `Carrier` possui um identificador único, UUID, mas também pode servir como identificador o seu nome, visto que quaisquer duas transportadoras não podem ser registadas com o mesmo nome; possui informação acerca dos preços base para encomendas pequenas (um artigo), encomendas médias (dois a cinco artigos) e encomendas grandes (mais de cinco artigos). Possui também um imposto e um valor de lucro multiplicativo que se aplica sobre o preço base, e uma variável que guarda o dinheiro total ganho. Por último, possui um Map cujas chaves são os identificadores das encomendas e os valores são os *packages* relativos às encomendas, de modo a que uma transportadora tenha uma gestão dos *packages* de cada encomenda de que é responsável — no entanto, é uma funcionalidade pouco utilizada.

Como foi referido anteriormente, a transportadora é responsável por definir as datas de envio e de chegada de cada *package*, e com o objetivo de simular uma situação real, e como a transportadora não é real e não pode decidir essas datas, estas são geradas aleatoriamente, com uma duração entre 24h e 48h.

```
public void setPackageDates (Package p, LocalDateTime now) {
    LocalDateTime sentDate = this.decideDate(now);
    LocalDateTime arrivedDate = this.decideDate(sentDate);
    p.setSentDate(sentDate);
    p.setArrivedDate(arrivedDate);
}

private LocalDateTime decideDate (LocalDateTime date) {
    int randomHours = ThreadLocalRandom.current().nextInt(24, 48);
    return date.plus(randomHours, ChronoUnit.HOURS);
}
```

Relativamente ao cálculo de *shipping costs*, uma transportadora normal, `RegularCarrier`, efetua esse cálculo da seguinte maneira:

```
public double calculateShippingCost (Package p) {
    double basePrice;
    int numberOfArticles = p.getNumberOfArticles();

    if (numberOfArticles <= 1) basePrice = this.getBasePriceSmall();
    else if (numberOfArticles <= 5) basePrice = this.getBasePriceMedium();
    else basePrice = this.getBasePriceBig();

    return basePrice * (1 + this.getProfitRate() + this.getTax());
}
```

As `PremiumCarrier` diferem neste aspeto, visto que têm de ter em conta os artigos premium. Portanto, decidimos que uma transportadora premium cobra mais um valor fixo por cada artigo premium, possivelmente devido a maiores garantias de segurança e rapidez na entrega em comparação a uma transportadora normal.

```
basePrice *= (1 + this.getTax() + this.getProfitRate());
for (Article article : p.getArticles())
    if (article.isPremium())
        basePrice += this.additionalPriceForPremiumArticles;
```

1.5 Catálogos

Todos os elementos referidos anteriormente que possuem um identificador único, UUID, possuem também um respetivo catálogo. Percebemos que muitos dos métodos desses catálogos seriam semelhantes, como *remove*, *add*, *get*, *size*, *exists*, *etc.*, por isso foi construída uma classe abstrata e genérica, *GenericCatalog*, cujo tipo, *T*, tem apenas de implementar a interface *HasId* de modo a que seja possível utilizar os métodos *getId* e *setId*.

```
public interface HasId {
    UUID getId();
    void setId (UUID id);
}
```

```
public abstract class GenericCatalog<T extends HasId & Serializable> implements
Serializable {
    protected Map<UUID, T> catalog;

    public GenericCatalog() { this.catalog = new HashMap<>(); }
    // ...
    public void add (T item) {
        UUID uuid = UUID.randomUUID();
        while (this.catalog.containsKey(uuid)) uuid = UUID.randomUUID();
        item.setId(uuid);
        this.catalog.put(uuid, item);
    }

    public void remove (T item) { this.catalog.remove(item.getId()); }
    public T get (UUID id) { return this.catalog.get(id); }
    public boolean exists (UUID id) { return this.catalog.containsKey(id); }
    // ...
}
```

Para além disso, também possui um método que retorna um array de todos os valores do catálogo em strings, o que é útil para a View, e para realizar a opção View Data no menu principal do programa.

Assim, tanto o catálogo de utilizadores, como de transportadoras, encomendas, e artigos são subclasses desta classe genérica. Além disso, é importante realçar que, apesar de não ser explícito, visto que não são utilizadas interfaces para caracterizar *observers* e *observables*, é utilizado um *observer pattern* implicitamente em alguns dos catálogos. Por exemplo, em relação ao catálogo de artigos, o catálogo é o *observable*, possuindo o método *notifyArticles*, e os artigos do catálogo são os *observers*, possuindo um método de *update*, *updatePremiumPriceCorrection*, visto anteriormente. O catálogo em si não é a classe que muda de estado, mas sim o *clock* do programa, então poderia-se argumentar que este é o “*observable* fundamental do programa”, isto é, quando um utilizador decide avançar no tempo, o *clock* do programa é mudado, e o *controller* age como *middleman* para executar o método *notify* do catálogo.

Exemplo do método no catálogo de encomendas que notifica todas as encomendas acerca do avanço no tempo:

```
public void notifyAllOrders (LocalDateTime now) {
    for (Order order : this.catalog.values()) {
        order.send(now);
        order.arrived(now);
    }
}
```

É relevante acrescentar que o catálogo de utilizadores e de transportadoras também possui um `Map`, mas em vez de UUIDs, possui emails e nomes como chaves, respetivamente, de modo a que não haja dois utilizadores com o mesmo email ou duas transportadoras com o mesmo nome — tal não pode ser possível, como referido anteriormente sobre essas duas classes.

1.5.1 Estatísticas

Para além do que já foi mencionado, dois catálogos habilitam o processamento de estatísticas, o catálogo de utilizadores e de transportadoras.

Começando pelo catálogo de transportadoras, este possui um método que retorna a transportadora que ganhou mais dinheiro desde sempre:

```
public Carrier getRichestCarrier() {
    double highest = -1;
    Carrier chosen = null;
    for (Carrier carrier : this.catalog.values())
        if (carrier.getTotalMoneyEarned() > highest) {
            highest = carrier.getTotalMoneyEarned();
            chosen = carrier;
        }
    return chosen;
}
```

O mesmo é feito no catálogo de utilizadores, relativamente ao utilizador que mais gastou ou ganhou desde sempre. Porém, também é possível obter o utilizador que mais ganhou ou gastou num período de tempo (por isso é que a classe `User` possui os dois `Maps` `moneySpentByDay` e `moneyEarnedByDay`), e uma lista de utilizadores ordenada por quem ganhou ou gastou mais num dado período de tempo.

Foi utilizada a classe `Function` para reutilização de código, com uma expressão `lambda`:

```
public User getHighestEarnerPeriodOfTime (LocalDate infDate, LocalDate supDate) {
    return this.getHighestInPeriodOfTime(user ->
        user.getMoneyEarnedInPeriodOfTime(infDate, supDate));}

public User getHighestSpenderPeriodOfTime (LocalDate infDate, LocalDate supDate) {
    return this.getHighestInPeriodOfTime(user ->
        user.getMoneySpentInPeriodOfTime(infDate, supDate)); }
```

Em relação à listagem ordenada de utilizadores, foram utilizados *comparators*.

Como os dois *comparators* necessários têm características em comum, possuir duas datas, foi criado um *comparator* abstrato com duas subclasses:

```
public abstract class UserMoneyInPeriodOfTimeComparator implements Comparator<User> {
    protected LocalDate infDate;
    protected LocalDate supDate;

    protected UserMoneyInPeriodOfTimeComparator (LocalDate infDate, LocalDate supDate) {
        this.infDate = infDate;
        this.supDate = supDate;
    }
}
```

Na subclasse, é implementado necessariamente o método *compare* visto que esta não é abstrata:

```
@Override
public int compare (User u1, User u2) {
    double moneySpent1 = u1.getMoneySpentInPeriodOfTime(infDate, supDate);
    double moneySpent2 = u2.getMoneySpentInPeriodOfTime(infDate, supDate);
    return Double.compare(moneySpent2, moneySpent1);
}
```

Assim, é possível realizar a listagem ordenada através do seguinte método:
(Um método semelhante é criado em relação aos gastos).

```
public ArrayList<User> sortedHighestEarningInPeriodOfTime (LocalDate infDate, LocalDate
supDate) {
    ArrayList<User> users = new ArrayList<>(this.catalog.values());
    UserMoneyEarnedInPeriodOfTimeComparator comparator = new
UserMoneyEarnedInPeriodOfTimeComparator(infDate, supDate);
    Collections.sort(users, comparator);
    return users;
}
```

1.6 User Manager

A classe *UserManager* é responsável por gerir as *conexões* entre utilizadores e artigos. Para isso são utilizados diversos Maps de artigos vendidos, à venda, e comprados de um utilizador, juntamente com um Map das encomendas, e um para o carrinho.

```
private Map<UUID, ArrayList<Article>> soldArticles; // sellerID -> Articles
private Map<UUID, ArrayList<Article>> forSaleArticles; // sellerID -> Articles
private Map<UUID, ArrayList<Article>> boughtArticles; // buyerID -> Articles
private Map<UUID, ArrayList<Order>> userOrders; // buyerID -> Orders
private Map<UUID, Order> userCart; // buyerID -> order
```

Portanto, são necessários diversos métodos de adição e remoção, como *addArticleToMap*, *addArticleToCart* e *removeArticleFromCart*, e métodos que retornam informações acerca dos artigos relativos a um utilizador (array de *ArticleInfo*), como *getUserSoldArticlesInfo*, *getNotInCartForSaleArticlesInfo* (útil para *display* dos artigos disponíveis à venda) e *getArticlesInCartInfo* (para visualização dos artigos no carrinho).

Um método importante é o que é gere a compra de uma encomenda, ou seja, dos artigos que estão no carrinho. Para isso, é necessário remover os artigos do carrinho do utilizador que efetua a compra, remover os artigos do array de artigos à venda dos utilizadores que vendem os artigos, adicionar os artigos ao array de artigos comprados do utilizador, adicionar os artigos ao array de artigos vendidos dos utilizadores que os vendem, e, por fim, adicionar a encomenda ao array de encomendas do utilizador.

Também é relevante destacar mais dois métodos, os métodos que gerem o cancelamento e a devolução de uma encomenda. Para cancelar uma encomenda, é necessário remover a encomenda do array de encomendas, remover os artigos do array de artigos comprados do utilizador que cancela a encomenda, e dos arrays de artigos vendidos dos utilizadores que vendem os artigos, e por fim adicionar novamente os artigos aos arrays de artigos à venda dos utilizadores que vendem os respetivos artigos.

```

public void canceledOrder (Order order) {
    if (this.userOrders.containsKey(order.getBuyerId())) {
        ArrayList<Order> orders = this.userOrders.get(order.getBuyerId());
        Iterator<Order> it = orders.iterator();
        while (it.hasNext()) {
            Order el = it.next();
            if (el.equals(order)) {
                it.remove();
                order.removePackages();
            }
        }

        ArrayList<Article> articles = order.getArticles();
        for (Article article : articles) {
            if (this.boughtArticles.containsKey(order.getBuyerId())) {this.boughtArticles.get(order.getBuyerId()).remove(article);}
            if (this.soldArticles.containsKey(article.getSellerId())) {this.soldArticles.get(article.getSellerId()).remove(article);}
            this.addForSaleArticle(article.getSellerId(), article);
        }
    }
}

```

Para gerir a devolução de uma encomenda, é semelhante ao cancelamento, exceto que a encomenda não é removida do array de encomendas do utilizador, esta fica marcada como *returned*, e serve como o “histórico” de que essa encomenda foi enviada e depois devolvida.

1.7 Article Factory

Foi criada uma classe responsável por criar artigos, isto é, instanciar classes com superclasse `Article`. Como muita da complexidade da criação de artigos é tratada pelo *builder*, foi utilizado um *simple factory pattern*, em vez de por exemplo o *factory method pattern*, ou *abstract factory pattern*, para instanciar artigos, quer sejam T-Shirts, Malas ou Sapatilhas, e premium, usadas, premium e usadas ou não premium e não usadas. Essas possíveis combinações são diferenciadas através dos argumentos que os métodos recebem, por exemplo:

```

// used
public Article createTShirt(String description, String brand, double basePrice, int
numberOfOwners, double stateOfUse, Carrier carrier, UUID sellerID, TShirt.Size size,
TShirt.Pattern pattern) {
    return new TShirt.TShirtBuilder()
        .setUsedBehaviour(numberOfOwners, stateOfUse)
        .setPremiumBehaviour()
        // ...
        .buildWithoutPriceCorrection();
}

// none
public Article createTShirt(String description, String brand, double basePrice, Carrier
carrier, UUID sellerID, TShirt.Size size, TShirt.Pattern pattern) {
    return new TShirt.TShirtBuilder()
        .setUsedBehaviour()
        .setPremiumBehaviour()
        // ...
        .buildWithoutPriceCorrection();
}

```

1.8 App Clock

O programa tem de reagir ao avanço do tempo, porém este avanço no tempo não é o convencional, visto que é feito através de um relógio mutável, i.e., um relógio que guarda um instante específico no tempo, e o tempo apenas “avança” caso se escolha adicionar um valor a esse instante. Assim, o tempo para o programa não passa de forma equivalente à realidade, mas sim por escolha do utilizador, podendo, por exemplo, passar diretamente para n dias no futuro.

Para isso foi utilizada `AtomicReference` com o tipo `Instant`, simplesmente para que seja seguro atualizar a instância sem necessidade de sincronização explícita e que seja *thread-safe* em caso de atualização concorrente.

```
public final class AppClock extends Clock implements Serializable {

    private final AtomicReference<Instant> instantHolder;
    private final ZoneId zone;

    // ...
    public void add(TemporalAmount amountToAdd) {
        instantHolder.updateAndGet(current -> {
            ZonedDateTime currentZoned = current.atZone(zone);
            ZonedDateTime result = currentZoned.plus(amountToAdd);
            return result.toInstant();
        });
    }

    public int getYear() { return this.instantHolder.get().atZone(zone).getYear(); }
    public LocalDate getLocalDate() { return instantHolder.get().atZone(zone).toLocalDate(); }
    // ...
}
```

Relativamente a “`extends Clock`”:

Implementation Requirements:

This abstract class must be implemented with care to ensure other classes operate correctly. All implementations that can be instantiated must be final, immutable and thread-safe.

1.9 State

De modo a ser possível guardar o estado de um programa foi criada uma classe `State` responsável por salvar e recuperar o estado do programa. Como o que muda de estado para estado são os dados, o objeto que tem de ser salvo e recuperado é apenas o relativo ao `Model`. Sendo assim, não é necessário serializar ou desserializar objetos relativos às `Views` ou aos `Controllers`, mas sim todos os objetos relativos ao `Model` — assim, todos esses objetos têm de implementar a interface `Serializable`. No entanto, caso um utilizador guarde um estado, termine e inicie novamente o programa, ao recuperar o estado, terá de fazer login novamente.

```
public void saveData (AppModel appModel) throws FileNotFoundException, IOException {
    FileOutputStream fileOutputStream = new FileOutputStream(this.fileOut);
    ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
    objectOutputStream.writeObject(appModel);
    objectOutputStream.flush();
    objectOutputStream.close();
}
```

1.10 Model

A classe `AppModel` é uma classe que tem como objetivo ter acesso a todas as funcionalidades das classes abordadas anteriormente, fornecendo métodos que agregam uma ou mais dessas acessíveis funcionalidades (criando novos métodos de interação e de gestão), i.e., é uma classe que gere a “*back-end*” do programa, e que ao mesmo tempo simplifica a interação com os dados do programa e, portanto, oculta a complexidade, o que é similar ao *facade pattern* — torna a utilização das classes relativas ao Model (catálogos, `userManager`, etc.) mais fácil.

```
private final AppClock clock;
private final ArticleCatalog articleCatalog;
private final UserCatalog userCatalog;
private final CarrierCatalog carrierCatalog;
private final OrderCatalog orderCatalog;
private final UserManager userManager;
private double vintageMoneyEarned;
```

Assim, é necessário existir uma série de métodos: métodos que adicionam elementos aos catálogos, que adicionam e removem artigos dos repetitivos Maps do `userManager`, que verificam se utilizadores ou transportadoras já existem no sistema, que fornecem informações através de arrays de *records*, `ArticleInfo` e `OrderInfo`, que fornecem informações de dados estatísticos, que fazem *update* de artigos e de encomendas de acordo com o `AppClock`, que permitem comprar, cancelar ou devolver uma encomenda, e que permitem avançar no tempo. Exemplo de um dos métodos:

```
public void sellArticle (User seller, Article article) {
    article.updatePriceCorrection(this.clock.getYear());
    this.addArticleToCatalog(article);
    this.userManager.addForSaleArticle(seller, article);
}
```

Um dos métodos mais complexos desta classe é o método relativo à compra de uma encomenda. Depois de ser chamada a função do `userManager` para a compra da encomenda, é ainda necessário adicioná-la ao catálogo de encomendas, marcar a encomenda como finalizada e fazer a gestão de dinheiro gasto e ganho no total e de acordo com o dia. Para fazer isso, é preciso percorrer todos os artigos da encomenda e, por exemplo, adicionar o dinheiro ganho relativamente ao dia do relógio do programa aos utilizadores que vendem os artigos.

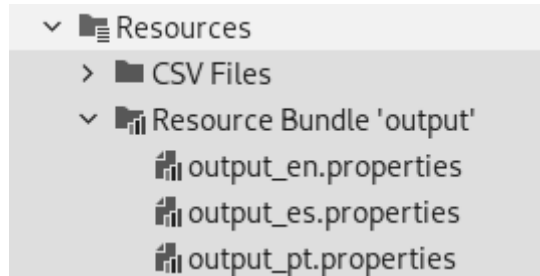
Por último, apesar de haver muitos outros métodos a demonstrar, achamos significativo explicar o funcionamento do método que permite avançar no tempo. Este método avança para um dia específico, e portanto precisa de fazer *update* das encomendas e dos artigos, chamando os métodos *notify* dos catálogos. Esta atualização é feita iterando pelos dias até chegar à data. Decidimos fazer o avanço dessa forma, pois torna a simulação mais realista, sendo o intervalo quantizado de avanço equivalente a um dia, em vez de ser feito um “salto” equivalente à duração inteira entre as duas datas.

```
public void timeMachine (LocalDate date) {
    while (this.clock.isDateAfter(date)) {
        int year = this.clock.getYear();
        this.clock.add(Duration.ofDays(1));
        if (this.clock.getYear() != year) this.updateArticleCatalog();
        this.updateOrders();
    }
}
```

1.11 Views

As Views são responsáveis por mostrar ao utilizador informações necessárias no ecrã, como diferentes opções, menus, listagens de artigos, encomendas, estatísticas, *prompts* de interação, etc.

De modo a que seja possível mudar de idioma de uma forma fácil e organizada, foram utilizados *Resource Bundles*:



Quando uma View é criada, esta selecciona o *bundle* de acordo com o *default* Locale: (relativo à instância da Java Virtual Machine)

```
public class AppView implements View {
    private ResourceBundle bundle;
    public AppView () {
        this.bundle = ResourceBundle.getBundle("output", Locale.getDefault());
    }
    public void updateLanguage() {
        this.bundle = ResourceBundle.getBundle("output");
    }
    // ...
}
```

Assim, quando o utilizador decide mudar o idioma do programa, o *controller* muda o *default* Locale (`Locale.setDefault(new Locale.Builder().setLanguage(str).build())`), e é chamado um método da View que atualiza o *bundle* que está a utilizar (o *bundle* é selecionado automaticamente de acordo com o novo *default*).

Para conseguir obter uma String de um *bundle* é necessário utilizar a *chave* relativa à String, e a mesma String em diferentes idiomas têm a mesma chave. Por este motivo, é bastante fácil mudar de idiomas.

Por exemplo, no *bundle* relativo ao idioma português uma linha seria formada desta forma:

```
order_limit_exceed_to_return=Tempo limite excedido para devolução da encomenda.
```

No *bundle* de idioma inglês:

```
order_limit_exceed_to_return=Time limit exceeded to return order.
```

No *bundle* de idioma espanhol:

```
order_limit_exceed_to_return=Se excedió el límite de tiempo para devolver el pedido.
```

Assim, a View, neste caso a *UIView*, pode obter essa String da seguinte maneira:

```
public void cannotReturnOrder() {this.bundle.getString("order_limit_exceed_to_return");}
```

Para além disso, as Views possuem métodos que fazem: *display* de menus cujas Strings também estão nos *bundles*, e estes métodos numeram as opções/Strings, para que o utilizador possa escolher facilmente; e *display* de listas de Strings recebidas do *controller*.

1.12 Controllers

Os *controllers* são o *middleman*, intermediário, visto que recebem as *requests* do utilizador através de métodos de leitura de input, e de acordo com essa leitura, interagem com o Model, caso seja necessário obter, criar, modificar ou apagar dados ou aplicar uma combinação dessas operações sobre um conjunto de dados, e interagem fornecendo instruções à View para que esta comunique com o utilizador.

Como os *controllers* tratam da leitura de input, estes possuem inevitavelmente uma forte densidade de lógica condicional, criando inúmeros caminhos de interação.

O `AppController` torna possível a interação e o acesso às seguintes funcionalidades:

- Registo;
- Login de utilizadores;
- Visualização de estatísticas;
 - Utilizador que ganhou mais (desde sempre ou num período de tempo);
 - Utilizador que gastou mais (desde sempre ou num período de tempo);
 - Transportadora que ganhou mais desde sempre;
 - Lista de utilizadores ordenada por maior ganho (desde sempre ou num período de tempo);
 - Total ganho pela Vintage;
- Salvar o estado do programa;
- Visualizar dados (todos os utilizadores, transportadoras, artigos ou encomendas);
- Mudar de idioma.

E relativamente ao `UserController`:

- Vender artigos;
- Visualizar artigos disponíveis para compra (possibilidade de adicionar ao carrinho);
- Ver carrinho (possibilidade de remover artigos do carrinho, ou de fazer encomenda);
- Ver artigos que o utilizador vendeu;
- Ver artigos que o utilizador tem à venda (possibilidade de remover artigos);
- Ver encomendas do utilizador (possibilidade de cancelar ou devolver encomenda);
- Avançar no tempo (para uma data específica no futuro).

Para além disso, na inicialização do programa existe a opção de ler ficheiros *csv* de utilizadores, transportadoras, artigos, e de “simulação”. Estes ficheiros são convertidos em dados do programa pela classe `Parser`.

O ficheiro de simulação contém uma série de instruções que automatizam o programa e que simulam a passagem do tempo. E existe, também, a opção de recuperar o estado do programa a partir de um ficheiro `.dat`. A criação da classe `Parser` foi mais simples/ rápida, visto que se “reutilizou” os métodos já existentes do Model, inicialmente necessários para o `AppController` e `UserController`.

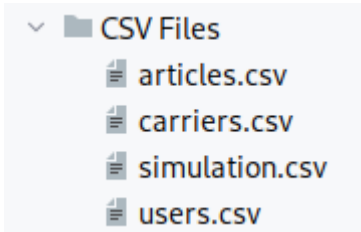
2 Funcionalidades

2.1 Menu inicial

```
-----VINTAGE-----  
1 - Read file  
2 - Load state  
0 - Continue
```

Como mencionado anteriormente, no menu inicial existe a opção de ler ficheiros. Estes ficheiros podem ser ficheiros de dados *csv*, i.e., informações de criação de utilizadores, transportadoras e artigos, ou ficheiros de simulação. Juntamente com a opção de recuperar o estado de um programa.

```
-----PARSER-----  
1 - Users  
2 - Carriers  
3 - Articles  
4 - Simulation  
0 - Exit
```



Os “comandos” que podem ser utilizados no ficheiro de simulação são os seguintes:

- CreateUser, CreateCarrier, SellArticle, AddToCart, BuyOrder, CancelOrder, ReturnOrder, ChangeUserValue, ChangeCarrierValue, e SaveState.

Exemplo de parte de um ficheiro de simulação:

```
25.05.2023,14:15,SellArticle,pedro@mail.com,CTT,SH0ES,Stylish and comfortable Puma shoes perfect for any occasion,Puma,65.99,41,with laces,white,2022,0.2  
26.05.2023,08:58,AddToCart,artur@mail.com,Stylish and comfortable Puma shoes perfect for any occasion  
26.05.2023,08:59,BuyOrder,artur@mail.com,100  
26.05.2023,09:15,AddToCart,artur@mail.com,The greatest shoes to ever exist  
26.05.2023,16:08,BuyOrder,artur@mail.com,30  
26.05.2023,16:50,CancelOrder,artur@mail.com,2
```

Primeiro, o utilizador Pedro, identificado a partir do seu email, coloca um artigo à venda.

No dia seguinte, o Artur decide comprar esse artigo, identificado a partir da sua descrição (pode e deve ser utilizado o ID do artigo, no entanto desta forma fica um ficheiro de exemplificação mais inteligível). Artur decide finalizar a compra e é feita a encomenda. Depois, decide adicionar um artigo ao carrinho agora vazio e finaliza outra compra. Umas horas depois, muda de ideias e decide cancelar a segunda encomenda que fez.

2.2 Menu principal

```
-----VINTAGE-----  
1 - Register  
2 - Login  
3 - Statistics  
4 - Save  
5 - View data  
6 - Change language  
0 - Exit
```

O menu principal serve em parte como uma visão de administrador do programa, visto que permite registar transportadoras, ver estatísticas, salvar o estado do programa, ver qualquer tipo de dados, etc.

Exemplo de registo de um utilizador e de uma transportadora:

```

-----REGISTER-----
1 - User
2 - Carrier

2
--> REGISTER-CARRIER
Name:
CTT
Base price for small-sized package (1 article):
1.50
Base price for a medium-sized package (2 - 5 articles):
5.65
Base price for a large-sized package (more than 5 articles):
9.99
Tax [0, 1]:
0.23
Profit [0, 1]:
0.30
Is premium? [y/n]
y
Additional price for premium articles:
2

-----REGISTER-----
1 - User
2 - Carrier

1
--> REGISTER-USER
Name:
Rodrigo
Email:
rodrigo@mail.com
Country:
Portugal
City:
Braga
Street:
Gualtar
Postal-code:
4610-200

```

Menu de estatísticas:

```

-----STATISTICS-----
1 - User who earned the most
2 - User who spent the most
3 - Carrier who earned the most
4 - Users sorted by highest earnings
5 - Users sorted by highest spending
6 - Total earned by vintage
0 - Exit

```

Tanto na opção 1 como na 2, utilizador que mais ganhou ou gastou, é possível escolher relativamente a um intervalo de datas ou desde sempre:

```

1
1 - Of all time
2 - Time period
0 - Exit

1
User {email = 'rodrigo@mail.com',

2
Start date (DD/MM/YYYY):
25/05/2023
End date (DD/MM/YYYY):
26/05/2023
User {email = 'artur@mail.com',

```

Para as opções 4 e 5 é necessário escolher um intervalo de datas:

```

4
Start date (DD/MM/YYYY):
11/05/2023
End date (DD/MM/YYYY):
04/06/2023

5
Start date (DD/MM/YYYY):
11/05/2023
End date (DD/MM/YYYY):
04/05/2023

1 - User {email = 'rodrigo@mail.com',
2 - User {email = 'artur@mail.com',
3 - User {email = 'pedro@mail.com',
4 - User {email = 'maria@mail.com',
5 - User {email = 'gramoso@mail.com',

1 - User {email = 'pedro@mail.com',
2 - User {email = 'artur@mail.com',
3 - User {email = 'gramoso@mail.com',
4 - User {email = 'maria@mail.com',
5 - User {email = 'rodrigo@mail.com'

```

Menu para visualizar dados: (útil para identificar/ verificar se alguns comportamentos do programa estão corretos ou não quando testados, por exemplo, se um artigo é criado, este tem de aparecer na visualização de todos os artigos, caso contrário existe um erro)

```
-----VIEW-DATA-----
1 - Users
2 - Carriers
3 - Articles
4 - Orders
0 - Exit
```

Menu de mudança de idioma: (muda o idioma de todos os menus, *prompts* e mensagens de erro)

```

2
-----VINTAGE-----
Select a language:
1 - English
2 - Portuguese
3 - Spanish
0 - Exit
1 - Registrar
2 - Login
3 - Estatísticas
4 - Salvar
5 - Visualizar dados
6 - Mudar idioma
0 - Sair

```

2.3 Modo de Utilizador

Opção de vender um artigo: (começa perguntando se é premium, usado, etc., de seguida qual é a transportadora, depois o tipo de artigo e por fim os detalhes relativos ao tipo de artigo escolhido)

```

Select one article behaviour:
1 - Used
2 - Premium
3 - Used and Premium
4 - None

4

-----LOGIN-----
Email:
rodrigo@mail.com

-----USER-MODE-----
1 - Sell article
2 - View articles
3 - View cart
4 - View my sold articles
5 - View my for sale articles
6 - View orders
7 - Use time machine
8 - Account details
0 - Back

Select the type of article:
1 - T-Shirt
2 - Bag
3 - Shoes

1

Description:
...
```

Visualização dos artigos disponíveis para compra (os artigos à venda relativos ao utilizador que está a visualizar não aparecem; o artigo é escolhido a partir de um index, caso se queira adicionar ao carrinho/cart) e do carrinho de compras/cart (opção de finalizar encomenda ou de remover artigo):

```

Articles in cart: (0 - Exit)

2
Add article to cart: (0 - Exit)
[1] : t-shirt
Description = 'Nice T-Shirt'
Brand = 'Nike'
Price = 39.99
Used behaviour = [not used]
Premium = [not premium]
Size = M
Pattern = SOLID

1
Article added to cart

[1] : t-shirt
Description = 'Nice T-Shirt'
Brand = 'Nike'
Price = 39.99
Used behaviour = [not used]
Premium = [not premium]
Size = M
Pattern = SOLID

1 - Remove from cart
2 - Make order
Order total price: 43.69$
2

```

Visualização de encomendas do utilizador:

```

[3] : (FINALIZED)
Packages:
{premium carrier = CTT, status = NOT_SENT, sentDate = 04-06-2023 15:01:00, arrivedDate = 06-06-2023 14:01:00, numberOfArticles = 1}
Price = 43.690000000000005
Creation date = 2023-06-03T12:01

Select an order: (0 - Exit)
3
1 - Cancel order
0 - Exit

```

Nesta situação, como a encomenda ainda não foi enviada, existe a opção de cancelar a encomenda, o que não será feito, para exemplificar como a passagem do tempo afeta a encomenda:

```

7 - Use time machine
8 - Account details
0 - Back

7
Enter the date (DD/MM/YYYY):
10/06/2023

[3] : (ARRIVED)
Packages:
{premium carrier = CTT, status = ARRIVED, sentDate = 04-06-2023 15:01:00, arrivedDate = 06-06-2023 14:01:00, numberOfArticles = 1}
Price = 43.690000000000005
Creation date = 2023-06-03T12:01

Select an order: (0 - Exit)
3
1 - Return order
0 - Exit

```

Aqui, o utilizador já não poderia devolver a encomenda, uma vez que já passaram mais de 48h desde a entrega.

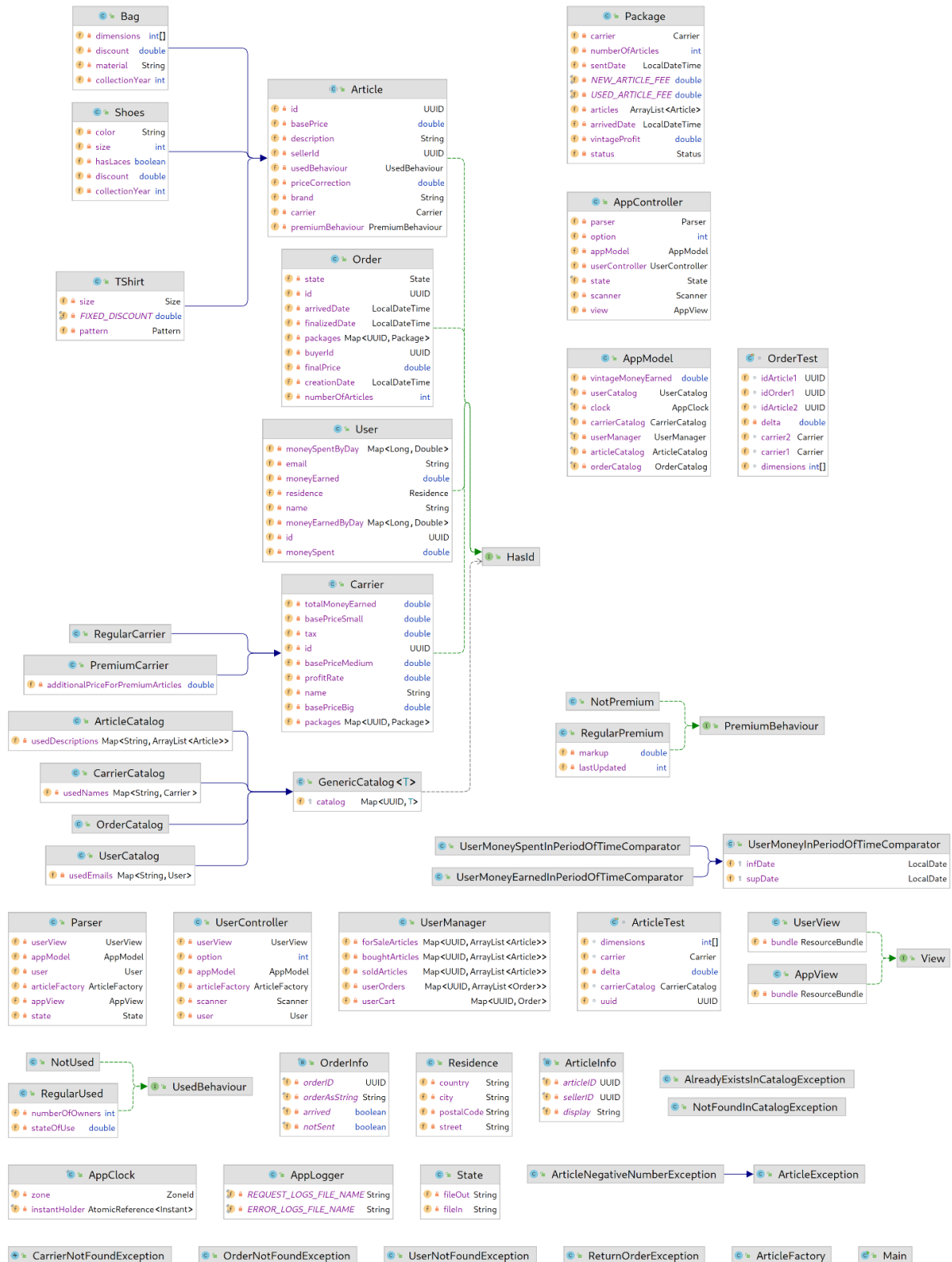
```

1
Time limit exceeded to return order.

```

3 Diagrama de classes

3.1 Fields



4 Conclusão

Para concluir, consideramos relevante destacar alguns aspetos que poderiam ser melhorados ou adicionados: melhor *error handling*, por exemplo, a construção de uma melhor hierarquia de *exceptions*; um sistema de paginação; um sistema de devolução de encomendas mais sofisticado/ realista, sendo possível devolver artigos em específico; e muitos outros, pois um projeto desta natureza permite inúmeras adições de funcionalidades. Não obstante, achamos que conseguimos concretizar todos os requisitos pedidos no enunciado do trabalho com sucesso, utilizando os conceitos de programação orientada a objetos estudados nas aulas teóricas e práticas da unidade curricular.