Projeto Prático - Vintage

Programação Orientada a Objetos

Daniel Pereira A100545
Mário Rodrigues A100109
Pedro Sousa A100823

Projeto desenvolvido para a Licenciatura em Engenharia Informática e para a Licenciatura em Ciências da Computação

Departamento de Informática
Universidade do Minho
maio 2023

Conteúdo

1 Introdução

O presente relatório é referente ao projeto prático, desenvolvido em Java, da Unidade Curricular de Programação Orientada a Objetos da Licenciatura em Engenharia Informática e Licenciatura de Ciências da Computação do Departamento da Informática da Escola de Engenharia da Universidade do Minho, para o ano de 2023.

O objetivo do projeto consistia em implementar uma aplicação que permitisse simular uma loja com produtos em 1ª e 2ª mão, encomendas, transportadoras e usuários. Entre os principais requisitos do mesmo estava a possibilidade de carregamento de dados a partir de ficheiros, incluindo automatização da simulação.

O projeto apresentado foi realizado pelo grupo número 1, constituído por Daniel Pereira (A100545), Mário Rodrigues (A100109), e Pedro Sousa (A100).

Serão enunciadas as principais decisões tomadas pelo grupo na conceção do trabalho, analisando as suas consequências e eventuais vantagens e desvantagens; assim como será explicada em grande detalhe a arquitetura da aplicação desenvolvida.

2 Aplicação Desenvolvida

2.1 Interface

A aplicação desenvolvida tem uma interface pelo terminal. A interação com esta é feita através de comandos.

Figura 1: Interface Login

Os dados são mostrados em formato de tabela quando mais do que um objeto é apresentado, ou uma lista de atributos para um objeto singular, como mostrado nas figuras a baixo.

2.2 Funcionalidades

Todas as funcionalidades requeridas para a nota máxima de 20 valores foram implementadas. Assim, e para as listar, o programa desenvolvido suporta

- Inserção de produtos / atualização de produtos / remoção de produtos / visualizar os seus produtos
- Avançar / recuar no tempo, emitir e consultar recibos
- Responder a estatísticas relativamente ao estado da simulação
- Guardar / carregar a simulação a partir de ficheiros, bem como carregar ficheiro de automatização da simulação

As funcionalidades de estatísticas podem ser feitas utilizando os seguintes comandos:

- *Biggest Sellers By Products* - obtêm o top de vendedores por produtos.
- *Biggest Seller By Products* - obtêm o maior vendedor pro produtos.
- *Get Seller Orders* - obtêm as encomendas de um vendedor.
- *Biggest Buyers By Products* - obtêm o top de compradores por produtos.
- *Biggest Sellers By Money* - obtêm o top de vendedores por dinheiro.

A forma como estas foram implementadas é descrita na secção 3. A seguir mostram-se exemplos da utilização da aplicação.

Figura 2: Vista inicial do programa como utilizador

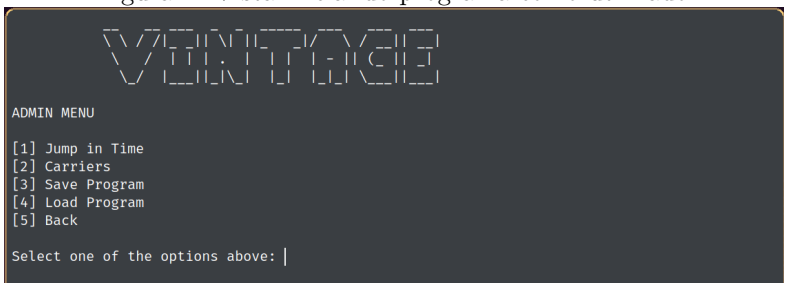


Figura 3: Vista inicial do programa como administrador

3 Arquitetura

3.1 Model-View-Controller

A decisão mais impactante no projeto foi a escolha da utilização da arquitetura *Model-View-Controller*, de forma a separar a camada lógica da aplicação da camada de apresentação. A cada uma destas camadas foi atribuído um *package* único.

Dada a maior complexidade do diagrama de classes devido ao maior número destas proveniente da arquitetura MVC, apresentam-se em anexo 2 diagramas distintos: 1 corresponde ao diagrama do *núcleo* do trabalho, e outro corresponde ao diagrama do módulo usado para processamento de anotações (algo que será explicado mais adiante).

3.2 Gradle

Para facilitar o desenvolvimento do projeto, escolheu utilizar a ferramenta Gradle para auxiliar na compilação e gestão das dependências da aplicação. Para compilar e rodar o projeto utilizam-se os comandos (a partir da diretoria base do projeto):

```
cd Vintage
./gradlew build
java -cp build/classes/java/main vintage.Main
```

4 Modelo

4.1 Produtos

A classe "Item"é a classe base para todos os outros tipos de produto. Ela contém atributos básicos, como descrição, marca, preço e avaliação. Além disso, possui referências a um usuário (proprietário) e um transportador (carrier), que são instâncias da classe "User"e "Carrier", respectivamente. O atributo "owners"indica quantas vezes o produto já teve proprietários anteriores.

A classe "Bag"representa um tipo específico de produto que é uma bolsa. Ela herda todos os atributos da classe "Item", mas adiciona atributos específicos para bolsas, como tamanho, material e ano de lançamento.

A classe "Shoes"representa outro tipo específico de produto, neste caso um par de sapatos. Ela também herda todos os atributos da classe "Item", mas adiciona atributos específicos para sapatos, como tamanho, cor, se tem cadarços ou não e ano de lançamento.

Por fim, a classe "TShirt"representa um tipo de produto diferente, uma camiseta. Ela também herda todos os atributos da classe "Item", mas adiciona atributos específicos para camisetas, como tamanho e padrão.

4.2 Transportadores

A classe "Carrier"em Java é uma entidade que representa uma transportadora. Em um sistema de vendas, as transportadoras são importantes porque são responsáveis pela entrega dos produtos aos clientes.

A classe "Carrier"possui atributos como nome, lucro, tempo de entrega e premium, que podem ser usados para determinar qual transportadora é a mais adequada para cada tipo de envio. O atributo "premium"indica se a transportadora transporta produtos do tipo Premium ou não.

4.3 Encomendas

A classe "Order"em Java é uma entidade que representa uma encomenda em um sistema de gerenciamento de vendas. Em um sistema de e-commerce, por exemplo, as encomendas são uma parte fundamental do processo de compra e venda, pois permitem que os clientes solicitem produtos e que as empresas os entreguem.

Ao criar uma instância da classe "Order", é necessário fornecer informações como o comprador, os itens que compõem a encomenda, o tamanho da encomenda, o estado atual da encomenda, o preço total e o endereço de entrega.

A classe "Order"possui um atributo "items", que é um mapa que associa cada item da encomenda com seu estado atual. Essa associação é importante porque permite que a empresa de envio saiba qual é o estado atual de cada item da encomenda e, assim, possa monitorar a entrega de forma eficiente.

Outro atributo importante da classe "Order"é o "size", que é calculado automaticamente com base no número de itens na encomenda.

Além disso, a classe "Order"também possui um atributo "state", que representa o estado atual da encomenda. Esse estado pode ser atualizado conforme a encomenda é pendente, terminada e entregue.

4.3.1 Cálculo de Encomendas

Na classe "Order"apresentada, o método "calculatePrice()"é responsável por calcular o preço total da encomenda com base nos itens incluídos nela. O cálculo é feito somando os preços de cada item e adicionando uma taxa adicional com base na condição do item.

4.4 Recibos

Na classe Java apresentada, existem duas classes de recibos: o "BuyerReceipt", que é um recibo emitido para o comprador da encomenda, e o "SellerReceipt", que é um recibo emitido para o vendedor de cada produto. Ambas as classes contêm informações como o ID da encomenda, o preço total, os itens incluídos e a data de emissão.

4.5 Modelo

Decidimos adotar o modelo MVC (Model-View-Controller) é um padrão de arquitetura de software que separa as preocupações em uma aplicação em três componentes principais: modelo, visualização e controlador.

O modelo é responsável por gerenciar os dados e a lógica de negócios da aplicação. Ele é independente da interface do usuário e deve ser projetado de forma a ser facilmente reutilizado em outras partes da aplicação.

A visualização é responsável por exibir as informações ao usuário de uma forma amigável. Ela deve ser construída de forma independente do modelo, de forma que possa ser facilmente adaptada a diferentes tipos de dispositivos ou interfaces.

O controlador é responsável por receber as entradas do usuário, interagir com o modelo e atualizar a visualização. Ele é o intermediário entre a visualização e o modelo, garantindo que as ações do usuário sejam refletidas corretamente no estado da aplicação.

O uso do modelo MVC promove a modularidade e a reutilização do código, tornando a aplicação mais fácil de desenvolver, manter e testar.

4.6 Ficheiros de estado

A interface Serializable indica que a classe pode ser serializada, ou seja, pode ser convertida em uma sequência de bytes para ser armazenada ou transmitida. Para que isso aconteça, é necessário que a classe implemente a interface Serializable e seus atributos também sejam serializáveis.

Na loja, por exemplo, todas as classes relevantes implementam a interface Serializable, pois precisam ser salva em um arquivo para ser recuperada posteriormente. Ao salvar ,por exemplo, uma Order em um arquivo, o Java converte todos os seus atributos em bytes, o que permite que a ordem seja recuperada posteriormente, com todos os seus dados intactos.

Em resumo, a interface Serializable é importante para a persistência de dados em Java, permitindo que objetos sejam convertidos em bytes para serem armazenados ou transmitidos. Na loja, ela é amplamente utilizada para garantir a consistência dos dados em casos de falhas do sistema ou desligamentos inesperados.

5 Vistas

Para o projeto foram desenvolvidas várias vistas, uma por cada tipo de dados a mostrar no ecrã .

As vistas correspondem normalmente a 4 métodos: 2 deles genéricos e aplicáveis a todas, que correspondem a métodos que mostram ao utilizador que houve um erro ao processar um comando, e mostrar um aviso (por exemplo, tratar uma lâmpada como sendo uma câmara). Os restantes dois métodos imprimem para o ecrã o objeto que lhes é específico (uma casa, uma lâmpada,

.etc) e imprimem uma lista (para haver a noção de ordem de impressão) de objetos.

Para facilitar a impressão de dados em forma de tabela, criou-se a classe TablePrinter, que possui um método com número variável de argumentos que imprime uma tabela para o ecrã a partir das suas colunas, que é utilizada em praticamente todas as vistas. Também possui um método auxiliar que converte listas em listas de objetos, de forma a poder generalizar os tipos de dados que podem ser impressos no ecrã.

6 Controladores

Os controladores desenvolvidos serviram para ligar ambas as interfaces desenvolvidos ao modelo lógico e aplicacional. Foram desenvolvidos vários controladores distintos, um para cada entidade relevante. Assim sendo, os controladores expõem a funcionalidade da aplicação às vistas.

A forma como os controladores foram usados e ligados uns aos outros é detalhada de seguida.

7 Testes unitários

De forma a assegurar o correto funcionamento do projeto, com ênfase no modelo da aplicação, foram desenvolvidos testes unitários em paralelo com o desenvolvido do código do programa. O objetivo destes testes é, por um lado, ajudar a garantir a correção do programa, motivo pelo qual o grupo tentou garantir que a cobertura dos testes fosse praticamente total, objetivo que foi cumprido no *package* do modelo. Por outro lado, também foram criados testes unitários para avaliar se as decisões arquiteturais relativas a composição/agregação de classes foram respeitadas.

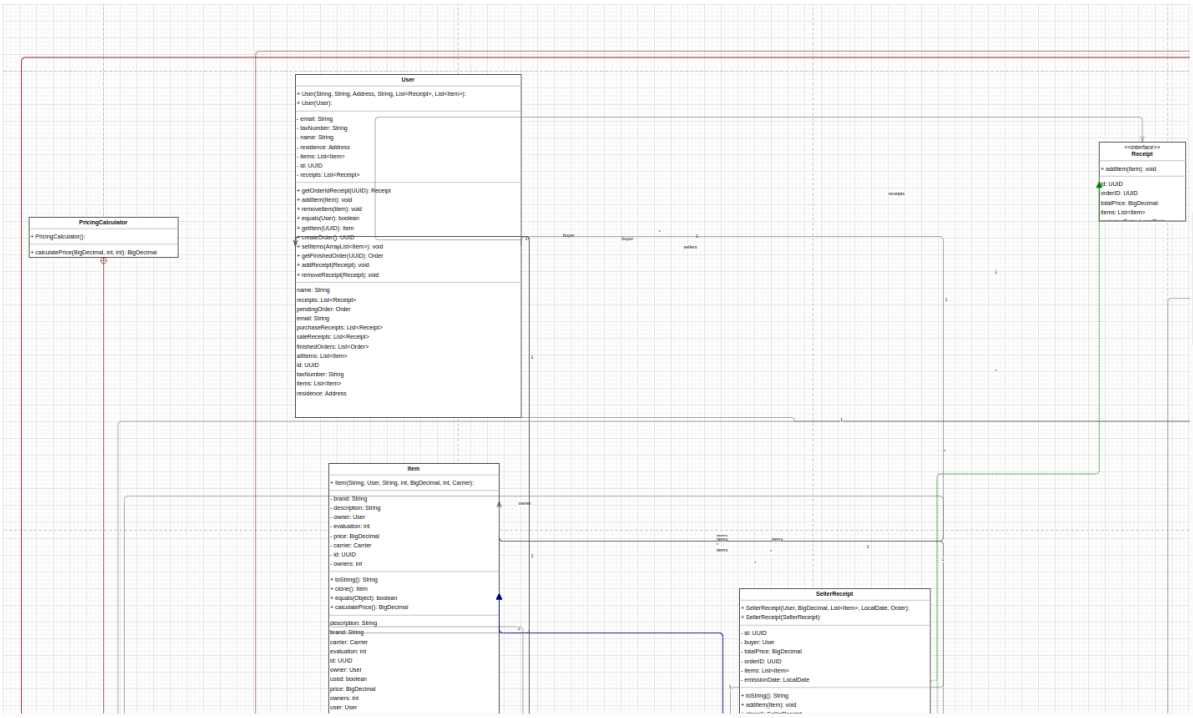
Tal como abordado nas aulas práticas, utilizou-se a biblioteca JUnit para o desenvolvimento dos testes. Estes encontram-se na diretoria *test/*.

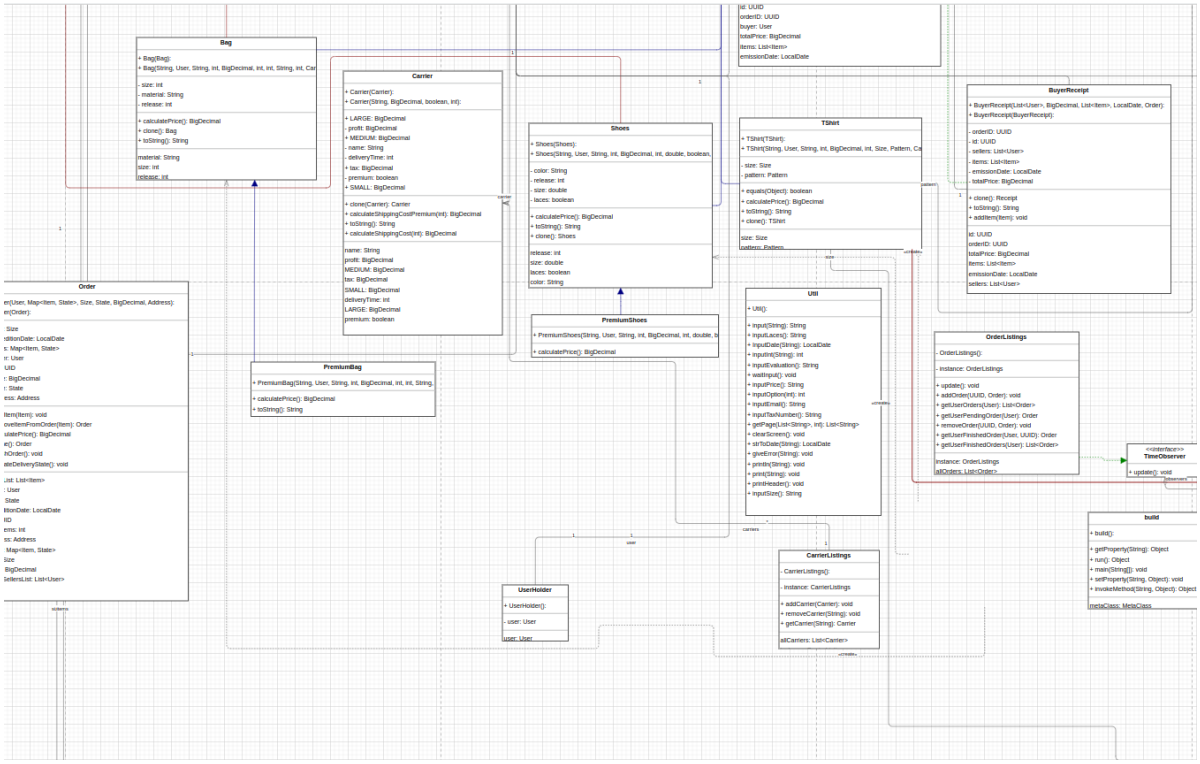
8 Conclusão

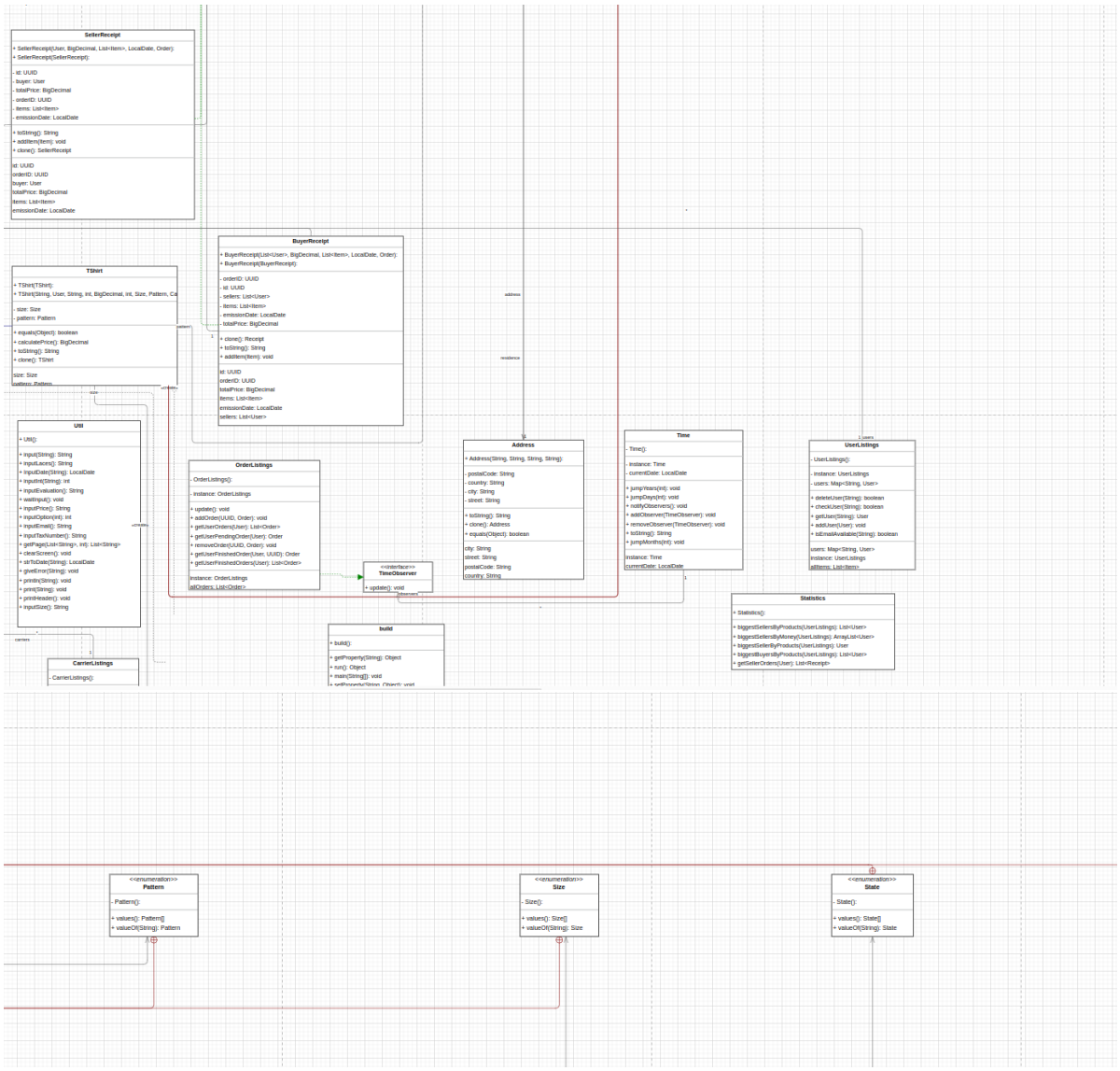
Para concluir, o projeto foi bem sucedido, todos os elementos do grupo tiveram uma atitude bastante positiva para com este e perante os normais problemas que surgem em engenharia de *software*, o que é visível na qualidade da aplicação desenvolvida.

9 Anexos

9.1 Diagrama de classes




Bag
+ Bag(Bag):
+ Bag(String, User, String, int, BigDecimal, int, int, String, int, Car
- size: int
- material: String
- release: int
+ calculatePrice(): BigDecimal
+ clone(): Bag
+ toString(): String
material: String
size: int
release: int
Carrier
+ Carrier(Carrier):
+ Carrier(String, BigDecimal, boolean, int):
+ LARGE: BigDecimal
- profit: BigDecimal
+ MEDIUM: BigDecimal
- name: String
- deliveryTime: int
+ tax: BigDecimal
- premium: boolean
+ SMALL: BigDecimal
+ clone(Carrier): Carrier
+ calculateShippingCostPremium(int): BigDecimal
+ toString(): String
+ calculateShippingCost(int): BigDecimal
name: String
profit: BigDecimal
MEDIUM: BigDecimal
tax: BigDecimal
SMALL: BigDecimal
deliveryTime: int
LARGE: BigDecimal
premium: boolean
id: UUID
orderID: UUID
buyer: User
totalPrice: BigDecimal
items: List<Item>
emissionDate: LocalDate
Shoes
+ Shoes(Shoes):
+ Shoes(String, User, String, int, BigDecimal, int, double, boolean,
- color: String
- release: int
- size: double
- laces: boolean
+ calculatePrice(): BigDecimal
+ toString(): String
+ clone(): Shoes
release: int
size: double
laces: boolean
color: String
TShirt
+ TShirt(TShirt):
+ TShirt(String, User, String, int, BigDecimal, int, Size, Pattern, Ca
- size: Size
- pattern: Pattern
+ equals(Object): boolean
+ calculatePrice(): BigDecimal
+ toString(): String
+ clone(): TShirt
size: Size
pattern: Pattern
BuyerReceipt
+ BuyerReceipt(List<User>, BigDecimal, List<Item>, LocalDate, Order):
+ BuyerReceipt(BuyerReceipt):
- orderID: UUID
- id: UUID
- sellers: List<User>
- items: List<Item>
- emissionDate: LocalDate
- totalPrice: BigDecimal
+ clone(): Receipt
+ toString(): String
+ addItem(Item): void
id: UUID
orderID: UUID
totalPrice: BigDecimal
items: List<Item>
emissionDate: LocalDate
sellers: List<User>
Order
er(User, Map<Item, State>, Size, State, BigDecimal, Address):
er(Order):
.Size
stItemDate: LocalDate
s: Map<Item, State>
r: User
UID
r: BigDecimal
r: State
ess: Address
Item(Item): void
ownItemFromOrder(Item): Order
alatePrice(): BigDecimal
er(): Order
nOrder(): void
ateDeliveryState(): void
ist: List<Item>
: User
State
ItemDate: LocalDate
ID
ems: int
ss: Address
Map<Item, State>
Size
BigDecimal
SellersList: List<User>
PremiumBag
+ PremiumBag(String, User, String, int, BigDecimal, int, int, String,
+ calculatePrice(): BigDecimal
+ toString(): String
PremiumShoes
+ PremiumShoes(String, User, String, int, BigDecimal, int, double, b
+ calculatePrice(): BigDecimal
Util
+ Util():
+ input(String): String
+ inputLaces(): String
+ inputDate(String): LocalDate
+ inputInt(String): int
+ inputEvaluation(): String
+ waitInput(): void
+ inputPrice(): String
+ inputOption(int): int
+ inputEmail(): String
+ inputTaxNumber(): String
+ getPage(List<String>, int): List<String>
+ clearScreen(): void
+ strToDate(String): LocalDate
+ giveError(String): void
+ println(String): void
+ print(String): void
+ printHeader(): void
+ inputSize(): String
OrderListings
- OrderListings():
- instance: OrderListings
+ update(): void
+ addOrder(UUID, Order): void
+ getUserOrders(User): List<Order>
+ getUserPendingOrder(User): Order
+ removeOrder(UUID, Order): void
+ getUserFinishedOrder(User, UUID): Order
+ getUserFinishedOrders(User): List<Order>
instance: OrderListings
allOrders: List<Order>
<<interface>>
TimeObserver
+ update(): void
build
+ build():
+ getProperty(String): Object
+ run(): Object
+ main(String[]): void
+ setProperty(String, Object): void
+ invokeMethod(String, Object): Object
metaClass: MetaClass
UserHolder
+ UserHolder():
- user: User
user: User
CarrierListings
- CarrierListings():
- instance: CarrierListings
+ addCarrier(Carrier): void
+ removeCarrier(String): void
+ getCarrier(String): Carrier
allCarriers: List<Carrier>
1
*



* Caso não seja percetível o diagram de classes, dentro da pasta zip estará um ficheiro com a extensão draw.io que é o diagrama de classes.