Universidade do Minho

MiEI

UMCarroJá

Grupo 9

86268 - Maria Pires

85242 - Maria Regueiras

25 de maio de 2019

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este relatório contém a apresentação do projeto desenvolvido ao longo do semestre para a disciplina de *Programação Orientada a Objetos*. Serão expostas as soluções implementadas para a construção da plataforma *UMCarroJá!*.

O trabalho prático tem como objetivo a criação de uma plataforma em Java que permite o aluguer de veículos particulares, para tal desenvolvemos todas as funcionalidades, desde a criação de utilizadores e veículos e aluguer de viaturas à criação de uma viajem no carro alugado e imputação do preço.

Para cumprir as metas do projeto foram desenvolvidas as funcionalidades básicas da aplicação requeridas no enunciado que permitem:

Aos Clientes: solicitar o aluguer de um veículo de acordo com vários parâmetros tais como: proximidade da viatura, o preço, a distância que os clientes terão que percorrer a pé até ao veículo, uma marca específica, autonomia.

Aos Proprietários: adicionar veículos e efetuar ações sobre estes tais como: o seu abastecimento, alteração de preços, permitir que seja alugado por um determinado cliente e alterar a disponibilidade dos seus veículos.

Finalmente, a aplicação permite guardar registo de todas as operações efetuadas e possui mecanismos para aceder a essa informação.

Capítulo 2

Modularização

O projeto foi iniciado pela criação das classes mais básicas, as duas superclasses USER e Vehicle e as suas subclasses. Ao longo do desenvolvimento do projeto foram adicionadas mais classes e as que foram implementadas inicialmente sofreram várias alterações, como seria de esperar.

Apresenta-se de seguida o diagrama do projeto:

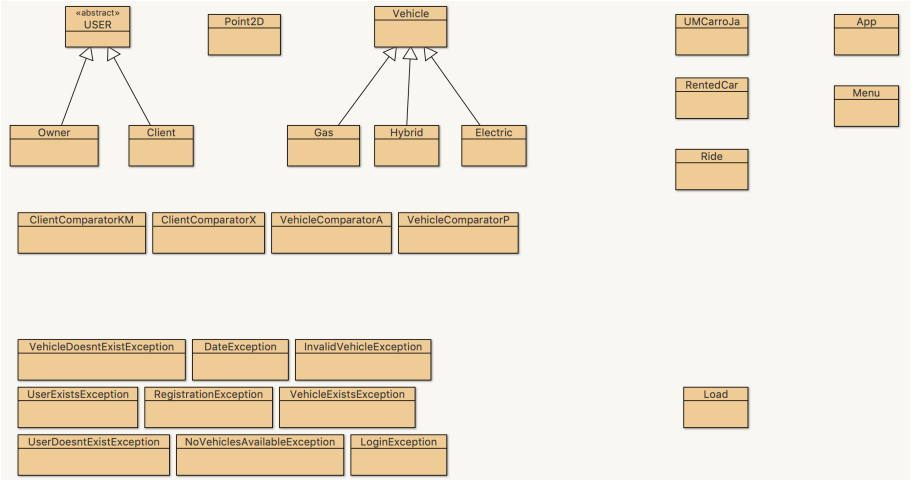

Figura 2.1: Modularização

2.1 Classe User

A classe *USER* permite-nos criar a identificação de um utilizador. Aqui criamos todas as variáveis necessárias para que quer o cliente, quer o proprietário sejam identificados. Para tal, temos nesta classe definidos o email do utilizador e o seu nif, que são essenciais para que o utilizador possa fazer login na aplicação. Possui ainda o nome, a morada e a classificação atribuída ao cliente.

Uma vez que tanto os clientes como os proprietários necessitavam de ser especializados criamos duas classes a *Client* e a *Owner* por herança, isto é, que

são subclasses da classe *USER*. Como na aplicação apenas criamos clientes e proprietários, a classe *Utilizador* passou a ser abstrata, visto que não criávamos nenhum objeto a partir dos seus construtores, mas era necessário manter a hierarquia da herança.

```
public abstract class User implements Serializable
{
    private String email;
    private String name;
    private String address;
    private int nif;
    private int rating;
}
```

Listing 2.1: Classe User

2.2 Classe Client

A classe *Client* é uma subclasse de *USER*, logo herda todas as informações desta. No entanto, para além de ter uma identificação, também necessita de ter um histórico de viagens do cliente, a sua localização e o que está disposto a caminhar até ao veiculo.

```
public class Client extends Entidade implements Serializable
{
   private double walk;  //4km por hora
   private double x;
   private double y;
   Point2D location = new Point2D(x,y);
   private Set<RentedCar> rentingHistory;
}
```

Listing 2.2: Classe Client

2.3 Classe Owner

A classe *Owner*, também subclasse de *USER*, especifica-se através da implementação de um histórico de alugueres dos seus veículos e uma lista de todas as viaturas que ele possui.

```
public class Onwer extends Entidade implements Serializable
{
    private Set<RentedCar> rentingHistory;
    private Map<String,Vehicle> vh;
}
```

Listing 2.3: Classe Onwer

2.4 Classe Point2D

Nesta classe definimos a posição **xOy** no espaço de um determinado utilizador ou veículo, o que nos permite aceder à sua localização para poder,por exemplo, calcular qual o veículo mais próximo do cliente e a distancia entre o fim e o inicio de uma viagem.

```java
public class Point2D implements Serializable
{
    private double x;
    private double y;
}
```

Listing 2.4: Point2D

2.5 Classe Vehicle

A classe *Vehicle*, essencialmente, permite-nos criar um veículo. O veículo possui um tipo de combustível *Gas, Electric, Hybrid*, uma marca e a sua matricula, que é única. As suas características incluem também a velocidade média, o consumo, a autonomia a sua localização. Foi definido um valor fixo para a variável *deposit* (deposito de combustível), visto ser necessária para os cálculos entre a autonomia e o consumo de cada veículo. Em ultimo, foi adicionado um histórico de alugueres do carro.

```java
public abstract class CAR implements Serializable
{
    private static double deposit = 100;
    private String type, brand, plate;
    private double speed, price, comsuption, autonomy, x, y;
    Point2D location = new Point2D(x,y);
    private boolean available;
    private int rating, nif;
    private Set<Ride> rentingHistory;
}
```

Listing 2.5: Classe CAR

2.5.1 Classes Gas, Electric, Hybrid

As classes *Gas*, *Electric* e *Hybrid* são subclasses de *Vehicle* cujo proprosito é especializar as viaturas tendo em conta o seu tipo de combustível.

2.6 Classe RentedCar

Os históricos dos clientes e dos proprietários são distintos dos históricos dos veículos em termos de informação. A classe *RentedCar* define as caraterísticas de um aluguer realizado por um cliente.

Decidimos que seria essencial guardar os emails do cliente e do proprietário e o carro que foi alugado. A localização da partida e do destino é também necessária para poder obter o preço de modo a calcular os quilómetros percorridos, que também são guardados. No final da viagem é calculado o preço e a sua duração que são também registados no histórico.

```java
public class Renting implements Comparable<RentedCar>, Serializable
{
    private String ownerEmail, clientEmail;
    private Vehicle car;
    private double price;
    private Point2D start;
    private Point2D destination;
```

```
8       private double autonomy;     //Autonomia usada para viagem
9       private double time;        //Tempo de duracao da viagem
10      private LocalDateTime date;
11      private double kms;
12  }
```

Listing 2.6: Classe RentedCar

2.7 Class Ride

A classe *Ride* define as caraterísticas de uma viagem efetuada por um veículo num aluguer.

```
1  public class Ride implements Comparable<Ride>, Serializable
2  {
3      private String clemail;
4      private LocalDateTime date;
5      private Point2D start;
6      private Point2D destination;
7      private double kms;
8      private double time;            //Tempo de duracao da viagem
9      private double realPrice;
10 }
```

Listing 2.7: Classe UMCarroJá

2.8 Classe UMCarroJá

Com todas as classes que necessitávamos como base para o desenvolvimento da aplicação implementadas, criamos os métodos necessários na *UMCarroJa* para responder assim ao pretendido no enunciado.

Na *UMCarroJá* é possível criar um novo registo para utilizadores, tanto clientes como proprietários. Tal como podemos adicionar também podemos remover, e validar o acesso de um utilizador. Além dessas funcionalidades, pode-se proceder à realização de um novo aluguer, com a opção de escolher o carro pretendido segundo os parâmetros especificados no enunciado. Tendo sido o aluguer terminado, é necessário calcular o tempo, e o custo real e estimado associado a esta, tendo em conta os fatores de aleatoriedade. É possível também aceder-se ao histórico das viagens de um utilizador, ao total faturado por cada viaturae ao top dez clientes que mais utilizou a app ou percorreu mais quilómetros.

```
1  public class UMCarroJ  implements Serializable
2  {
3     private Map<String , Client> clients;
4     private Map<String , Owner> owners;
5     private Map<String , Set<Vehicle>> vehicles;
6  }
```

Listing 2.8: Classe UMCarroJá

2.9 Classe App

Nesta classe, e com auxilio da classe Menu, é implementada a nossa aplicação. O grupo decidiu adicionar os seguintes menus de modo a satisfazer as condições

do enunciado.

- Menu Principal
- Menu Cliente
- Sub-menu Cliente
- Menu Proprietário

Esta classe possui todos os métodos que garantem que o programa seja bem executado e que sejam satisfeitas todas as condições ao longo do código e caso algo falhe permite avisar o utilizador.

```
public class App implements Serializable
{
    private static Menu menu;
    private UMCarroJa umcj;
}
```

Listing 2.9: Classe App

2.10 Classes de Ordenação

Foram implementadas 4 classes de ordenação:

- **ClientComparatorKM:** Comparator do tipo *Client* para a criação de um TreeSet de clientes ordenado crescentemente pelo numero de quilómetros percorridos.

```
public class ClientComparatorKM implements Comparator<Client>

```

- **ClientComparatorX:** Comparator do tipo *Client* para a criação de um TreeSet de clientes ordenado crescentemente pelo numero vezes que usaram a aplicação.

```
public class ClientComparatorX implements Comparator<Client>

```

- **VehicleComparatorA:** Comparator do tipo *Vehicle* para a criação de um TreeSet de veículos ordenados crescentemente pela sua autonomia.

```
public class VehicleComparatorA implements Comparator<Vehicle>

```

- **VehicleComparatorP:** Comparator do tipo *Vehicle* para a criação de um TreeSet de veículos ordenados crescentemente pelo seu preço.

```
public class VehicleComparatorP implements Comparator<Vehicle>

```

2.11 Exceptions

As *exceptions* são fundamentais na execução de um programa, como por exemplo, quando é requisitado um veículo que não existe, ou são inseridos dados inválidos. Foram implementadas as seguintes *exceptions*:

- VehicleDoesntExistException que devolve uma mensagem caso o veículo especifico procurado pelo cliente não exista;
- VehicleExistsException que devolve uma mensagem caso o veículo a adicionar já exista na aplicação;
- InvalidVehicleException para quando se tenta adicionar um novo veículo que não seja do tipo *Gas*, *Electric* ou *Hybrid*;
- NoCarsAvailableException que devolve uma mensagem caso ao procurar por um veículo, nenhum veículo esteja disponível de momento;
- UserDoesntExistException que devolve uma mensagem caso o utilizador procurado não exista;
- UserExistsException que devolve uma mensagem caso o utilizador que se quer adicionar já exista na aplicação;
- LoginException para quando o utilizador insere os dados errados ao tentar entrar na sua conta;
- RegistrationException que ocorre caso ao criar uma conta, o email já exista na base de dados da aplicação;
- DateException para quando o formato da data esteja incorreto. Este erro poderá acontecer ao pedir uma nova viagem, ao registar uma viagem, ou ao solicitar o total faturado e o registo de viagens entre datas.

Capítulo 3

Funcionamento da UMCarroJá!

3.1 Menu Principal

Quando a aplicação é iniciada é possível fazer um registo como proprietário, caso pretenda disponibilizar os seus veículos para aluguer, ou caso deseje alugar algum carro existente, então poderá fazer o registe como cliente. O utilizador da aplicação tem que selecionar o número correspondente à opção que pretende e inserir os dados que são requisitados. Caso o utilizador já tenha conta apenas tem de selecionar o número correspondente ao login e inserir os seus dados de acesso.

```
********** Menu **********

1 - Registar Cliente
2 - Registar Proprietário
3 - Login
0 - Sair

**************************
```

Figura 3.1: Menu Principal da aplicação

3.2 Menu de Clientes

Ao fazer o login como cliente o utilizador é direcionado para o menu de clientes onde tem a possibilidade de visualizar o seu perfil, alugar um veículo, obter o seu histórico de alugueres, consultar o *top 10 clientes* em função dos quilómetros percorridos e do uso da aplicação. Por ultimo, tal como há a possibilidade de criar também existe a possibilidade de eliminar o perfil.

Caso o cliente escolha a primeira opção, alugar um veículo, é redirecionado para o sub-menu da área de cliente onde pode escolher as definições do aluguer a realizar, tal como consta na figura 3.3.

```
*********** Menu ***********

1 - Ver Perfil
2 - Alugar um veículo
3 - Histórico de Alugueres
4 - Top 10 Clientes -> km
5 - Top 10 Clientes -> Uso
6 - Eliminar Perfil
0 - Sair

****************************
```

Figura 3.2: Menu de Clientes

```
*********** Menu ***********

1 - Escolher o veículo mais próximo
2 - Escolher um veículo específico
3 - Escolher o veículo mais barato
4 - Escolher o veículo mais barato + caminhada
5 - Escolher um veículo com a autonomia desejada
0 - Sair

****************************
```

Figura 3.3: Menu de aluguer

3.3 Menu de Proprietários

Ao fazer o login de um proprietário o utilizador é direcionado para o seu menu, onde tem a possibilidade de registar um novo veículo, obter o seu histórico de alugueres, a lista de todos os seus veículos, o lucro que obteve de cada viatura e consultar o *top 10 clientes* em função dos quilómetros percorridos e do uso da aplicação. Por ultimo, tal como há a possibilidade de criar também existe a possibilidade de eliminar o perfil.

```
*********** Menu ***********

1 - Ver Perfil
2 - Adicionar um veículo novo
3 - Lista dos meus veículos
4 - Top 10 Clientes -> km
5 - Top 10 Clientes -> Uso
6 - Lucro dos meus Carros
7 - Eliminar Perfil
0 - Sair

****************************
```

Figura 3.4: Menu de Proprietários

Capítulo 4

Conclusão

Em suma, o grupo considera que o projeto foi concluído com sucesso. permitiu-nos compreender como a otimização e o processamento de grandes volumes de dados não é igual em todos os paradigmas da programação, tal como os seus cuidados, tivemos a possibilidade de por em prática os conhecimentos adquiridos na unidade curricular. Tivemos de ter noções sobre classes, tendo sempre cuidado com o encapsulamento dos dados, utilizando o clone ao receber objetos à entrada dos métodos ou ao retornar objetos.

O processo de desenvolvimento da nossa aplicação realizou-se de forma progressiva. Começamos por definir as classes chave, de seguida implementamos os requisitos básicos, tendo criado uma classe de teste onde foram sendo testadas as funções implementadas ao longo do trabalho.

Foi necessário decidir o tipo de dados a utilizar ao definir conjuntos, tendo sido implementados *TreeSets* e *HashMaps*. Implementamos *TreeSet* para podermos utilizar a interface *Comparable*. Por exemplo, na ordenação dos clientes de modo a responder ao requisito aplicação sobre os top 10 clientes. Como nestes casos não havia necessidade de ter uma chave de procura, optamos pelo Set. O *HashMap* utilizamos por exemplo na *UMCarroJa*. Assim conseguimos ter um conjunto de *Client*, *Owner* e *Vehicle* onde a chave seria o nif, e a informação da chave o utilizador/veículo em si.

Para além disso, tivemos de ter noções sobre iteradores externos e internos, *Iterator* e *Stream*. A escolha destes iteradores teve sempre em atenção a situação onde estavam a ser aplicados e a dua eficiencia. A titulo de exemplo temos a utilização da *Stream* para filtrar e verificar a existência de elementos sem ter de percorrer o ciclo até ao fim.

Recorremos também à *Herança* para reutilização de código e compatibilidade de tipos, nomeadamente na definição dos veículos e utilizadores. Foi utilizado também o *Abstract*, as interfaces, por exemplo *Serializable*, e *Comparable*, tornando assim o código modular, e a captura de erros com *Exceptions*.