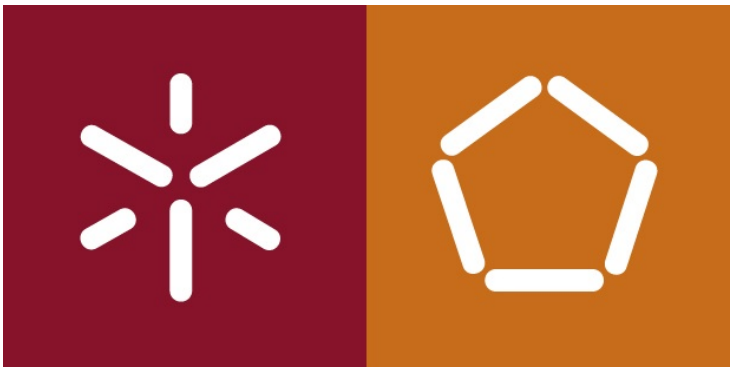
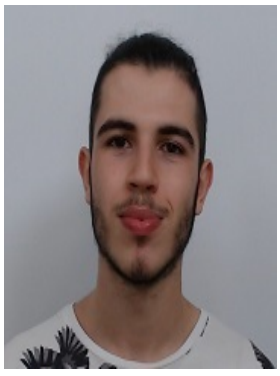


POO2019 - grupo 20

Ivo Baixo a86579, João Correia a84414, Lucas Leite a83364

Maio 2019


Conteúdo

1 Introdução **2**
1.1 Abordagem ao problema . . . 3

2 Principais Entidades do Projeto **4**
2.1 Ator . . . 4
2.1.1 Cliente . . . 5
2.1.2 Proprietário . . . 6
2.2 Viatura . . . 7
2.3 ViaturaInfo . . . 7
2.4 OcorrenciaAluguer . . . 8
2.5 NavegadorDeLista . . . 9
2.6 Como incluir novos tipos de viaturas na aplicação . . . 9

3 Estruturas de Dados **10**
3.1 BancoInformacao . . . 10
3.1.1 Armazenamento do estado do programa . . . 12
3.1.2 Tratamento dos Logs . . . 13

4 Comparadores e Exceções **14**
4.1 Comparadores . . . 14
4.2 Exceções . . . 15

5 Manual de Utilização **16**
5.1 Cliente . . . 16
5.2 Proprietário . . . 17

6 Conclusão (Análise crítica e perspetivas de melhoria) **18**

Capítulo 1

Introdução

Este trabalho foi realizado no âmbito da unidade curricular Programação Orientada a Objetos. O seu principal objetivo foi a criação de um software de aluguer de viaturas (conceito de certa forma idêntico ao do Uber), utilizando o paradigma da programação orientada a objetos. Em seguida se segue uma visão global do projeto.

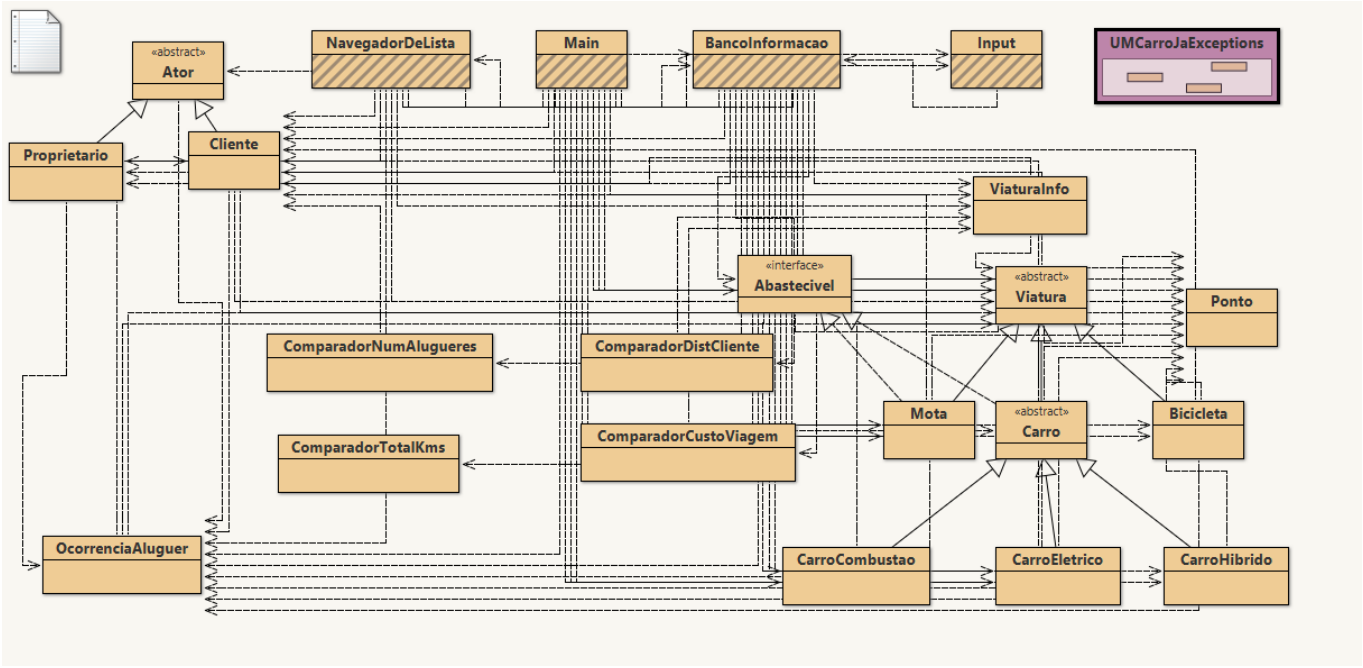


Figura 1.1: Arquitetura das classes da aplicação

1.1 Abordagem ao problema

Neste projeto tentamos seguir uma estratégia orientada a objetos. Começamos por planear quais seriam as entidades principais da aplicação (*Viaturas* e *Atores*) e onde seria armazenada e manipulada toda a informação da aplicaçao (classe *BancoInformacao*). Decidimos também que a parte relativa ao *output* seria tratada nas classes *Main* e *NavegadorDeLista* enquanto que os *inputs* seriam tratados numa classe *Input*. Utilizamos a arquitetura *Model-View-Controller* onde:

- Model: BancoInformacao
- View: Main e NavegadorDeLista
- Control: classe *Input*

Foi decidido ainda que uma viajem seria representada por uma classe própria, dando origem à classe *OcorrenciaAluguer*.

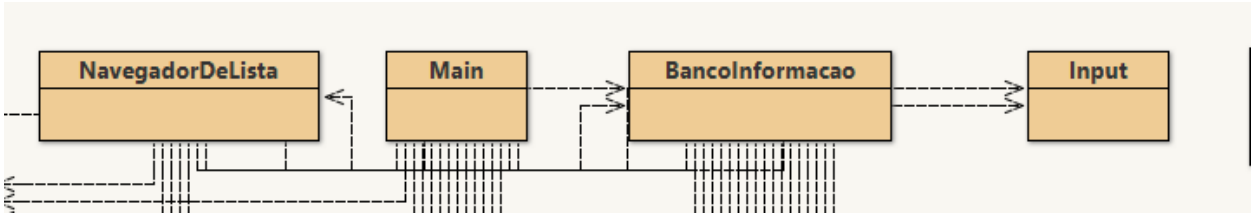


Figura 1.2

Capítulo 2

Principais Entidades do Projeto

2.1 Ator

```
/** Representa o nif de um cliente*/
private String nif;
/** Representa o email de um ator */
private String email;
/** Representa a password de um ator */
private String password;
/** Representa a morada de um ator */
private String morada;
/** Representa a data de nascimento de um ator */
private LocalDateTime dataNascimento;
/** Representa o nome de um ator */
private String nome;
/** Lista com os alugueres realizados pelo ator */
protected List<OcorrenciaAluguer> historial;
/** Total de pontos que lhe foram atribuidos na classificaçao */
private float classificacao;
/** nº de vezes que foi classificado */
private int nClassificacoes;
```

Figura 2.1: Variáveis de instância de Ator

Classe abstrata que representará um utilizador da nossa aplicação. Esta classe será depois extendida ou pela classe *Cliente* ou pela classe *Proprietário*. Contém informação básica necessária sobre o utilizador assim como o historial de alugueres e a classificação do utilizador.

2.1.1 Cliente

```
/** Posicao onde o cliente se encontra */
 private Ponto posCliente;
/** Posicao para onde o cliente se pretende deslocar */
 private Ponto posDestino;
/**
 * Inteiro que revela informaçao em relaçao ao ultimo pedido feito pelo cliente
 * 0 - Nenhum pedido foi feito
 * 1 - O Pedido esta a ser processado
 * 2 - O Pedido foi rejeitado
 * 3 - O Pedido foi aceite e ainda falta classificar a viagem
 *  */
 private int estadoPedido;
```

Figura 2.2: Variáveis de instância de Cliente

Esta classe extende a definição da classe Ator, pelo que irá conter as variáveis de instância dessa classe como a morada, nome, etc. Para além dessas, a classe Cliente terá três variáveis de instância específicas desta classe, estas são:

- **int** estadoPedido: - Este inteiro servirá como um código para o estado do pedido mais recente que o cliente possa ter feito, indicando se já foi processado ou não.
- **Ponto** posCliente: - Um ponto que simboliza a posição onde o cliente se encontra, será bastante importante para calcular a distância do cliente à *viatura* que pretende alugar.
- **Ponto** posDestino: - Um ponto que simboliza a posição para onde o cliente pretende viajar. É bastante utilizado na parte que diz respeito à logística do aluguer, sendo útil no cálculo do combustível a gastar na viagem, assim como o custo e duração desta mesma.

2.1.2 Proprietário

```
/** Lista de pedidos de aluguer */
private List<OcorrenciaAluguer> pedidosAluguer;
```

Figura 2.3: Opções Proprietário

Esta classe também extende a definição da classe *Ator*, visto ser uma classe que representa uma porção dos utilizadores.

A classe *Proprietário* terá como variável uma lista de objetos da classe *OcorrenciaAluguer*, isto serão os pedidos de aluguer que o proprietário ainda não processou. Caso eles sejam aceites, serão movidos para o historial, caso sejam rejeitados serão eliminados por completo.

Mais informação sobre o funcionamento das classes pode ser encontrado no tópico do *Manual de Utilização* (Capítulo 5).

2.2 Viatura

As **Viaturas** são os objetos que podem ser alugados. Nesta classe são guardadas informações gerais para todas as viaturas como:

- estado (se está livre ou ocupada)
- matrícula
- histórico dos alugueres daquela viatura
- id do proprietário

Nesta aplicação existem três tipos de viaturas: **Carros**, **Motas** e **Bicicletas**. Existem dentro dos **Carros** três tipos de motores: motor a combustível, híbrido e elétrico.

2.3 ViaturaInfo

Esta classe contém informação relativa a uma viagem (com um determindado carro). Contém como variáveis de instância:

```
private Cliente cliente;
private Viatura viatura;
private float distCliente;
private float custoViagem;
```

Figura 2.4: Variáveis instância **ViaturaInfo**

A principal função desta classe é permitir ordenar os carros por fatores que dependem também do cliente, como por exemplo a *distância ao cliente* ou o *custo da viagem*.

Serão os objetos desta classe que serão inseridos num *treeSet* para serem ordenados quanto ao parâmetro indicado pelo utilizador, pelo que foi por essa razão que os comparadores da nossa aplicação foram feitos para esta classe e não para a classe *viatura*, que não consegue conter informação relativa ao cliente, ao passo que esta classe consegue.

2.4 OcorrenciaAluguer

Nesta classe estão armazenas as informações relativas ao aluguer de uma viatura (a própia viatura vai conter uma List<OcorrenciaAluguer>).

```
private LocalDateTime data;
private Viatura viatura;
private String  emailCliente;
private float custo;
private Ponto partidaCliente;
private Ponto partidaViatura;
private Ponto destino;
private double distancia;
private boolean foiClassificada;
```

Figura 2.5: Variáveis instância **OcorrenciaAluguer**

Como se pode ver na figura, nesta classe ficam assim guardados todos estes dados. Os utilizadores desta aplicação terão um registo de todos os alugueres feitos. Em especial, o proprietário terá até duas listas de *OcorrenciaAluguer*, uma relativa aos alugueres que já foram processados e outra relativa aos alugueres que ainda não foram aceites ou rejeitados.

2.5 NavegadorDeLista

Para facilitar a navegação no terminal do *BlueJ* criou-se uma classe **NavegadorDeLista** que divide a informação a ser apresentada por várias páginas, permitindo uma fácil movimentação pelas listas de utilizadores,viaturas e pedidos de aluguer.

```
/** ArrayList que guarda todas as sublistas com x carros*/
ArrayList<ArrayList<String>> listaStrings;
/** Numero de elementos a aprecer por pagina*/
private int tamanhoPag;
/** Indice da pagina em que estamos*/
private int pagAtual;
/** Numero de paginas total*/
private int nPags;
```

Figura 2.6: Variáveis instância NavegadorDeLista

Esta classe divide assim a informação a apresentar por páginas e permite que o utilizador 'salte' entre estas à vontade.

```
A mostrar a página 1 de 1
Email                Matricula   Marca    Data       Hora     Ponto do cliente  Ponto da viatura   Destino

300003856@gmail.com   FB-39-32 - Lotus   24/5/2019   23h34m  (x:19.69 , y:-17.3) (x:-69.58 , y:84.12)  (x:-



Prima',' para regressar à página anterior, '.' para  ir para a página seguinte, e qualquer outra tecla para
```

Figura 2.7: Exemplo NavegadorDeLista

2.6 Como incluir novos tipos de viaturas na aplicação

A nossa aplicação foi construída de forma a que fosse o mais simples possível acrescentar novos tipos de viaturas. Para adicionar um novo tipo de viatura essa classe deve extender a classe abstrata *Viatura* e, caso a nova viatura consuma combustível, implementar a interface *abastecível*.

Capítulo 3

Estruturas de Dados

3.1 BancoInformacao

Esta classe tem um papel muito importante no projeto, visto que é nela que se encontra e é gerida toda a informação da aplicação, nomeadamente os *utilizadores* e as *viaturas*. Optamos por armazenar os dados em *HashMaps* devido à sua eficiência e fácil acessibilidade.

```
/** Frota de carros*/
private Map<String,Viatura> viaturas;
/** Map de utilizadores*/
private Map<String,Ator> utilizadores;
/** Utilizador atual */
private Ator utilizador;
```

Figura 3.1: Variáveis de instância de **BancoInformacao**

Relativamente à estrutura de dados dos utilizadores ,esta será um *Map* em que a chave de cada elemento será o *e-mail* do utilizador. Esta escolha foi feita a pensar que seria a informação mais fácil do utilizador se lembrar. Como tal, grande parte dos métodos relativos aos utilizadores assentam em cima da *API* dos maps do *java*, visto que a maioria dos métodos relativos à estrutura de dados dos utilizadores serão de insereção. Temos como exemplo de métodos relativos aos utilizadores os métodos *loginUtilizador* , *registaCliente* e *registaProp*.

Relativamente à estrutura de dados das viaturas ,esta também será um *map*. A chave será a *matrícula* de cada viatura, visto ser uma informação que é universal a qualquer viatura da nossa aplicação e porque uma matrícula será sempre única.

Na classe *BancoInformacao*, grande parte dos métodos utilizam de alguma forma esta estrutura de dados, o que é plausível, visto que grande parte do processamento global da nossa aplicação se centra nas próprias viaturas. Alguns destes métodos serão os *ordenaViatura(...)* que se encarregarão de devolver um *Set* (que no método estará implementado sobre a forma de um *TreeSet* de maneira a permitir a ordenação das viaturas) de objetos da classe *ViaturaInfo*.

Aqui estão definidos métodos que permitem registar viaturas na aplicação (assim como eliminar), métodos que permitem ordenar as viaturas por diversos parâmetros, entre outros.

```
/** Insere uma viatura na frota */
public void registaViatura(Viatura viatura) throws JaExisteViaturaException{
if(existeViatura(viatura.getMatricula())){
throw new JaExisteViaturaException(viatura.getMatricula());
}else{
this.viaturas.put(viatura.getMatricula(),viatura.clone());
}
}
```

Figura 3.2: Método de registo de viaturas

```
/**
* Devolve os viaturas ordenador por custo de viagem
* Recebe tambem um argumento distanciaLimite que,
* se positivo, filtra os viaturas que estao a uma distancia menor qure esse argumento
*/
public Set<ViaturaInfo> ordenaCustoViagem(Cliente cliente, int distanciaLimite,
                                          int filtroCombustivel){
 Set<ViaturaInfo> ts = new TreeSet<>(new ComparadorCustoViagem());
 boolean inDistancia = false;
 for(Viatura viatura : this.viaturas.values()){
  if((viatura.distancia(cliente.getPosCliente())*1000<=distanciaLimite))
  inDistancia= true;
  else
  inDistancia=false;
  if (viatura.viagemPossivel(cliente.getPosDestino())
  && inDistancia && filtraCarroMotor(viatura,filtroCombustivel)){
       ts.add(new ViaturaInfo(viatura,cliente));
  }
 }
 return ts;
}
```

Figura 3.3: Método de ordenação de viagens pelo custo

3.1.1 Armazenamento do estado do programa

A classe **BancoInformacao**, como dito anteriormente, trata de gerir a informação da aplicação. É nesta classe que se guarda o estado do programa num ficheiro binário, assim como se carrega também o estado através de um ficheiro.

```
public void guardaEstado(String nomeFicheiro)
throws FileNotFoundException, IOException{
  FileOutputStream fos = new FileOutputStream(nomeFicheiro);
  ObjectOutputStream oos = new ObjectOutputStream(fos);
  oos.writeObject(this);
  oos.flush();
  oos.close();
  }
```

```
public static BancoInformacao carregaEstado(String nomeFicheiro)
throws FileNotFoundException,IOException,ClassNotFoundException{
  FileInputStream fis = new FileInputStream(nomeFicheiro);
  ObjectInputStream ois = new ObjectInputStream(fis);
  BancoInformacao bi = (BancoInformacao)ois.readObject();
  ois.close();
  return bi;
  }
```

Figura 3.4: Métodos de leitura e escrita de um ficheiro

3.1.2 Tratamento dos Logs

É também na classe *BancoInformacao* que se interpretam os *logs*.

```
/** Interpreta a lista de strings que representam o ficheiro de logs*/
public List<Exception> interpretaLogs(List<String> logs) {
```

Figura 3.5: Método que interpreta os Logs

Este método, sempre que encontra uma ação que não consegue interpretar (por exemplo querer registar uma viatura que já está registada), adiciona uma exceção a uma lista de exceções que depois é apresentada na *main*.

```
public void processaAluguerLogs(String idCliente, double posXDest,
double posYDest,String tipoCombustivel,String preferencia)
  throws AluguerImpossivelException{
```

Figura 3.6: Método que processa os alugueres vindos de logs

O método **processaAluguerLogs** faz um trabalho ligeiramente diferente dos métodos de aluguer normais, visto que não é necessário que o proprietário aceite o aluguer.

Capítulo 4

Comparadores e Exceções

4.1 Comparadores

Neste projeto foram usados quatro *comparadores*:

- **ComparadorDistCliente** - Compara duas viaturas pela sua distância a um cliente.
- **ComparadorNumAlugueres** - Compara o número de alugueres de dois clientes.
- **ComparadorTotalKms** - Compara o número de kms percorridos entre dois clientes.
- **ComparadorCustoVIagem** - Compara duas viaturas pelo custo da viagem.

4.2 Exceções

Foi necessária a criação de subclasses de **Exception** para permitir que a aplicação fosse capaz de lidar diverso tipo de erros.

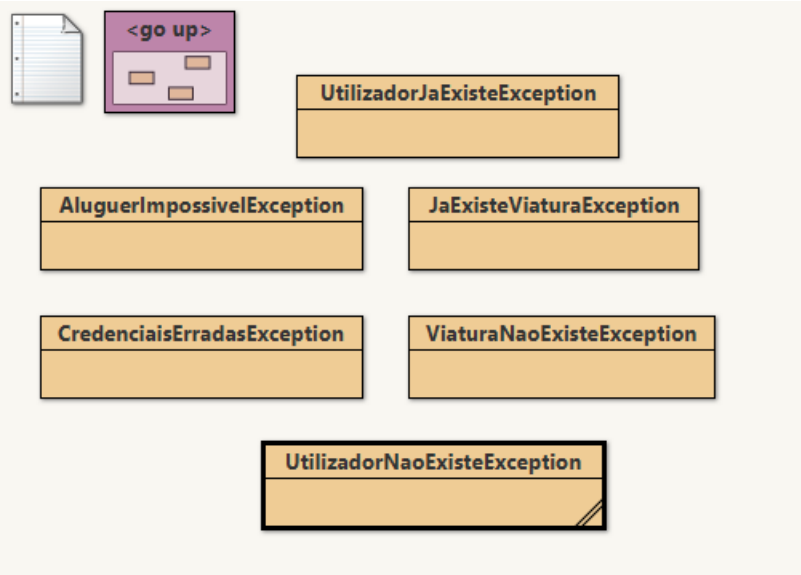


Figura 4.1: Exceções

As exceções criadas tratam de situações onde se tenta registar um utilizador ou viatura já existentes, de situações de login onde são dadas credenciais erradas e situações onde se tenta fazer um aluguer e não é possível (muitas vezes quando o utilizador tem um pedido já em espera).

Capítulo 5

Manual de Utilização

Aquando o início da aplicação, é apresentado ao utilizador o menu inicial, onde este pode decidir se pretende registar-se na aplicação ou se deseja fazer login.

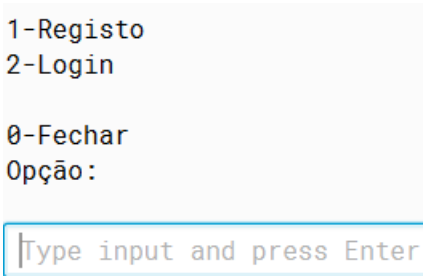


Figura 5.1: Menu inicial

Caso decida registar-se ser-lhe-á pedido que introduza as informações básicas de um utilizador do sistema, como Nif, email, morada, etc. Caso decida fazer login, após introduzir as credenciais corretas, as opções serão diferentes consoante o utilizador seja um *Cliente* ou um *Proprietário*.

5.1 Cliente

Ao entrar no menu de cliente, o utilizador poderá escolher uma das seguintes opções:

Aluguer de uma viatura: Se o utilizador decidir alugar uma viatura, terá várias escolhas acerca de como ordenar as viaturas:

- Ordenar as viaturas pela distância a si
- Ordenar as viaturas pelo custo da viagem
- Ordenar as viaturas pela distância a si, mas com um limite da distância a pé
- Alugar por um Id específico, apresentando todas as viaturas disponíveis sem nenhuma ordenação especial
- Alugar por um Id específico, apresentando todas as viaturas disponíveis sem nenhuma ordenação especial
- Alugar por uma marca, onde são apresentadas todas as viaturas a marca indicada pelo utilizador

Para além de escolher uma das opções acima mencionadas, o utilizador terá ainda de indicar as suas coordenadas e em que tipo de viatura pretende viajar. O sistema encarregar-se-á então de apenas apresentar ao utilizador as viaturas em que é possível efetuar a viagem, evitando assim que seja escolhida uma viatura que esteja indisponível.

Quando o utilizador faz um pedido de aluguer uma viatura, fica impedido de efetuar outro aluguer até que o proprietário aceite ou rejeite o pedido que fez atualmente.

Fatores de aleatoriedade Caso o utilizador possua uma classificação acima de 95%, terá um desconto automático de 10%. Os preços de aluguer variarão consoante a hora do dia: entre as 8 e as 10 horas e as 17 e 19,(horas de ponta matinais e de tarde) o preço sofrerá um aumento de 25% e entre as 12 e as 14 o preç sofrerá um aumento de 10%.

Consulta do historial O cliente pode consultar o seu historial de viagens efetuadas. Para isso terá de indicar o intervalo de tempo em que as viagens foram efetuadas que pretende consultar.

Consulta de Informação O cliente pode consultar as informações relativas a si, como o nif, morada, etc. Pode ainda consultar o número de alugueres que já efetuou e ainda a distância percorrida.

5.2 Proprietário

Quando um proprietário entra na aplicação, terá acesso às seguintes opções:

Registar uma viatura O proprietário regista uma viatura. Terá de introduzir os dados relativos à viatura como consumo, marca, tipo de viatura, etc. É efetuado um controlo para que não seja introduzida uma matrícula já existente.

Verificar pedidos de aluguer Aqui o proprietário poderá gerir todos os pedidos que foram feitos para alugar uma das suas viaturas. Os pedidos serão apresentados pela sua ordem cronológica e o proprietário terá de decidir se aceita o pedido ou o rejeita.

Consultar o historial Após introduzir uma data inicial e uma data final, serão apresentados ao proprietário todas as viagens efetuadas por todas as suas viaturas nesse intervalo de tempo.

Gerir viaturas Neste menu serão apresentadas ao utilizador todas as suas viaturas, onde o utilizador poderá alterar informação relativamente a elas. Se a viatura o suportar, o proprietário poderá ainda abastecê-la.

Gerir classificações Aqui, serão apresentadas ao proprietário todas as viagens que ainda este ainda não classificou. Assim como na mecânica de gerir os pedidos de aluguer, também aqui as viagens serão apresentadas por ordem cronológica da mais antiga para a mais recente. O utilizador classificará o cliente então com um inteiro entre 1 e 100.

Top 10 clientes por número de alugueres/distância ao cliente Nestas duas opções, serão apresentados ao proprietário os 10 clientes mais ativos da aplicação em relação ao número de alugueres ou à distância total percorrida.

Capítulo 6

Conclusão (Análise crítica e perspetivas de melhoria)

Neste projeto foram cumpridos os objetivos da programação orientada a objetos (há encapsulamento do código, é possível a reciclagem do código e ainda facilmente se expande a aplicação).

Uma possível forma de melhorar seria utilizando *API's* externas como a do *Google Maps*, para que houvesse um melhor controlo das posições das entidades, ou como uma *API* para controlo meteorológico que tornaria a estimativas de tempo das viagens muito mais certas.

A nossa principal dificuldade foi a implementação da mecânica de avaliação para o proprietário, em que, ao invés do cliente que só tem de avaliar uma viagem de cada vez, o proprietário poderá ter que avaliar uma quantidade delas de seguida, devido à nossa decisão de separar a mecânica de gestão dos pedidos da mecânica de avaliação das viagens.

O trabalho permitiu-nos entender melhor como estruturar uma aplicaçãol, aproximando-nos mais do tipo de programação que é feita no *'mundo do trabalho'*.