Programação Orientada aos Objetos

Relatório de Projeto

Mestrado Integrado em Engenharia Informática

Isabel Sofia Pereira
A76550

Liliana Monteiro
A76956

Maria de La Salete Teixeira
A75281

11 de junho de 2017

Conteúdo

0.1 Introdução 2
0.2 Descrição da arquitetura de classes utilizada 3
0.2.1 Classe Espaco2D 3
0.2.2 Classe Utilizador 3
0.2.3 Classe Veículo 4
0.2.4 Classe Viagem 5
0.2.5 Classe EmpresaTaxis 5
0.3 Descrição da Aplicação desenvolvida 6
0.3.1 Classe UMeR 6
0.3.2 Classe UMeRApp 8
0.4 Conclusão 12

0.1 Introdução

A *Uber* é um serviço de transportes de passageiros, que através da sua aplicação, permite viajar de forma rápida, segura e em questão de minutos. Assim, a *Uber* conecta clientes a motoristas, em qualquer lugar e a qualquer hora. Esta tornou-se um grande sucesso devido à facilidade de pagamento e à rapidez de serviço, isto é, ao possuir informações precisas e em tempo real da localização dos seus motoristas, o utilizador pode chamar, se assim pretender, o carro mais próximo, reduzindo o tempo de espera.

No âmbito da unidade curricular de Programação Orientada aos Objetos, foi-nos proposto o desenvolvimento de uma aplicação *UMeR* que faça concorrência à *Uber*. Deste modo, o desafio foi desenvolver a aplicação em *Java* permitindo que um utilizador consiga realizar uma viagem num dos veículos UMeR. O cliente pode, por exemplo, aceder ao seu histórico com toda a informação sobre as viagens efetuadas ou até dar uma classificação ao motorista. Para tal, criamos todos os mecanismos necessários, tais como, os clientes, motoristas, automóveis, a marcação das viagens, a realização das mesmas e respectiva aplicação do preço. Para além disso, criamos também toda a parte de interação com cliente e guardamos o registo de todas as operações efectuadas para que depois seja possivel disponibiliza-las novamente.

0.2 Descrição da arquitetura de classes utilizada

0.2.1 Classe Espaco2D

Na classe *Espaco2D* defimos a posição **xOy** no espaço de um determinado utilizador. Deste modo, permite-nos ter a localização de um utilizador, caracteristica muito importante, visto que é através da localização que podemos calcular, por exemplo, quantos kilometros foram percorridos numa determinada viagem ou até ver qual o veículo mais próximo do cliente.

Figura 1: Classe Espaco2D

0.2.2 Classe Utilizador

A classe *Utilizador* permite-nos criar a identificação de um utilizador. Aqui criamos todas as variáveis necessárias para que quer o cliente, quer o motorista sejam identificados. Para tal, temos nesta classe definidos o **email** e a **password** do utilizador, que são essenciais para que o utilizador possa fazer login na aplicação, o **nome**, a **morada** e a **data de nascimento**.

Como quer os clientes, quer os motoristas necessitavam de uma identificação e de modo a evitar repetir código sem haver necessidade, criamos duas classes a *Cliente* e a *Motorista* por herança, isto é, que são subclasses da classe *Utilizador*. Como na aplicação apenas criamos clientes e motoristas, a classe *Utilizador* passou a ser abstracta, visto que não criavamos nenhum objeto a partir dos seus construtores, mas era necessário manter a hierarquia da herança.

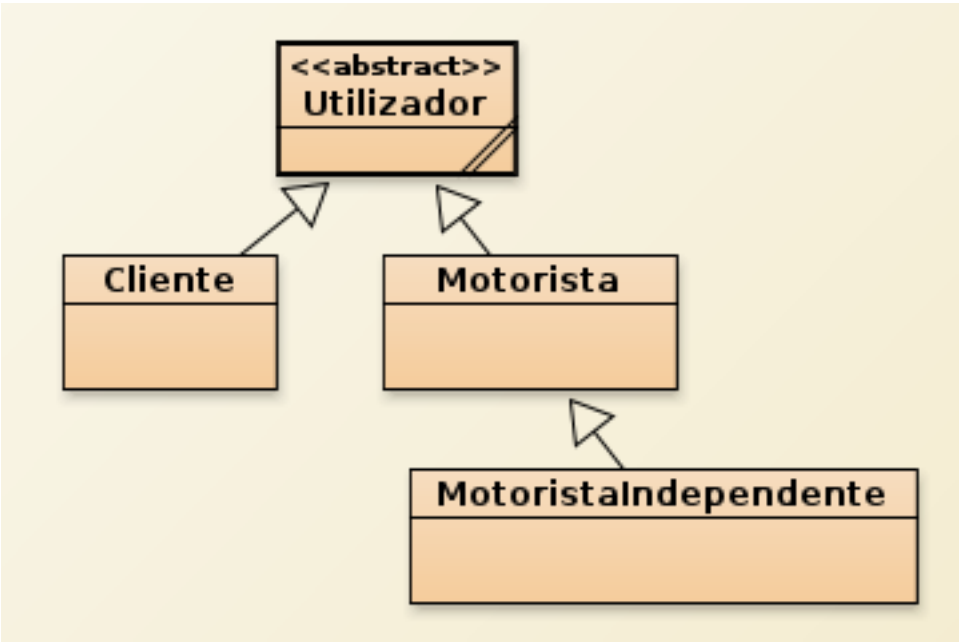

Figura 2: Classe Utilizador

Classe Motorista

Tal como foi referido anteriormente, a classe *Motorista* é uma subclasse de *Utilizador*, herdando, portanto, uma identificação. Para além disso, necessita, também de um **histórico de viagens efetuadas**, sendo para tal implementado um *TreeSet*, necessita também dos **kilometros totais percorridos**, da **disponibilidade**, da **quantidade de viagens efetuadas**, de uma **classificação média** e de um **grau de cumprimento**. Todas estas informações são essenciais para o perfil do motorista.

Adicionalmente, devido à questão *Motoristas individuais vs Empresas de Táxis* decidimos que seria uma boa abordagem, criar uma classe *Motorista* e outra *Motorista Independente*, ligados por herança, assim permite-nos ter um motorista ao qual mais tarde associamos um veículo, que é ideal para as *Empresas de Táxis*, e ter um motorista que já possui o seu carro. Esta separação é muito útil visto que nas *Empresas de Táxis*, estas possuiem vários veículos que podem ser utilizados por vários motoristas, logo o motorista, neste caso, não pode, à partida, ter o seu carro pessoal.

Classe Motorista Independente

Esta classe é uma subclasse de *Motorista*, e como mencionado previamente, apenas acrescenta a toda a informação herdada do *Motorista* um **veículo**, que é pessoal e com o qual pratica os serviços *UMeR*.

Classe Cliente

A classe *Cliente* é uma subclasse de *Utilizador*, logo herda todas as informações desta. No entanto, tal como o motorista, para além de ter uma identificação, também necessita de ter um **histórico de viagens do cliente**. No entanto, as viagens do cliente contem mais informação que a do motorista. Nomeadamente, o nome do motorista que o serviu e a classificação que o cliente lhe atribui no fim da viagem. Já por isso é que o historico de viagens não é definido na super classe que os une. Desta forma, implementamos um *TreeSet*, isto é, uma árvore binária auto-balanceada, que nos permite guardar todas as informações de uma viagem sem que se repita informação.

0.2.3 Classe Veículo

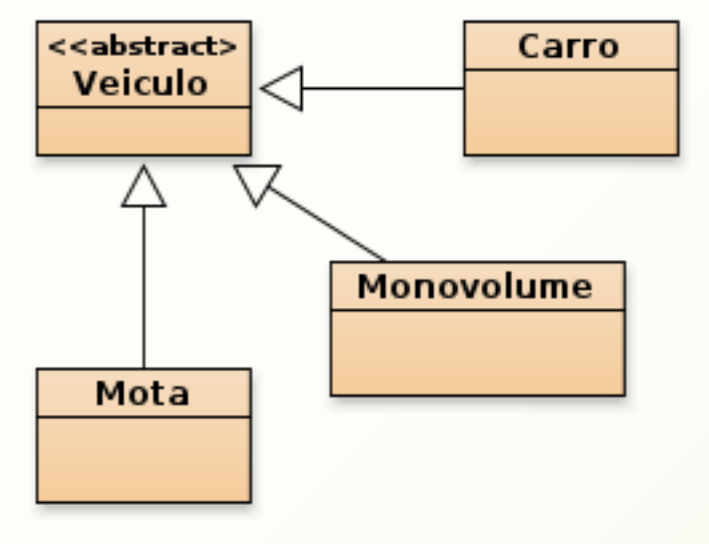

Figura 3: Classe Veiculo

A classe *Veiculo*, essencialmente, permite-nos criar um veículo para que o motorista possa usar. Para efeito, necessita de **matricula**, **número de lugares**, da **velocidade média por kilometro**, do **preço médio por kilometro**, da **fiabilidade**, de uma **localização** e a **disponibilidade** do veículo. Esta última é necessária para as *Empresas de Taxi* visto que temos que, de uma lista de veículos, associar um a um motorista e pô-lo indisponível para que mais nenhum motorista o possa usar ao mesmo tempo.

Na *UMeR* é possível haver várias classes de veículos, assim criamos, por herança, as classes *Carro*, *Monovolume* e *Mota*. Da mesma forma que a classe *Utilizador*, a classe *Veículo* é abstracta, visto que, não criamos nenhum objeto a partir dos seus construtores, mas é necessária para garantir uma hierarquia de herança. Assim, elimina-se a possibilidade de erros de criação de veículos que não estão disponiveis na aplicação.

Caso futuramente se queira adicionar novos tipos de veículos à aplicação, basta criar subclasses a *Veiculo* com o nome do tipo desejado.

Classes Carro, Mota e Monovolume

As classes *Carro*, *Mota* e *Monovolume* são subclasses de *Veículo* e servem o propósito de haver várias tipos de veículos que um motorista pode utilizar.

0.2.4 Classe Viagem

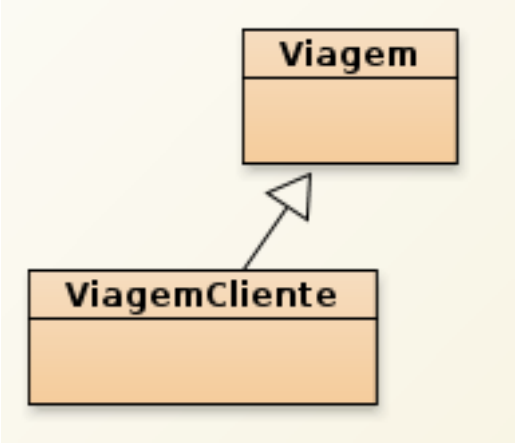

Figura 4: Classe Viagem

Na classe *Viagem* definimos todas as informações necessárias a uma viagem, como, a **data e o tempo** local, as **coordenadas no inicio da viagem**, as **coordenadas do destino**, o **veículo utilizado**, os **kms percorridos**, o **tempo, o preço real e estimado da viagem**. Aqui definimos um comparador que ordena os *TreeSets* criados em *Motorista* e *Cliente* pela ordem das datas de modo a ser mais fácil organizar e procurar entre datas o histórico dos motoristas e dos clientes.

Classe ViagemCliente

A classe *ViagemCliente* é uma subclasse de *Viagem* que específica certas informações que são apenas do histórico do cliente. Assim definimos o **nome do motorista** que serviu o cliente e a **classificação do motorista** que a este foi atribuída no final da viagem pelo cliente.

0.2.5 Classe EmpresaTaxis

Figura 5: Classe EmpresaTaxis

Uma empresa de táxis, contráriamente ao que ocorre na UMeR, possui motoristas e veículos criando-os de forma independente, isto é, é possivel associar qualquer motorista a qualquer veículo. Assim a classe *EmpresaTaxis* cria uma empresa com um **nome** e com dois *HashMap* um para

guardar os **motoristas** e outra para guardar os **veículos**. Para além disso permite-nos atualizar certos dados dos motoristas e veiculos à medida que novas viagens vão sendo realizadas por estes.

0.3 Descrição da Aplicação desenvolvida

0.3.1 Classe UMeR

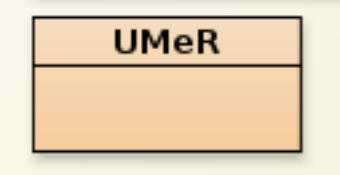

Figura 6: Classe UMeR

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

Com a *UMeR* temos a possibilidade de criar um novo registo para utilizadores, veículos e empresas. Tal como podemos adicionar também podemos remover, e validar o acesso de um utilizador. Além dessas funcionalidades, pode-se proceder à realização de uma nova viagem, podendo esta ser realizada chamando um veículo especifico ou o veículo mais próximo. Tendo sido a viagem realizada, é necessário calcular o tempo, e o custo real e estimado associado a esta. É possivel também aceder-se ao histórico das viagens de um utilizador, ao total faturado quer dos motoristas independentes, quer das empresas de taxis, ao top dez clientes que mais gasta, e ao top cinco dos motoristas com mais desvio entre o preço estimado para a viagem e o preço real da viagem.

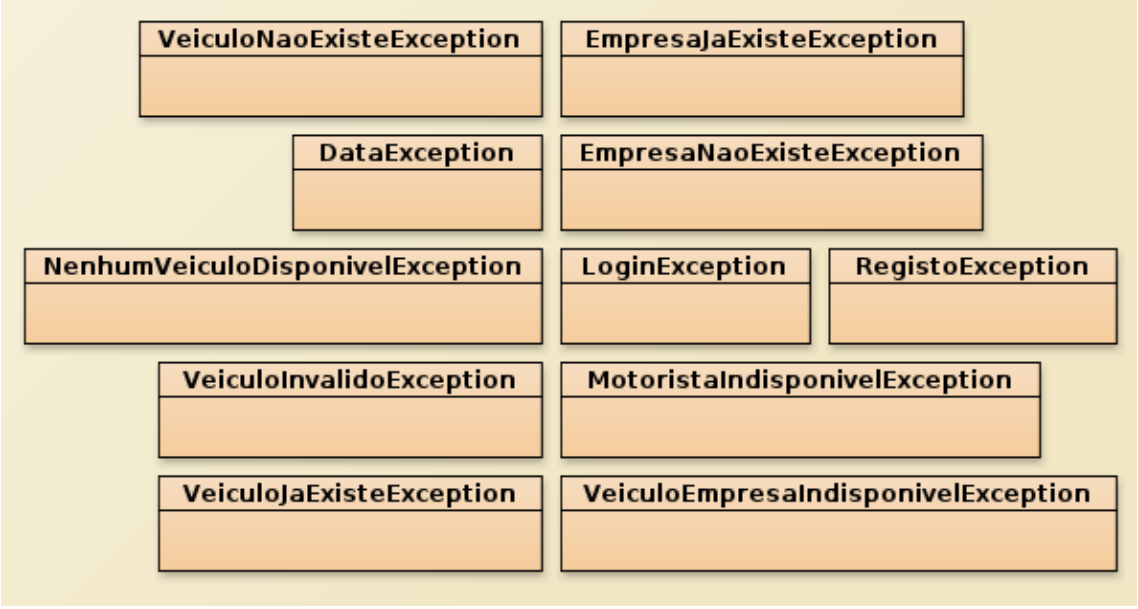

Figura 7: Exceções

Ao codificar estas funções, foi necessário que várias delas tivessem de ser criadas com exceções. Sendo assim, foram criadas as seguintes classes:

- *RegistoException* que ocorre caso ao criar uma conta, o email já exista na base de dados da aplicação;
- *LoginException* para quando o utilizador insere os dados errados ao tentar entrar na sua conta;

- *VeiculoInvalidoException* quando se tenta adicionar um novo veículo que não seja do tipo *Mota*, *Carro* ou *Monovolume*;
- *VeiculoJaExisteException* que devolve uma mensagem caso o veículo que se quer adicionar já existe na aplicação;
- *EmpresaJaExisteException* que ocorre quando se tenta adicionar uma nova empresa e já existe uma empresa com o nome dado;
- *VeiculoNaoExisteException* que devolve uma mensagem caso o veículo especifico procurado pelo cliente não exista;
- *VeiculoEmpresaIndisponivelException* acontece quando o veículo procurado pelo cliente, e este pertence a uma empresa, esteja indisponível;
- *MotoristaIndisponivelException* para caso o cliente ao procurar pelo veículo, o motorista não esteja disponível;
- *NenhumVeiculoDisponivelException* que devolve uma mensagem caso o cliente ao procurar pelo veículo mais proximo, nenhum veículo esteja disponivel de momento;
- *EmpresaNaoExisteException* que devolve uma mensagem caso a empresa não exista. Este erro poderá acontecer ao procurar o total faturado por uma empresa, ao tentar adicionar um novo motorista a uma empresa especifica, ao procurar um veículo especifico por empresa, ou ao procurar a lista de veículos de uma empresa;
- *DataException* 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;

Na função que calcula o tempo real de uma viagem, para além dos fatores aleatórios requeridos, fiabilidade do veiculo e o grau de cumprimento do motorista, foram adicionados fatores aleatórios que afetam a cidadão comum no seu dia-a-dia, nomeadamente o trânsito, a chuva, e o nevoeiro.

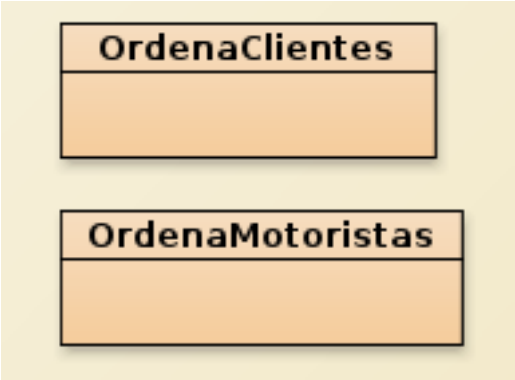

Figura 8: Classes utilizadas para responder às questões dos tops

Para responder às perguntas dos tops sentimos a necessidade de criar mais duas classes *OrdenaClientes* e *OrdenaMotoristas*. A primeira ordena decrescentemente os clientes pelo preço total gasto em viagens. A segunda ordena, também decrescentemente, os motoristas, podendo ser estes independentes ou pertencentes a uma empresa, pelo desvio de preços das suas viagens. Desta forma é criado nessas questões uma *TreeSet* com a ordenação correspondente. De seguida é adicionado na *TreeSet* os utilizadores em questão, podendo assim obter uma coleção organizada onde posteriormente apenas é necessário ir buscar os n primeiros utilizadores.

Com a realização das funcionalidades requeridas para o trabalho, começamos a pensar "Se eu fosse um utilizador desta aplicação que mais é que eu gostaria de ter acesso?". Com este pensamento em mente foram implementadas funções que informam o cliente quais os veiculos disponiveis na aplicação. Assim, para além do histórico de viagens que contem os veículos já utilizados, o cliente tem a hipotese de escolher um veiculo que nunca utilizou. Além destas funções, e agora a pensar nos motoristas, foi adiciona uma função que permite mudar a sua disponibilidade pois estes não

são obrigados a estar sempre disponiveis ou sempre em repouso. Para os motoristas independentes, foi ainda implementadas funções que permitem-no alterar o seu veiculo e a localização do mesmo, pois este não é um taxista, é provavelmente uma pessoa que utiliza o UMeR para ganhar dinheiro extra além de um trabalho já fixo, logo, este não pode permanecer sempre no local onde a viagem terminou, muito provavelmente vai proceguir com a sua vida pessoal.

Com a aplicação completamente funcional restava-nos conseguir que os dados que inserimos na aplicação não se perdessem com o termino do programa. Assim, usamos a biblioteca *java.io.** de modo a podermos aplicar métodos, tais como, *carregaEstado* ou *guardaEstado* permitindo-nos guardar o estado da aplicação, isto é, toda a informação que foi nela adicionada, num ficheiro para que mais tarde quando a aplicação for reiniciada esse ficheiro seja carregado e tenhamos acesso a todas as informações da aplicação.

0.3.2 Classe UMeRApp

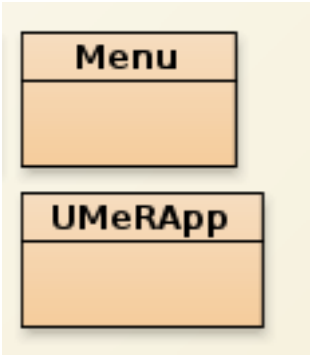

Figura 9: Classe UMeRApp

A *UMeRApp* é a aplicação que permite a interação com o utilizador, tendo vários menus com várias opções. Os menus foram gerados através da classe *Menu* fornecida nas aulas.

```
 *** Menu ***
1 - Novo Registo Cliente
2 - Novo Registo Motorista Independente
3 - Area Empresarial
4 - Login
0 - Sair
Opção:
```

Figura 10: Menu inicial

Esta é a estrutura do menu principal, o primeiro menu que aparece quando se inicia a aplicação. Este permite o registo de novos utilizadores, perguntado pelo email, password, entre outros, ou que o utilizador já com uma conta, entre na sua area pessoal.

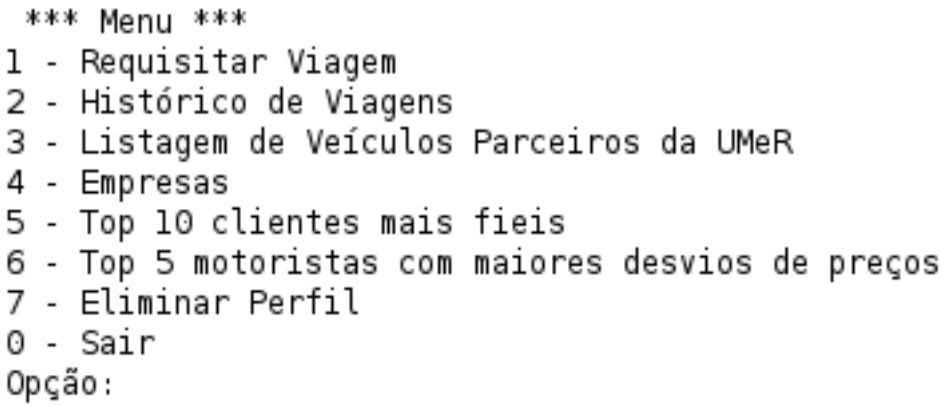

Figura 11: Menu para registo ou login de um cliente

Caso o utilizador que faça login ou se registe com sucesso seja um cliente, este é direcionado para a sua area pessoal com a estrutura a cima representada.

O cliente pode requerer viagens (onde posteriormente será redirecionado para um outro menu), verificar o seu histórico de viagens com a UMeR, verificar quais os veículos de Motoristas Independetes registados na aplicação, ir para o menu com dados de empresas, solicitar o top 10 clientes e o top 5 motoristas, eliminar a sua conta, ou sair da sua area pessoal.

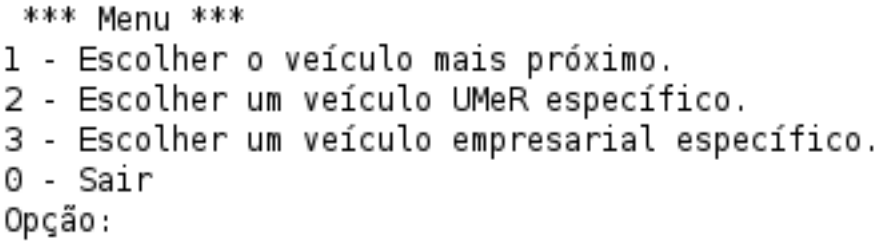

Figura 12: Menu de requisito de viagens

Caso o Cliente opte por requisitar uma viagem, este tem a opção de escolher um veiculo como mostrado na imagem.

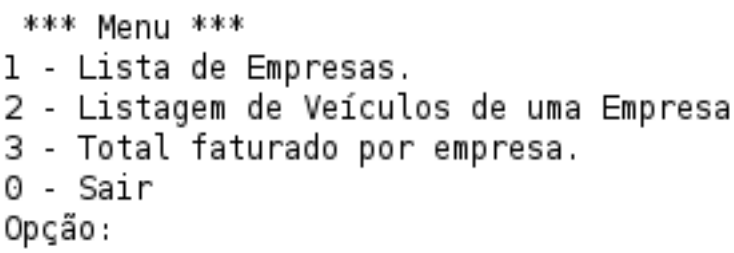

Figura 13: Menu de requisito de viagens

Um dos seus outros submenus é o menu com os dados das empresas. Aqui o Cliente pode verificar quais as empresas sócias da UMeR, quais os veículos disponíveis e o total faturado por estas.

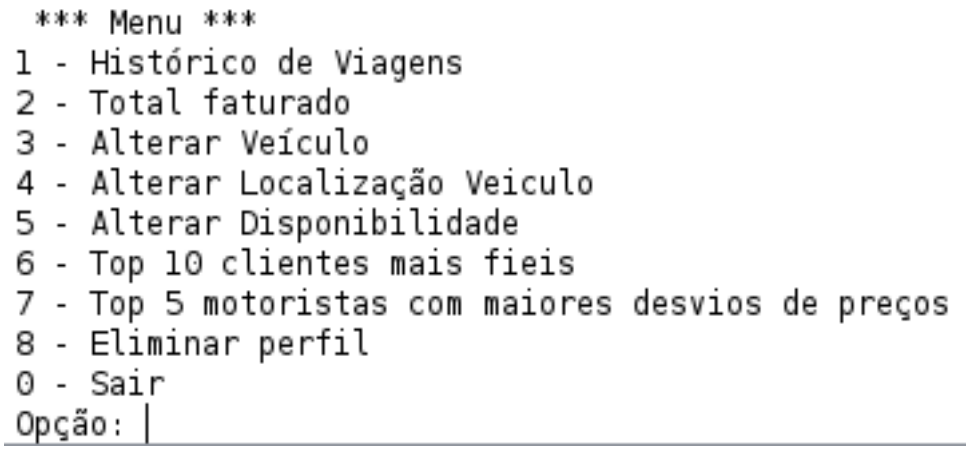

Figura 14: Menu de um Motorista Independente

Caso o utilizador chegue a este menu é porque é um Motorista Independente. Pode-se chegar aqui através do login ou do registo de um novo utilizador deste tipo.

O Motorista Independente verificar o seu histórico de viagens com a UMeR, consultar o total por ele faturado entre datas, alterar o seu veículo, a sua localização e a sua disponibilidade. Pode ainda consultar o top 10 clientes e o top 5 motoristas, eliminar a sua conta, ou sair da sua area pessoal.

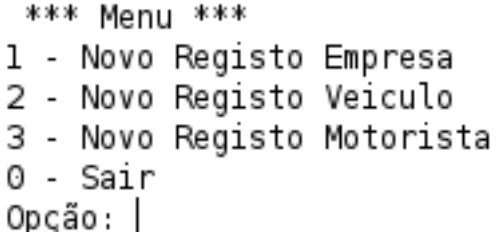

Figura 15: Área Empresarial

Caso no menu principal se tenha selecionado a "Area Empresarial", o utilizador da aplicação é redirecionado para este menu.

Aqui pode-se registar uma nova *Empresa de Taxis*, um novo *Veículo* ou um novo *Motorista de uma Empresa*.

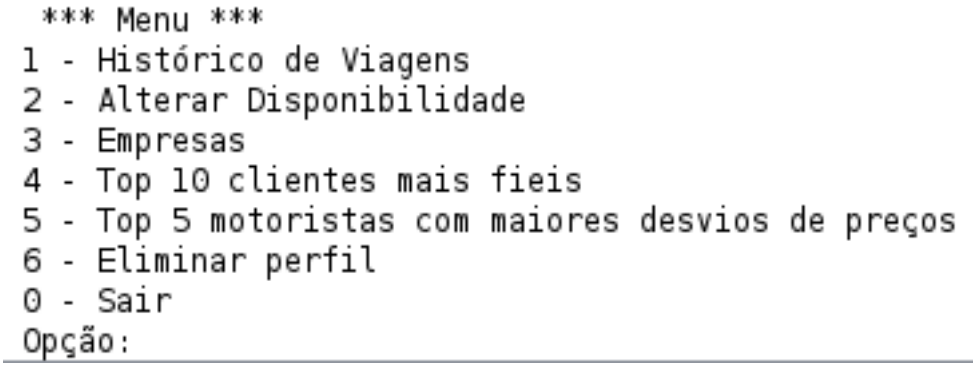

Figura 16: Menu do Motorista de uma Empresa de Taxis

Caso se tenha criado um novo Motorista de uma Empresa de Taxis, ou este fez login na sua conta, este é enviado para esta area.

O Motorista pode verificar o seu histórico de viagens, alterar a sua disponibilidade, ou ir, tal

como o Cliente, para o menu com os dados de empresas (já mostrado anteriormente). Pode ainda consultar o top 10 clientes e o top 5 motoristas, eliminar a sua conta, ou sair da sua area pessoal.

Assim, os utilizadores da aplicação têm cada um a sua area pessoa onde podem realizar ações ao seu dispor, tendo acesso a todos os requisitos do projeto.

0.4 Conclusão

O processo de desenvolvimento da aplicação *UMeR* occorreu de forma progressiva.

Começamos por definir as classes chave. Depois destas definidas, implementamos os requisitos básicos na UMeR tendo criado uma classe *Test* onde foram sendo testadas as funções implementadas ao longo do trabalho. Tendo essa parte do trabalho concluída e algum tempo ainda em mãos, conseguimos implementar com sucesso a Empresa de Taxis e os fatores de aleatoriabilidade. No entando, com a chegada de mais trabalhos não foi possível implementar as listas de espera.

Apesar de não termos concluido o trabalho no seu total, este foi muito importante porque tivemos a possibilidade de por em prática os conhecimentos adquiridos na unidade curricular.

Tivemos de ter noções sobre classes, tendo sempre o cuidado com o encapsulamento dos dados, utilizando o *clone* ao receber objetos à entrada dos métodos ou ao retornar objectos.

Foi necessário tomar decições sobre que tipo de dados utilizar ao definir conjuntos (Listas, Sets ou Maps), tendo implementado *TreeSets* e *HashMaps*. Implementamos *TreeSet* para podermos utilizar a interface *Comparable*. Por exemplo, na ordenação dos clientes pelos gastos para responder ao top 10 clientes, e à organização das viagens por data para ser mais pratico aceder ao registo de viagens entre datas. Como nestes casos não havia necessidade de ter uma chave de procura, optamos pelo Set. O *HashMap* utilizamos por exemplo na *UMeR*. Assim conseguimos ter um conjunto de *Utilizadores* onde a chave seria o email, e a informação da chave o utilizador em si. Utilizamos também para ter um conjunto de *EmpresaTaxis* onde a chave é o nome da empresa.

Para além disso, tivemos de ter noções sobre iteradores externos e internos, tendo utilizado *For*, *Iterator* e *Stream*. Estes iteradores foram utilizados conforme consideramos que seria o ideal para a situação onde estavam a ser aplicados. A titulo de exemplo temos a utilização da *Stream* para filtrar e verificar a existencia 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. Foi utilizado também o *Abstract*, a *Interface*, por exemplo *Serializable*, *Comparator* e *Comparable*, a *Composição* para reutilização de código nas classes, mas que não faria sentido estarem ligadas por herança, tornando assim o código modular, e a captura de erros com *Exceptions*.

Tendo posto em prática todos estes conhecimentos, ficamos com uma melhor preparação para o teste pois ficamos a entender melhor certas partes da matéria, nomeadamente *Stream*, *Coleções e Maps*.