Universidade do Minho

MIEI - Grupo 61

Programação Orientada aos Objetos

UMeR

(a) Sérgio Alves A78296

(b) Hugo Brandão A78582

(c) Tiago Alves A78218

Conteúdo

1 Introdução

Neste trabalho tínhamos como objetivo criar uma aplicação, em Java, para serviço de transporte de passageiros. Esta, teria de ser desenvolvida de forma a que toda a sua funcionalidade permitisse ao utilizador realizar uma viagem num dos táxis da UMeR. Para este processo ser bem realizado, implicava que desenvolvessemos um mecanismo que criasse utilizadores, motoristas, automóveis e posteriormente a marcação das viagens, a realização das mesmas e respectiva imputação do preço. Como é lógico, era obrigatório que todas as operações efetuadas na aplicação teriam de ser guardadas, para que quando arrancasse de novo, todos os processos realizados anteriormente fossem carregados com sucesso.

As funcionalidades que tínhamos como base eram:

- Os clientes poderiam:
  - Solicitar uma viagem ao táxi mais próximo das suas coordenadas;
  - Solicitar uma viagem a um táxi específico;
  - Fazer uma reserva para um táxi específico que, de momento, não estivesse disponível.
- Os Motoristas poderiam:
  - Sinalizar que estão disponíveis para serem requisitados;
  - Registar uma viagem para um determinado cliente;
  - Registar o preço que custou determinada viagem.

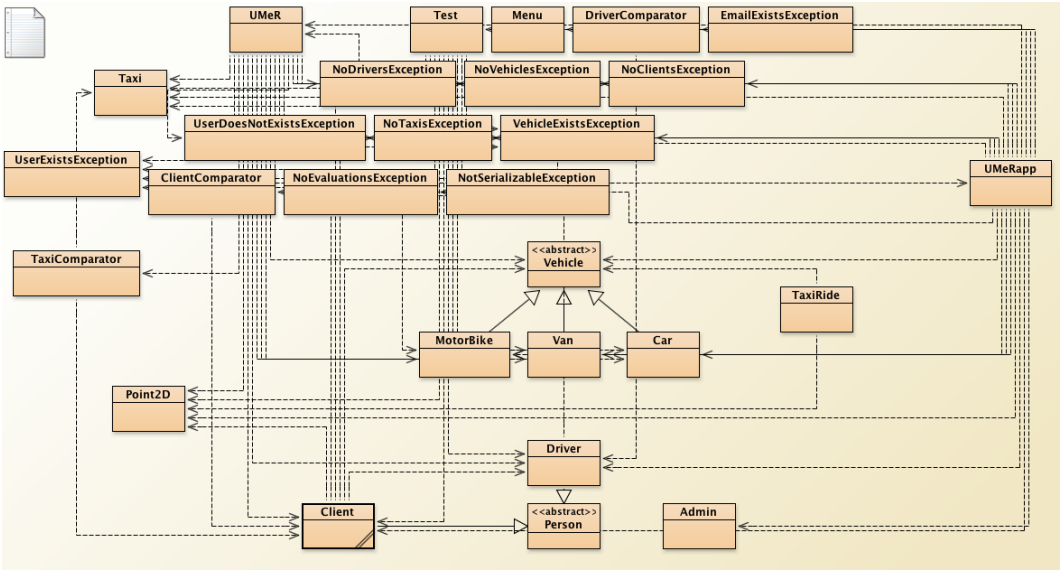
A nossa intenção para este projeto era desenvolver não só as funcionalidades básicas, como também as mais avançadas, como aquelas que eram requisitadas no enunciado: gestão de factores de aleatoriedade, na duração da viagem e fiabilidade das viaturas, e criação de viaturas com fila de espera e empresas de táxis UMeR.

2 Modularização - Classes criadas

O nosso pensamento inicial para a realização deste projeto foi criar as classes básicas, como por exemplo, as classes *Client*, *Driver*, *Vehicle*, *Taxi*. E à medida que íamos desenvolvendo a aplicação, acrescentávamos o que fosse necessário às classes.

Como seria de esperar, todas as classes que desenvolvemos no início do projeto sofreram bastantes alterações. Criámos classes abstratas, inserimos hierarquia (estes explicados mais abaixo), modulámos mais todo o código. Na imagem abaixo, está o diagrama deste projeto

Diagrama final do projeto



2.1 Classe *Person*

Antes de implementarmos esta super classe, na fase inicial do projeto, ainda não tínhamos pensado em hierarquia, apenas possuíamos as classes *Client* e *Driver*. No entanto, visto que com hierarquia de classes, o código é de mais fácil implementação e se torna mais simples e legível, decidimos então assim proceder.

Nesta classe, são guardados os valores básicos e comuns do cliente e do motorista, sendo eles o nome, o email, password, morada e data de nascimento. Adicionalmente, e pensando numa funcionalidade que permitisse tanto ao cliente como ao motorista ter acesso às viagens já realizadas, implementámos um *TreeMap* que associa uma determinada data(*key*) a um conjunto de viagens. Para guardar estas viagens, utilizámos um *ArrayList*. Não colocámos nesta super classe as coordenadas do utilizador, visto que apenas o cliente teria as suas próprias coordenadas, ou seja, o motorista não tem coordenadas individuais, elas são as mesmas que as do táxi em que ele estará a trabalhar.

```
1 public abstract class Person implements Serializable{
2   private String email;
3   private String password;
4   private String name;
5   private String address;
6   private String birthday;
7   private Map<Date, List<TaxiRide>> history;
8   ...
9 }
```

É também nesta classe que é imprimido o histórico das viagens de um dado utilizador, funcionalidade esta que é invocada pelo próprio no seu respetivo menu.

2.2 *Point2D* e Classe *Client*

Tal como foi referido acima, é nesta classe, *Point2D*, que são guardadas as coordenadas de cada cliente e de cada taxi. Estas coordenadas, duas variáveis, x e y, são do tipo double. Aquando do momento da viagem, são invocados os métodos *distanceTo(Point2D dest)* e *travelTo(Point2D p)*. A função destes métodos é, respetivamente, calcular a distância de um ponto a outro fornecido ao método, e igualar as coordenadas de um ponto ao dado como parâmetro.

Relativamente ao cliente, nós decidimos utilizar uma variável para guardar o valor gasto acumulado por um dado cliente, ao longo das viagens realizadas nos átxis da UMeR. Esse valor é aumentado sempre que um cliente realiza uma viagem, sendo também importante para, mais tarde, calcular o total faturado por um determinado táxi ou pela empresa. Como também queríamos adicionar a funcionalidade *favoritos*, implementamos um dois *TreeMap* para guardar os veículos e motoristas favoritos. Optámos por TreeMap, visto que é de procura rápida, bastando fazer o get de uma determinada key para devolver o veículo ou motorista desejado.

2.3 Classe *Driver*

Relativamente a esta classe, além dos valores da superclasse *Person*, tem guardadas variáveis que definem o fator de confiança de um dado motorista, a sua avaliação média dada pelos clientes, o total de quilómetros acumulados, e o desvio entre o tempo esperado de uma viagem e o tempo atual. Implementámos também um *ArrayList* que serve para calcular a média das avaliações dadas pelos clientes.

```
1 public class Driver extends Person{
2     private double trustFactor;
3     private double actualEvaluation;
4     private double kms;
5     private double timeExceeded;
6     private ArrayList<Double> evaluations;
7     ...
8 }
```

2.4 Classes *Vehicle*, *Car*, *Van*, *MotorBike*

Aqui, também utilizámos hierarquia. *Vehicle* é uma superclasse das classes *Car*, *Van* e *MotorBike*. Decidimos implementar desta forma, visto que um carro, uma carrinha e uma mota apenas vão diferir no número de lugares. Esta classe é simples, apenas guarda os dados básicos de um veículo: matrícula, fator de fiabilidade e velocidade média.

```
1 public abstract class Vehicle implements Serializable{
2     private double avgSpeed;
3     private double factor;
4     private String plate;
5     ...
6 }
```

2.5 Classe *TaxiRide*

As caraterísticas básicas de uma viagem, ou seja, o ponto de início, ponto de destino, o veículo usado, distância percorrida e preço, está tudo armazenado nesta classe. Adicionamos também o email do cliente e do motorista, apenas para, quando necessário, imprimir os dados da viagem.

2.6 Classe *Taxi*

Tal como na vida real as viagens são realizadas num táxi, também na nossa aplicação assim é procedido, isto é, a viagem é realizada na nossa classe *Taxi*. Para isto, guardamos aqui o cliente, o motorista, o veículo, os dados da viagem(TaxiRide), a localização do taxi, um "sinal"que avisa se está ou não ocupado, uma fila de espera, o preço base que cada taxi cobra por quilómetro, e o lucro total desse taxi.

No início, as coordenadas desse taxi são geradas por um random(), sendo alteradas como realizar da viagem.

Um dos métodos mais fundamentais de toda a aplicação está implementado nesta classe. *rideStart* é onde se calcula:

- Distância até ao destino do cliente;
- Tempo esperado de viagem. Este dado é calculado através da velocidade média e da distância calculada anteriormente.
- Tempo de viagem atual, isto é, o verdadeiro tempo que demorou a chegar ao ponto de destino. Este tempo pode ter variações devido às condições meteorológicas, fator de fiabilidade do motorista e também do veículo, e do trânsito.

No fim de cada viagem adicionamos esta tanto ao histórico do cliente como ao do motorista, assim como é feita a cobrança da respetiva viagem ao cliente.

```
1  public class Taxi implements Serializable{
2    private Driver driver;
3    private Vehicle vehicle;
4    private Client client;
5    private TaxiRide trip;
6    private Point2D location;
7    private boolean occupied;
8    private Queue<Client> waitingQ;
9    private double basePrice;
10   private double totalProfit;
11   ...
12 }
```

2.7 Classe *Admin*

Esta classe é restrita aos administradores, ou seja, apenas entra no modo administrador depois de inserir o código correto.

É aqui que se observam quais os 10 clientes mais gastadores, quais os 5 piores motoristas, isto é, os 5 motoristas que mais desvios apresentam entre tempo esperado de viagem e tempo de viagem atual. Além disso, é neste ponto do programa que se registam mais veículos.

Esta classe é necessária dadas as caraterísticas da nossa aplicação, explicas mais abaixo neste relatório.

```
public class Admin implements Serializable{
    private static int code = 1487;
    ...
}
```

2.8 Classe *UMeR*

```
public class UMeR implements Serializable{
  private int userType; // 1 is client; 2 is driver
  private int nVehicles;
  private int nDrivers;
  private TreeMap<String, Client> clients;
  private TreeMap<String, Vehicle> vehicles;
  private TreeMap<String, Driver> drivers;
  private TreeSet<Taxi> taxis;
  private static int driverCode = 611;
  private double totalProfit;
  ...
}
```

A par da *UMeRapp*, são as principais classes do nosso programa. A UMeR é a nossa empresa de táxis, é onde todos os dados estão registados. Para guardar os veículos, motoristas e clientes utilizámos um *TreeMap* para cada uma destas variáveis. Optámos por esta estrutura, devido à fácil procura e simples implementação. Por exemplo, quando queremos procurar um determinado motorista, para iniciar sessão ou até para atualizar os seus dados, basta fazer *get(email)*, em que "email"corresponde ao e-mail desse motorista. Ou então para verificar se existe esse motorista, fazemos *containsKey(email)* e obtemos a resposta eficaz e rapidamente. Além disso, utilizámos um *TreeSet* para guardar os táxis.

É nesta classe que estão os métodos que procuram pelo carro mais próximo, carro mais próximo livre, o mesmo se aplicando às carrinhas e às motas. Os métodos *login* e *startDay*, métodos importantes na execução bem sucedida no programa, encontram-se também nesta classe. *Login* permite ao utilizador iniciar sessão na aplicação. *startDay* é o começar do dia de trabalho de um motorista, ou seja, é neste método que se junta um motorista e um veículo, e se forma um táxi. Logicamente, temos também o método *endDay*, que faz o oposto: remove o táxi do *TreeSet*, tornando os motoristas disponveis novamente.

2.9 Classe UMeRapp

Toda nossa aplicação se desenrola aqui. Através da classe *Menu* fornecida pela equipa docente, e à medida que foi avançando o projeto e foi surgindo a necessidade de adionar mais um menu, uitlizamos os seguintes:

- *homeMenu* - menu inicial;
- *clientMenu* - menu do cliente;
- *driverMenu* - menu do motorista;
- *signUpMenu* - menu para registo de um utilizador;
- *adminMenu* - menu para administradores;
- *callingTaxiMenu* - menu para chamar um táxi;
- *favoriteMenu* - menu dos favoritos de um cliente;
- *signUpVehicleMenu* - menu para registo de um veículo;
- *specificVehicleMenu* - menu para o cliente chamar um veículo específico;
- *driverSubMenu* - menu de trabalho do motorista;
- *profitMenu* - menu para se observar o lucro de uma empresa ou de um determinado táxi;

A justificação do porquê de esta classe ter muitas linhas de código é o facto de haver muitas variáveis, isto é, muita coisa que pode acontecer. Para que seja bem executado, temos muitas condições ao logo do código, para que se algo corra mal, avisar o utilizador do que ocorreu. Por exemplo, um motorista que tenha iniado trabalho, caso se engane e selecione novamente a opção "Iniciar Trabalho", poderia ocorrer o "bug"de haver um motorista associado a dois veículos, o que não é possível. Portanto, para este exemplo, colocámos uma "flag"a verificar se ja iniciou sessão: caso tenha, avisa o motorista de que já tem o trabalho inicado, caso contrário, é permitido o início do mesmo.

Todo o funcionamento da aplicação está pormenorizadamente explicado na secção mais abaixo *Funcionamento da UMeR*.

```
1 public class UMeRapp implements Serializable {
2   private UMeR taxiCompany;
3   private static Admin admin;
4   private Client client;
5   private Driver driver;
6   private static int userType;
7   private static Menu homeMenu, clientMenu, driverMenu, signUpMenu,
        callingTaxiMenu, favoriteMenu, signUpVehicleMenu, adminMenu,
        specificVehicleMenu, driverSubMenu, profitMenu;
8     ...
9 }
```

2.10 Exceptions

Para os casos em que não haja, por exemplo, taxis no *TreeSet* correspondente aos taxis da nossa UMeR, temos que ter algo que nos permita resolver esse problema. As exceptions são fundamentais na execução de um programa. Neste caso, implementámos as seguintes:

- *NoClientsException*;
- *NoDriversException*;
- *NoEvaluationsException*;
- *NoTaxisException*;
- *NoVehiclesException*;
- *UserDoesNotExistsException*;
- *UserExistsException*;
- *VehiclesExistsException*;

3 Funcionamento da UMeR

Após idealizarmos sobre este projeto, e termos pensado como iríamos implementar o nosso programa, chegamos a um só pensamento: fazer com que a UMeR tivesse "todas"as funcionalidades, ou seja, lendo o enunciado deste trabalho e observando as condições que seriam necessárias para uma boa avaliação, optámos por implementar diversas funcionalidades na aplicação. Sendo assim, passemos então à explicação de como funciona a nossa UMeR.

Esta aplicação está programada para que o número de motoristas seja igual ou menor ao número veículos, para que não se dê o caso de haver motoristas sem trabalhar. É devido a isto que criámos a classe *Admin*, que serve fundamentalmente para o registo de viaturas na empresa.

Relativamente aos pedidos de táxis, primeiramente dizer que todos eles apresentam uma fila de espera, o que permite a um cliente que queira muito aquele táxi específico ser inserido nessa respetiva fila. Quanto ao tipo de pedidos que o cliente pode efetuar, nós implementamos da seguinte forma:

- O cliente pode pedir um táxi específico: um carro, carrinha ou mota. Para tal, implementámos três métodos, um que procura por um carro, outro por uma carrinha e outro por uma mota, sendo eles o *specificCar()*, *specificVan()* e *specificMotorBike()*, respetivamente. Os três métodos fazem exatamente a mesma coisa, mas para veículos diferentes. Para fazer a distinção, usámos *instanceof*.

  Mais concretamente, o que estes métodos fazem é procurar pelo repetivo veículo mais próximo. Se esse táxi mais próximo estiver ocupado, questionamos o cliente do que pretende fazer: caso o cliente queira ir para a fila de espera, inserimo-lo, ficando a aguardar; caso contrário, procuramos pelo mesmo tipo de veículo mas, desta feita, livre.

- O cliente pode pedir o táxi mais próximo. O que fazemos, através do método *closestTaxi()* na classe *UMeRapp*, é procurar pelo táxi mais próximo. Caso esse táxi esteja ocupado, perguntamos ao utilizador se deseja ir para a fila de espera desse mesmo táxi. Se "Sim", então inserimos, se "Não"chamamos o método *getClosestFreeTaxi()* que procura pelo táxi livre mais próximo.

- O cliente pode pedir o táxi que esteja a ser dirigido por um motorista específico. Para isto, primeiramente imprimimos os motoristas a trabalhar no momento. De seguida, pedimos ao cliente que escreva o e-mail do motorista pretendido. Para verificar se o e-mail escrito é correto, temos um método chamado *writeEmail()*, que retorna sempre um e-mail válido. Uma vez com o e-mail requerido por parte do cliente, o único passo a fazer é procrar pelo taxi dirigido por esse motorista e chamá-lo(uma vez mais com as questões sobre o estar ocupado ou não).

É a partir destas ideias base que todo o programa decorre:

3.1 Registo de Motoristas

Todos os motoristas que se queiram registar possuem o mesmo código (611), código este que serve como um tipo de teste para que apenas se registem motoristas autorizados a tal. Como também foi explicado acima, além do código, é necessário que haja mais ou igual número de veículos comparando com o número de motoristas, para que se possa registar.

Isto assegura que não haja um motorista associado a vários veículos, e que apenas se registam motoristas autorizados. Abaixo está uma ilustração do que acontece quando o número de veículos iguala o nº de motoristas.

Exemplo para quando nº veiculos = nº motoristas

```
*** Bem-vindo à UMeR ***
1 - Iniciar Sessão
2 - Registar utilizador
3 - Modo Administrador
0 - Sair
Opção: 2

*** Registo ***
1 - Sou cliente
2 - Sou motorista
0 - Sair
Opção: 2
Email:
x@y
Nome: Pizzi
Password: 36
Morada: Avenida dos aliados
Data de nascimento (dd-MM-yyyy): 22-11-1904
Digite o código de trabalhador:
611
Neste momento não há veículos disponíveis. Aguarde notificação.
```

3.2 Registo de Clientes

Todos os clientes que se quiserem registar na UMeR têm liberdade para isso, ou seja, regista-se como cliente quem quiser, apenas o e-mail que inserir tem que ser diferente daqueles já existentes.

Exemplo de um registo de um cliente

```
*** Registo ***
1 - Sou cliente
2 - Sou motorista
0 - Sair
Opção: 1
Email:
ola@email.com
Nome: Joaquim
Password: 1234
Morada: xyz
Data de nascimento (dd-MM-yyyy): 11-11-11
Registo efetuado com sucesso!
```

3.3 Modo Administrador

Tal como foi dito antes, este modo é para observar os 10 motoristas que apresentam mais desvios entre tempo esperado de viagem e tempo atual; os 5 clientes que mais gastam na UMeR; e também para registo de viaturas. Como é óbvio, há um código para entrar no modo administrador, sendo este 1487. Isto é necessário para quando haja um motorista "autorizado"a se registrar, e não haja veículos disponíveis.

```
*** Bem-vindo à UMeR ***
1 - Iniciar Sessão
2 - Registar utilizador
3 - Modo Administrador
0 - Sair
Opção: 3
Digite o código:
1487

*** Administrador ***
1 - Registar viatura
2 - 10 clientes mais gastadores
3 - 5 piores motoristas
4 - Lucro Total
0 - Sair
Opção:
```

(a) Menu Admin

```
*** Administrador ***
1 - Registar viatura
2 - 10 clientes mais gastadores
3 - 5 piores motoristas
4 - Lucro Total
0 - Sair
Opção: 2
Helena da POnte. O email é client13@email.com
Gonçalo Guedes Paciencia. O email é client5@email.com
Patricia Barros. O email é client6@email.com
Leandro Coelho. O email é client8@email.com
Vanessa Do Canto. O email é client4@email.com
Bruno Marrocos. O email é client1@email.com
Joana Barroso. O email é client11@email.com
Carolina Oliveira. O email é client3@email.com
Maria Tavares. O email é client10@email.com
Bruno Oliveira. O email é client14@email.com
```

(b) top 10 clientes

Exemplo Modo Administrador

5 piores motoristas

```
*** Administrador ***
1 - Registar viatura
2 - 10 clientes mais gastadores
3 - 5 piores motoristas
4 - Lucro Total
0 - Sair
Opção: 3
Fernando Santos. O email é driver11@email.com
José Esquina. O email é driver3@email.com
Tiago Atletico. O email é driver15@email.com
Sandro Corte. O email é driver7@email.com
Esteves. O email é driver13@email.com
```

3.4 Menu Cliente

Como se pode ver na imagem abaixo, o cliente tem várias opções à escolha.

- *Procurar Táxis por tipo* procura táxis por tipo, isto é, por carro, carrinha ou mota, imprimindo-os, estejam ou não ocupados.
- *Procurar Táxis disponíveis* imprime todos os táxis disponíveis, ou seja, livres.
- *Mostrar todos os Táxis* imprime todos os táxis.
- *Pedir Táxi* é a função principal do menu do cliente, tendo em conta que é o propósito da aplicação. Esta opção permite ao cliente fazer o pedido de táxi. Quando o cliente escolhe esta opção, são-lhe pedidas as suas coordenadas atuais, assim como as de destino. Após isto, dá-mos a escolher ao cliente três opções: pedir veículo específico, pedir táxi mais próximo, ou chamar por motorista específico, opções estas já explicadas acima.
- *Mostrar histórico de viagens* imprime o histórico de viagens do cliente.
- *Obter lista de favoritos* imprime os táxis favoritos de um cliente.
- *Mostrar Perfil* imprime os dados do cliente.

A acrescentar a estas funcionalidades temos os favoritos de um cliente. No fim de cada viagem, o cliente é questionado se deseja ou não adicionar o taxi aos favoritos, até dar-lhe uma nota.

```
*** Bem-vindo à UMeR ***
1 - Iniciar Sessão
2 - Registar utilizador
3 - Modo Administrador
0 - Sair
Opção: 1
Email: client1@email.com
Password: pass1

*** Cliente ***
1 - Procurar táxis por tipo
2 - Procurar táxis disponiveis
3 - Mostrar todos os Táxis
4 - Pedir Táxi
5 - Mostrar histórico de viagens
6 - Obter lista de favoritos
7 - Mostrar Perfil
0 - Sair
Opção:
```

(a) Menu CLiente

```
*** Chamar Taxi ***
1 - Pedir veículo específico
2 - Pedir taxi mais próximo
3 - Chamar por motorista específico
0 - Sair
Opção: 2
Taxi a caminho!
Tempo esperado de viagem: 9min
Tempo atual de viagem: 0min
Viagem iniciada em x:1.1 y:3.2 e terminada em x:5.7 y:8.3
 O motorista foi: driver14@email.com
 O cliente foi: client1@email.com
 Distância total: 9.48314293892062
Deseja atribuir uma nota ao motorista?
Sim
Avaliação:
10
Pretende adicionar o Taxi aos favoritos? [Sim/Não]
Sim

*** Favoritos ***
1 - Adicionar motorista
2 - Adicionar veículo
0 - Sair
Opção: 1
Motorista adicionado aos favoritos!
```

(b) Pedir um Taxi

Exemplo Modo Administrador

3.5 Menu Motorista

Assim como o cliente tem o seu menu, também o motorista tem o seu. Neste caso, existem as opções "Começar trabalho", "Mostrar histórico de viagens", "Terminar trabalho", e "Mostrar perfil". Em relação a esta parte do programa, há uma nota importante: tal como foi dito anteriormente, um taxi é formado juntando um motorista e um veículo, no método *startDay()* da classe *UMeR*. No entanto, não é apenas isto que acontece. Quando um motorista escolhe a opção "Começar trabalho", é invocado este método. É aqui que "se dirije a um veículo", ou seja, que se junta a um veículo livre. Logo, se o motorista selecionasse esta opção mais do que uma vez, o que iria acontecer é que um motorista iria estar associado a vários veículos, o que não pode acontecer. Então, para solucionar este problema, sempre que é invocado o método *startDay()*, primeiro verificamos se esse motorista já está associado a um táxi(através do treeSet dos taxis conseguimos aceder ao taxi desejado, logo ao motorista a conduzi-lo), e se já estiver enviamos uma mensagem a informar que já iniciou trabalho antes.

Menu motorista

```
*** Motorista ***
1 - Começar trabalho
2 - Mostrar histórico de viagens
3 - Terminar trabalho
4 - Mostrar Perfil
0 - Sair
Opção:
```

Menu motorista

```
*** Motorista ***
1 - Começar trabalho
2 - Mostrar histórico de viagens
3 - Terminar trabalho
4 - Mostrar Perfil
0 - Sair
Opção: 4
Sem avaliações
Esteves
driver13@email.com
Rua do Calcanhar de Aquiles nº1024
22-07-83
```

4 Conclusão

Em suma, este trabalho serviu para nos enriquecer numa programação diferente daquelas que até ao momento tinhamos encontrado. Houve novos problemas associados à mudança de paradigma e a sua resolução serviu para alargar o conhecimento que já possuíamos.

Além disso o enunciado requeria uma espécie de aplicação que de alguma forma replicasse a *UBeR*, logo grande parte do foco de desenvolvimento foi dado à interface e às funcionalidades que os diferentes tipos de utilizadores poderiam requerir, e não tanto na implentação e otimização na procura das soluções mais eficientes.

Acreditamos, então, que o resultado final foi o desejado e alinhado com os nossos objetivos, servindo, de uma forma muito produtiva e "com as mãos na massa", de aprender sobre *Java* e a programação orientada a objetos.