Universidade do Minho

Departamento de Informática

Programação Orientada aos Objetos

Relatório Projeto POO - UMeR

Grupo 18

Resumo

Neste relatório será abordado o desenvolvimento do projeto de Programação Orientada aos Objetos sobre uma empresa de táxis.

3 de Junho de 2017

Gil Cunha a77249

Nuno Faria a79742

Tânia Silva a76945

Conteúdo

1 Introdução

Neste relatório iremos abordar o desenvolvimento do projeto sobre a empresa de Táxis UMeR. Falaremos da forma como estruturamos os nossos dados, das classes que usamos e de que forma estas interagem umas com as outras, e de como implementamos a interface gráfica do programa.

2 Objetivo e Enunciado

2.1 Objectivo

O principal objectivo deste trabalho consiste no desenvolvimento de um programa que implemente os conceitos da programação por objetos. Deverá fazer uso das boas práticas da programação, sendo estas a reutilização de código, uma boa estrutura, implementação de estruturas de dados eficientes e garantir o encapsulamento de dados.

2.2 Enunciando

O projeto consiste na criação de uma empresa de táxis. É pedido a implementação de utilizadores, veículos diversos e sub-empresas que façam a sua própria gestão.

2.2.1 Utilizadores

O programa deverá implementar dois tipos de utilizadores, condutores e clientes, sendo que cada um tem requisitos específicos.

2.2.2 Viagens

Cada cliente poderá efetuar viagens, indicando para isso a posição onde se encontra de momento e onde pretende chegar. Será ainda dada-lhe a possibilidade de escolher um condutor específico ou o mais próximo. Caso escolha um que não esteja livre, será colocado na sua fila de espera. No final poderá dar-lhe a classificação que queira (caso queira), recebendo ainda a informação sobre a viagem (Fig-1)[1].

Figura 1: Exemplo de um folheto de uma viagem

2.2.3 Informação

Outro aspeto importante é a recolha de informação. Cada cliente, condutor e empresa deverão ter acesso ao seus registos de viagens e de estatísticas como o número de `kms` percorridos ou o tempo total em viagem. Terá que ser ainda possível a aceder a estatísticas globais (como por exemplo, top 10 clientes que gastaram mais dinheiro).

[1] Terá que conter ainda mais informação para além da apresentada na figura

2.2.4 Empresas

É também pedida a implementação de sub empresas que contenham os seus clientes e veículos. Ao contrário dos condutores normais, os condutores de uma empresa não tem o seu próprio carro, cabendo assim à gestão da empresa escolher que condutor irá conduzir numa determinada viagem.

2.2.5 Interface

Para poder fazer-se uso do programa, é pedido também a implementação de uma interface que permita registar utilizadores, veículos e empresas, fazer *login* numa conta já existente, efetuar viagens e aceder à informação pessoal.

3 Classes

Na figura seguinte podemos ver graficamente a estrutura geral do projeto.

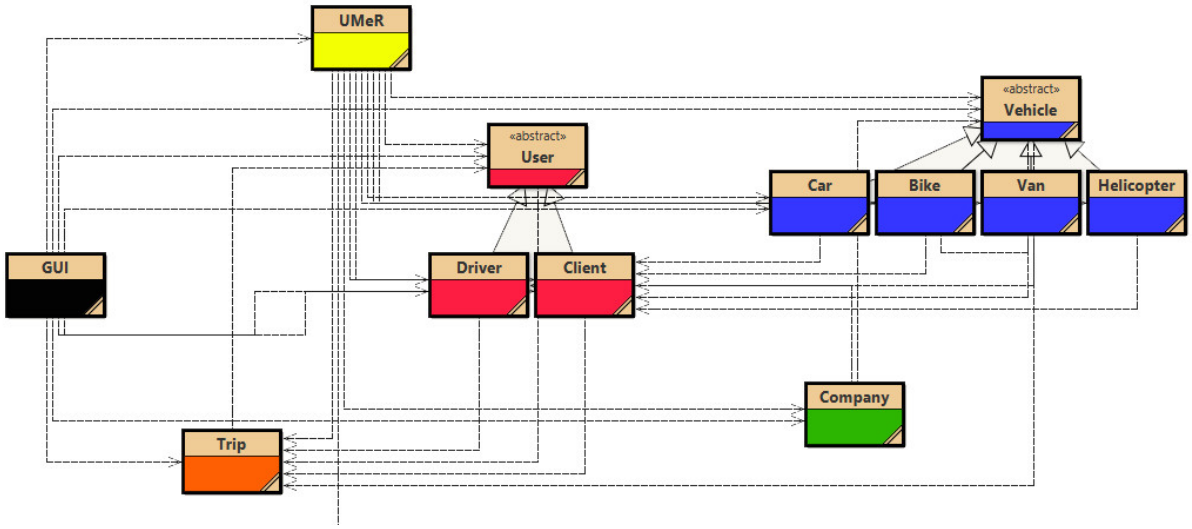

Figura 2: Hierarquia e dependências de classes

- Vermelho - Utilizadores
- Azul - Veículos
- Laranja - Viagem
- Verde - Empresa
- Amarelo - Classe principal
- Preto - Interface

Faltam ainda algumas classes auxiliares (como comparadores) que não estão presentes na figura. Podemos ver que `Driver` e `Client` são subclasses de `User`, e que `Vehicle` tem diversas subclasses que o especificam.

3.1 User

Esta será a classe responsável pela implementação do utilizador. Tanto clientes como condutores terão que guardar alguma informação que têm em comum, como nome ou email, tendo-se assim as variáveis de instância:

```
private String email; //Email
private String name; //Nome
private String password; //Palavra pass
private String address; //Morada
private LocalDate birthday; //Data de nascimento
private double totalDistance; //Distancia total em viagem
private ArrayList<Trip> trips; //Lista de viagens
private double money; //Dinheiro ganho/gasto
```

Esta classe não terá muitos métodos específicos, sendo esses deixados para `Client` e `Driver`.

3.1.1 Client

Classe que fará a especificação de um utilizador em cliente. Cada cliente terá que ter a sua posição atual, ao contrário do condutor, e terá também uma variável que dirá a matrícula do carro onde este está em fila de espera (`null` caso não esteja em nenhuma).

```
private Point2D.Double position; //Posicao atual
private int points; //Pontos
private String queue; //Matricula do carro em espera
```

A parte dos pontos serve apenas para indicar o quão este cliente já viajou e quanto pagou, sendo calculado da seguinte forma:
`points += trip.getPrice() / 2 + trip.distance() / 4;`

3.1.2 Driver

Classe que fará a especificação de um utilizador em condutor. Esta classe será mais complexa que `Client`, visto que todo o condutor terá que ter informação sobre o seu grau de condução, a classificação, se pertence a uma empresa, se tem carro, etc, tendo-se por isso as seguintes variáveis:

```
private double rating; //Classificao total
private boolean availability; //Disponibilidade
private double timeCompliance; //Grau de cumprimento de tempo
private int numberOfReviews; //Numero de classificacoes
private int exp; //Experiencia
private String vehicle; //Matricula do veiculo
private String company; //Empresa para qual trabalha
private double deviation; //Desvio total
```

A variável `exp` é análoga à `points` do cliente, à excepção de que esta irá influenciar a rapidez com que o condutor chega ao seu destino. É calculada assim:

```
this.exp += (trip.distance()+1)/2;
if (rating > 2)
        this.exp += 3 * (rating/5);
else this.exp -= 4-rating;
```

Ou seja, é influenciada não só pela distância percorrida mas também pela classificação obtida, perdendo pontos caso tenha uma nota negativa (1 ou 2).

A variável `timeCompliance` indica o grau de cumprimento dos horários.

As *strings* `vehicle` e `company` indicam a matrícula do seu carro e o nome da empresa onde trabalha, respetivamente, sendo que se uma não é `null`, a outra terá que o ser (visto que um condutor que pertença a uma empresa não tem carro e vice-versa).

O *double* `deviation` serve para guardar o desvio entre o dinheiro estimado da viagem e o dinheiro real, servindo mais tarde para efeitos de estatística.

Caso pretendermos implementar um novo tipo de condutor, apenas criamos uma *subclass* de `driver`, onde especificamos as novas variáveis e métodos pretendidos (como por exemplo, um condutor de luxo, que terá uma variável que indica o *grau de luxo*, podendo depois ser o único que pode conduzir determinados tipos de carro numa empresa).

3.2 Vehicle

Esta será a classe que irá implementar todos as variáveis e métodos dos veículos. Cada veículo terá que ter uma posição, a sua estrutura (número de lugares, velocidade e o preço/km), o seu dono, etc, tendo como variáveis:

```
private String licencePlate; //Matricula
private double speed; //Velocidade media
private double price; //Preco por km
private double reliable; //Grau de viabilidade
private boolean available; //Indica se esta disponivel
private int seats; //Numero de lugares
private Point2D.Double position; //Posicao atual
private LinkedList<String> queue; //Lista de espera
//Informacao sobre quem esta na fila de espera
private HashMap<String, ArrayList<Point2D.Double>> queueInfo;
private ArrayList<Trip> trips; //Lista de viagens
private String owner; //Dono do veiculo
```

As variáveis `queue` e `queueInfo` irão guardar a informação da fila de espera do veículo, sendo que a primeira gurda apenas o email do cliente que o requisitou e a segunda o início e o fim da viagem.

A *string* `owner` indicará a quem pertence o veículo, podendo ser um condutor ou uma empresa.

3.2.1 Veículos específicos

A classe `Vehicle`, tal como `User`, é abstrata, sendo que não pode ser instanciada. O processo de implementação de veículos concretos é através de subclasses de `Vehicle`. Como é assumido que todo o *carro* tem a mesma velocidade, lugares e preço, teremos apenas uma classe que representará todos os tipos de *carro* (sendo o mesmo com, por exemplo, *motas*).

Um exemplo de um veículo é a subclass `Car`, tendo o seguinte construtor:

```
(...)
this.setSpeed(65); //65 km/h
this.setPrice(1.5); //1.5 euros/km
this.setSeats(4); //4 lugares
```

Neste momento, o nosso projeto tem quatro tipos de veículos, sendo eles `Car`, `Bike`, `Van` e `Helicopter`. Caso pretendamos implementar um novo tipo de veículo, simplesmente criamos uma subclasse de `Vehicle` e colocamos lá os construtores (vazio, parametrizado e cópia) em que indicamos o novo valor das 3 variáveis acima (teremos ainda que implementar o `clone()`). O novo veículo também pode ter outro tipo de variáveis se for necessário (continuando o exemplo de `driver`, podemos ter uma variável que indica se é um carro de luxo, sendo possível ser apenas conduzido por condutores de luxo).

3.3 `Trip`

Classe que conterá toda a informação sobre uma viagem. Precisamos não só de guardar variáveis como o tempo da viagem ou o seu preço, mas também dos valores estimados pelo condutor antes da viagem.

```
private int id; //Numero de identificacao
private Point2D.Double start; //Posicao inicial
private Point2D.Double end; //Posicao final
private double time; //Tempo da viagem
private double price; //Preco
private LocalDate date; //Data
private String licencePlate; //Matricula do veiculo
private String driver; //Email do condutor
private String client; //Email do cliente
private int rating; //Classificacao
private Point2D.Double taxiPos; //Posicao inical do veiculo
private double estimatedTimeToDest; //T. estimado de viagem
private double estimatedTimeToClient; //T. est. ate cliente
private double realTimeToClient; //Tempo real ate cliente
private double estimatedPrice; //Preco estimado
```

A identificação será dada a partir da classe principal, sendo $id = id_{anterior} + 1$.

Além dos tempos e preços, guarda-se ainda distância total, posições inicial e final, o email dos utilizadores, a matrícula do veículo, a classificação dada e a data.

Todos os tempos (reais e estimados) e preços serão calculados na classe principal, os quais iremos falar mais à frente.

3.4 Company

Classe que representará uma sub-empresa de UMeR. Terá que fazer a sua própria gestão, por isso terá que guardar todos os condutores e veículos pertencentes, tendo como variáveis:

```
private String name; //Nome
private String password; //Palavra-pass
private HashMap<String, Driver> drivers; //Map de condutores
private HashMap<String, Vehicle> vehicles; //Map de veiculos
private ArrayList<Trip> trips; //Lista de viagens
private double moneyGenerated; //Dinheiro gerado
private int points; //Pontos
```

Analogamente a User, terá uma variável de dinheiro total (soma do gerado por todos os condutores), uma de viagens totais e outra de pontuação, que será dada por:
`points += trip.getPrice() - trip.distance();`
o que significa que quanto mais dinheiro os seus condutores geraram por km, melhor.

Quando é requisitada uma viagem com algum veículo da empresa, terá que ser escolhido um dos condutores. Existe por isso um método que irá, de todos os condutores disponíveis, escolher um aleatório. Terá ainda um método que, caso um cliente escolha especificamente esta empresa, irá selecionar o veículo mais perto deste.

3.5 UMeR

É a classe principal do projeto, sendo responsável pela gestão geral da empresa, desde registar um utilizador, procurar o veículo mais próximo, fazer uma viagem, etc. Guardará a informação necessária da seguinte forma:

```
private HashMap<String, Driver> driversP; //Cond. privados
private HashMap<String, Driver> allDrivers; //Todos cond.
private HashMap<String, Client> clients; //Clientes
private HashMap<String, Vehicle> vehiclesP; //V. privados
private HashMap<String, Vehicle> allVehicles; //Todos v.
private HashMap<String, Company> companies; //Empresas
private ArrayList<Trip> trips; //Lista de viagens totais
private double moneyGenerated; //Dinheiro total gerado
private double totalDistance; //Distancia total das viagens
private double totalTime; //Tempo total das viagens
private int tripID; //ID da proxima viagem
private int weather; //Meteorologia
```

Serão separados condutores e carros privados (sendo que não guardam uma cópia, mas sim apenas uma referência), para facilitar a gestão de novas viagens.

Tem-se ainda uma variável de meteorologia, que irá influenciar o tempo real, cujo o cálculo sera falado mais à frente.

3.6 Outras classes

O projeto tem ainda outras classe, sendo a interface (que iremos abordar mais à frente), comparadores e uma classe de distribuição de probabilidades simples.

3.6.1 `CustomProbabilisticDistribution`

Classe que irá gerar um valor de acordo com probabilidades.

```
1 private int size;
2 private int values[];
3 private int currentPos;
```

Para usar esta classe, depois de criarmos um novo objeto (a partir do construtor vazio) iremos invocar o método `addValues(int value, double probability)`, ao qual passamos o valor que queremos colocar no contradomínio da função e a probabilidade desse mesmo valor sair. O que será feito é adicionar valores ao *array* `values` de acordo com a probabilidade de serem escolhidos. Por ultimo, quando for pretendido um valor, simplesmente invoca-se `pickNumber()`, que erá escolher um índice aleatório e retornar o conteúdo dessa posição.

A classe será usada para auxiliar o cálculo do tempo real das viagens.

4 Estruturas de Dados

A maior parte dos dados é guardado em `HashMaps`. Utilizadores, veículos e empresas são todos colocados nessas estrutura visto que permite o endereçamento direto, sendo por isso mais eficientes que outras alternativas.

Já as viagens serão guardadas num `ArrayList`, visto que não será necessário aceder a nenhuma viagem diretamente (as únicas vezes que iremos aceder a uma lista será quando se vai imprimir para no ecrã ou quando se pretende contar o dinheiro entre datas), preferenciando-se a ordem de chegada.

A fila de espera dum veículo será guardada numa `LinkedList`, visto que contem métodos como `addlast()`.

Para gestão de memória, entre as classes do projeto (à excepção de `GUI`) não serão guardadas cópias se não forem necessárias, tornando a execução do programa mais rápida, poupando assim recurso da máquina onde está a correr.

ArrayList	**HashMap**	**LinkedList**
Viagens	Utilizadores	Filas de espera
QueueInfo (pontos)	Veículos	
	Empresas	
	QueueInfo	

Tabela 1: De que forma são guardados os dados do projeto

5 Gestão de UMeR

Como já vimos, a classe UMeR será responsável pela gestão geral do projeto. Terá que se encarregar de registar utilizadores, veículos, criar novas viagens, entre outras funções que vamos falar agora.

5.1 Registo de utilizadores/veículos

Para registar um utilizador, teremos que passar para o método `registerUser(User u, String company)` o objeto do utilizador e a empresa onde trabalha. Se for um cliente ou trabalhador privado, `company` será igual a `null`.

O método irá certificar-se que não existe mais nenhum utilizador com o mesmo nome. Caso for um condutor de uma determinada empresa, além de colocar a cópia do objeto em `allDrivers`, irá também inserir a referencia dessa cópia na empresa à qual pertence (caso seja privado coloca a referencia em `driversP`). Já quando é um cliente, é colocado em `clients`.

O processo de registo de veículos funciona de forma semelhante, sendo que este tem os métodos de registo privado e empresarial separados.

Este métodos devolverão `true` caso o registo seja efetuado com sucesso, ou `false` caso contrário.

5.2 Viagens

Para realizar uma viagem, teremos que fornecer os objetos de utilizadores e veículo e o ponto do destino.

```
newTrip(Client c, Driver d, Vehicle v, Point2D.Double dest)
```

Caso seja um condutor específico e este esteja ocupado, o cliente é colocado em fila de espera. Caso contrário, iremos calcular todos os tempos e preços necessários, tanto esperados como reais. Este classe assegurar-se-á ainda por realizar todas as viagens da lista de espera dum condutor, através dum `while (existem clientes) -> fazer viagem`.

Se o cliente pedir antes o condutor mais próximo, será feito a mesma coisa à excepção do veículo, que será escolhido pela classe (se não existir nenhum condutor mais próximo, não fará a viagem nem colocará em fila de espera).

5.2.1 Cálculo dos tempos

O cálculo do tempo esperado é feito a partir apenas da distância até ao destino e da velocidade do carro. À divisão destes dois iremos multiplicar por 1.2, para este não corresponda ao valor ótimo (visto que assim um condutor chegaria sempre depois do tempo estimado), tendo-se: `eta = start.distance(end)/ vehicleSpeed * 1.2;`

Já o cálculo do tempo real é mais complicado.

1. Calcularemos o **trânsito** existente num raio em torno do carro, raio esse que será metade da distância até ao destino. Este calculo é feito contando o número de carros nessa área. A este resultado iremos multiplicar por um número de 1 a 100, calculado aleatoriamente (representado os outros carros que não pertencem a UMeR). De seguida dividimos novamente pelo raio, sendo esse valor multiplicado por um valor absoluto da distribuição normal (Gaussiana) (Fig-3) com média 0 e desvio padrão 1 (ou seja, existe cerca de 70% de probabilidade do valor estar entre -1 e 1). Por fim, dividimos o resultado final por 100.

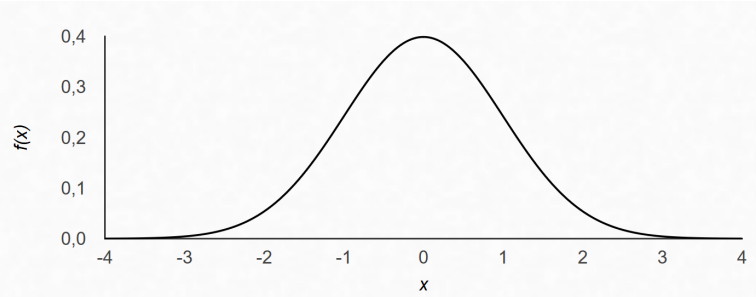

Figura 3: Distribuição normal para $\mu = 0, \sigma = 1$

Com isto obtemos, na maior parte das vezes, um valor adequado para servir de multiplicador com o tempo ótimo. Na rara eventualidade deste valor ser muito grande, será limitado a `0.4`, através do `Math.min`.

2. A partir do **grau de cumprimento** de tempo do condutor e da nossa distribuição de probabilidades `CustomProbabilisticDistribution`, iremos obter um valor 0 ou 1 (por exemplo, se o condutor tiver um grau de cumprimento de 64, existe 64% de probabilidade de ser 0, e 100-64=36 de ser 1). De seguida, multiplicamos esse valor por um número aleatório no intervalo `[0.1,0.2]`, significando isto que no pior caso irá ser acrescentado 0.2 ao multiplicador.

   Este cálculo será feito de forma análoga para a **viabilidade** do carro e a **experiência** do condutor (que é igual à distância a dividir pela experiência (pontos) deste)

3. Os fatores **meteorológicos** também terão impacto no tempo final. Será calculado um número no intervalo `[0,0.2]`, multiplicado-o pela variável `weather` (inteiro de 1 a 5) e divido por 5. Assim, no pior dos casos, teremos tempo 5 e multiplicador 0.2, o que resulta em 0.2 (igual aos fatores anteriores).

4. Por ultimo, somamos estes valores todos a 0.8, multiplicando-se de seguida com o tempo ótimo (= distância/velocidade).

Com isto tudo, no melhor dos casos, temos que o tempo real será 80% do tempo ótimo. Já no pior dos casos, será o dobro do tempo ótimo (multiplicador = 2.00).

5.2.2 Cálculo de preços

O cálculo do preço estimado será apenas multiplicar a distância pelo preço por km. Para o preço real temos dois casos: se a diferença de tempos reais e estimados de `táxi->cliente + tempo cliente->destino` for $< 25\%$, o preço final será `(%diff+1) * preço estimado`; se essa diferença for $> 25\%$, o preço real será o `preço estimado - distancia/2 * %diff`, ou seja, por exemplo, um condutor que atrase `30%` numa viajem de curta duração será menos penalizado que outro que se atrase na mesma percentagem mas numa viagem maior.

5.3 Guardar e carregar estado

Como é necessário assegurar a permanência de dados, esta classe terá um método que irá guardar o estado atual (num ficheiro binário) e outro que irá carregar o mesmo.

6 Interface (Funcionalidades/Manual)

Agora que já temos a parte essencial do projeto, falta ter forma de utilizar o mesmo. Para isso, desenvolvemos uma interface gráfica onde poderemos criar um utilizador, fazer uma viagem, etc (tínhamos a alternativa de fazer uma interface através do terminal, mas optamos por esta por proporcionar uma melhor experiência ao utilizador final).

6.1 Menu

Este é o menu do programa. É possível registar um utilizador (cliente ou condutor) ou uma empresa. O registo de veículos terá que ser feito ou ao registar um condutor ou no menu de uma empresa (para ambos os casos o utilizador terá que escolher o tipo do veículo e a sua condição).

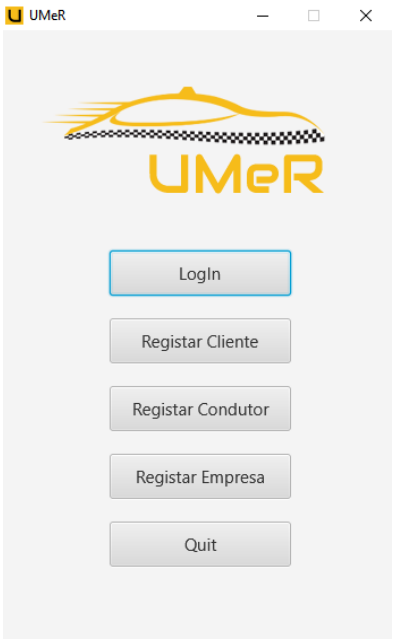

Figura 4: Menu principal

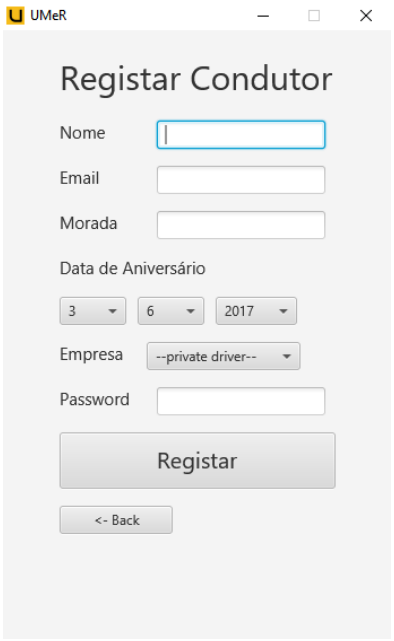

Figura 5: Registo condutor

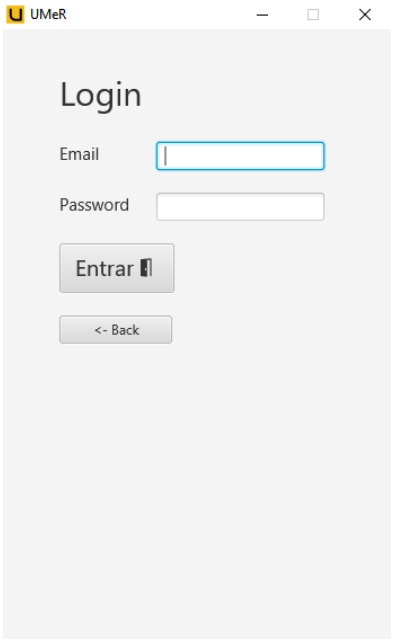

Figura 6: Início de sessão

Como podemos ver na Figura-5, quando queremos criar um condutor, teremos que preencher todos os dados necessários (o registo do cliente e da empresa funciona de forma semelhante). Caso os dados estejam incorretos ou já exista um utilizador com o mesmo email, não permitiremos o registo.

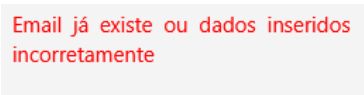

Figura 7: Mensagem de erro

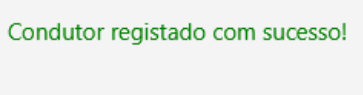

Figura 8: Mensagem de sucesso

6.2 Menu de utilizadores/empresas

O menu de ambos utilizadores e empresas tem dois separadores em comum: o que indica a informação geral (como o nome, dinheiro gasto/ganho, km percorridos, etc) e o que indica a informação das viagem feitas, separadas por dia.

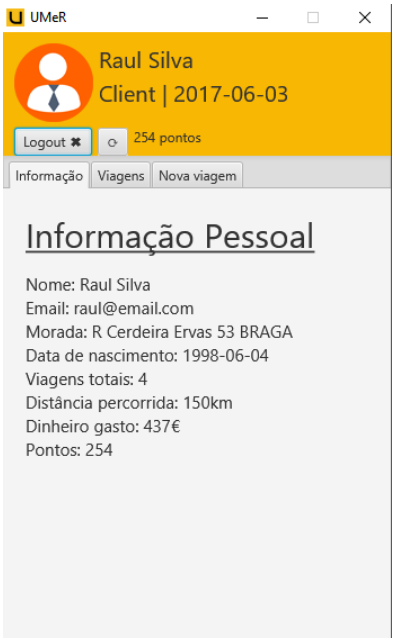

Figura 9: Menu cliente

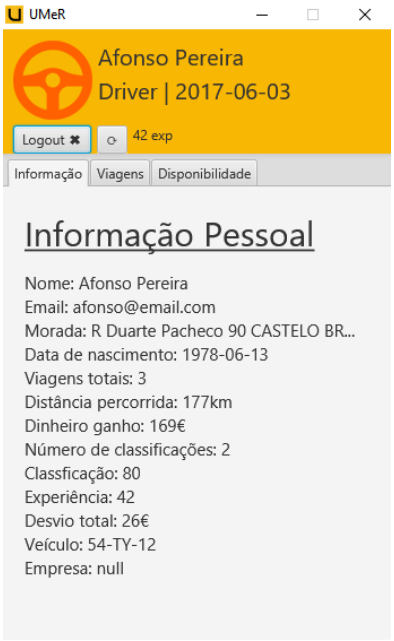

Figura 10: Menu condutor

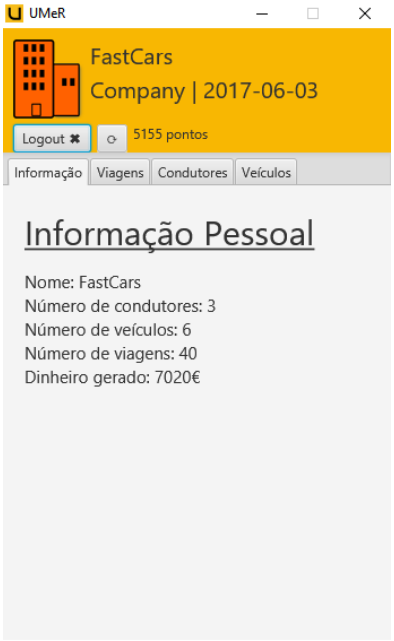

Figura 11: Menu empresa

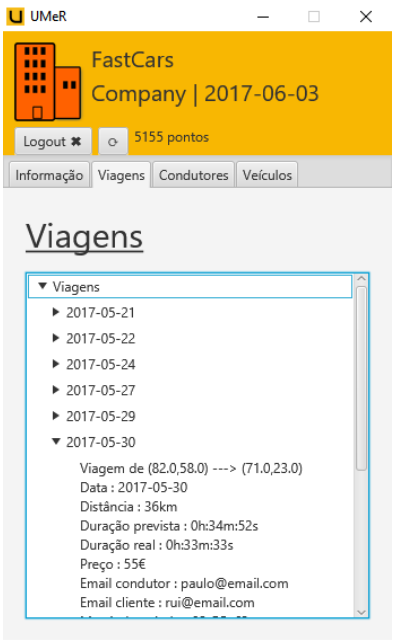

Figura 12: Separador de viagens (todos)

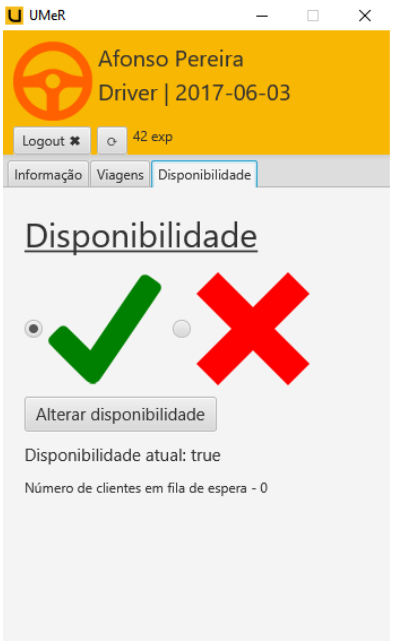

Figura 13: Separador de disponibilidade (condutores)

Figura 14: Separador de veículos (empresas)

O cliente irá ter um separador onde poderá efetuar uma nova viagem. O condutor terá um onde poderá escolher se está a trabalhar ou não. A empresa terá dois adicionais, um que tem a lista de condutores e outro dos veículos, sendo que neste último pode registar um novo (registo semelhante a um utilizador).

6.3 Nova viagem

Quando um cliente quiser fazer uma nova viagem, terá que indicar a sua posição inicial (onde se encontra) e até onde quer chegar. Pode escolher ainda um condutor específico (privado), uma empresa ou o veículo mais próximo. Caso a viagem seja realizada com sucesso, irá aparecer uma "caixa"com a nova viagem e a opção para classificar o condutor. Caso contrário, se escolher um condutor privado e este não estiver disponível, será colocado em fila de espera, com a opção de cancelar.

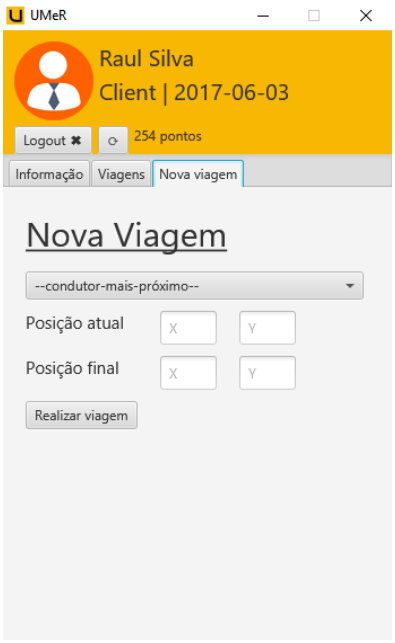

Figura 15: Separador nova viagem

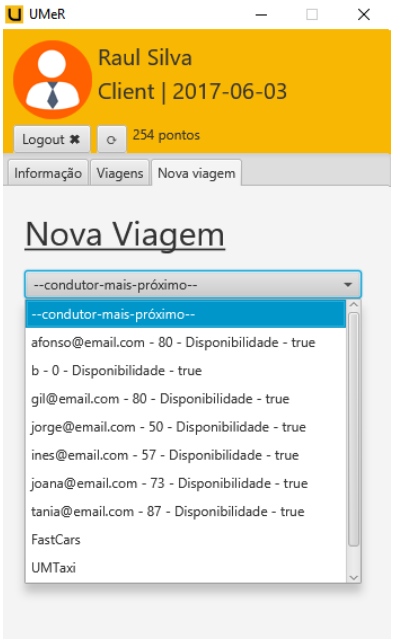

Figura 16: Opções de viagem

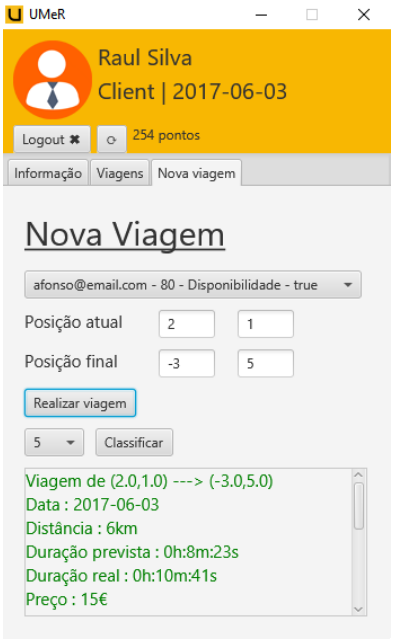

Figura 17: Viagem efetuada com sucesso

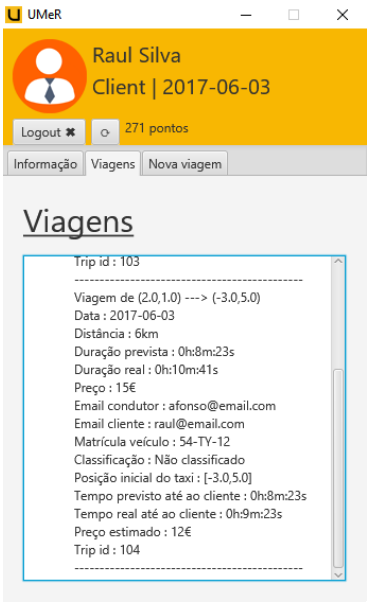

Figura 18: Consultar informação da viagem

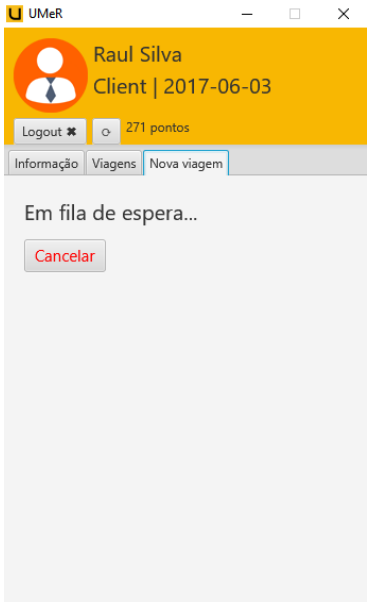

Figura 19: Caso condutor não esteja disponível

6.4 Admin

Para podermos aceder à informação global de `UMeR` e consultar, por exemplo, `top 10 classificações`, decidimos criar uma conta que poderá aceder a tudo isto (nome: `admin` password: `12345`). Nele poderemos aceder a todos os utilizadores, veículos e empresas, a todas as viagens, a diversos *tops* (classificações, dinheiro gerado/gasto e desvios, obtidos a partir dos comparadores respetivos) e ao dinheiro gerado por um determinado veículo ou empresa entre diversas datas.

Figura 20: Menu administrador

Figura 21: Separador dinheiro gerado

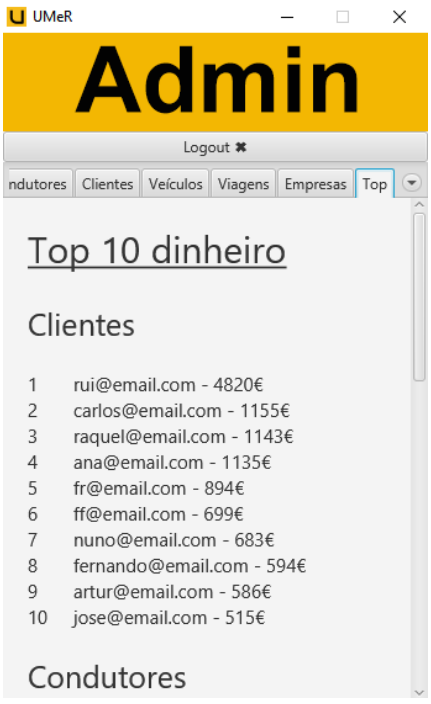

Figura 22: Separador *tops* (1)

Figura 23: Separador *tops* (2)

7 Conclusão

Após a realização deste trabalho conseguimos não só aprender como fazer um projeto (com início, meio e fim) através da programação por objetos mas também melhoramos as nossas capacidades gerais como programadores.

Os conceitos de hierarquia e de reutilização de código poupam muito trabalho em projetos deste tipo, onde, por exemplo, temos apenas que fazer uma classe de utilizador, sendo preciso adicionar o pouco necessário para criar diferentes tipos de utilizadores.

Com as estruturas eficientes do `Java` foi possível aumentar a produtividade, visto que não perdemos tempo a fazer tabelas de *hash* ou lista ligadas, e tínhamos a certeza de que quando aparecia algum problema, não era da estrutura.

A parte mais difícil, mas certamente a mais interessante, foi a construção da interface gráfica (em `JavaFX`). Foi bom poder construir algo visualmente apelativo e muito semelhante a programas "profissionais", sendo que no futuro já temos a experiencia para fazer interfaces deste tipo.

Para concluir, o que achamos que poderia ter sido feito melhor era talvez adicionar mais funcionalidades e ter uma melhor organização do código, sobretudo na classe da interface gráfica. Contudo, achamos que cumprimos os objetivos necessários, ficando assim com uma ideia positiva do trabalho final.