Relatório do Trabalho Prático de Programação Orientada a Objetos Grupo11 POO2017

Elísio Freitas Fernandes {55617}, Daniel Gonçalves Martins {73175}, and Nuno José Ribeiro da Silva {78879}

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mail: {a55617, a73175, a78879}@uminho.pt

Daniel Gonçalves Martins {73175}

Elísio Freitas Fernandes {55617}

Nuno José R. da Silva {78879}

Resumo Serve o presente como relatório do projeto elaborado no âmbito da unidade curricular de Programação Orientada a Objetos. Projeto esse no qual se previa a elaboração de um programa para a empresa UMeR, o qual fosse capaz de garantir a prestação continua do seu serviço. Os requisitos de tal programa incluem a criação e manutenção da base de dados que inclui os dados dos clientes, bem como as informações dos seus colaboradores e das suas viaturas, utilizadas para a prestação do serviço em questão. Mais ainda, está incluída a criação de uma interface de interação com os utilizadores com várias opções, como: criação de conta; posterior acesso e atualização dos dados da conta; consulta de histórico de serviços requeridos/prestados.

1 Introdução

Com o presente documento pretende-se apresentar resumidamente o trabalho prático elaborado pelo grupo, como proposta de resolução do enunciado apresentado, no seguimento da UC de programação Orientada aos Objetos. As necessidades da empresa UMeR foram abordadas e tratadas usando java, temos recorrido maioritariamente ao Atom como editor de texto e o Terminal como compilador, bem como o ***** e a consola do linux para windows. Irão ser apresentadas as classes elaboradas e a várias escolhas especificas feitas em cada uma delas, a sua hierarquia e sua razão de ser.

2 Tipos de dados usados

No geral, as estruturas de dados usadas para o armazenamento dos dados de clientes, veículos e viagens foram os *Maps* - verificar, por exemplo, **Figura 1**. Dada a sua eficácia em termos de armazenamento ordenado de

dados/rapidez na procura, à sua conveniência, uma vez que permite identificar cada objecto unicamente através de um dado único. Queríamos garantir a unicidade de dados, pois não haveria razão para termos duas entradas de registo para o mesmo utilizador, assim, usamos o email do mesmo como *key* de identificação no *Map* e assim garantimos que o mesmo *User* não se regista duas vezes. Poupamos memória e reduzimos dados duplicados nas bases de dados da empresa. O mesmo se dá para os *Vehicle*, *Trips* e para a associação de *Driver*/*Vehicle*.

```
public class UMeR implements Serializable
{
    private boolean isLogged;
    private String loggedUserEmail;
    private Map<String, User> userList;
    private Map<String, Vehicle> vehicleList;
    private Map<Integer, Trip> tripList;
    private Map<String, String> driverVehicle;
    private int tripNumber;
```

Figura 1. Variáveis da classe UMeR

Por outro lado, usamos também dados do tipo *List* em situações como na classe *User*, para guardar os ids das viagens efetuadas, uma vez que o tipo de dados é só um tipo de dados primitivo.

2.1 Metodologias de implementação dos tipos de dados

A nível da declaração das variáveis de instância foi usado sempre um tipo de dados com um grande nível de generalização. Com tal pretendemos facilitar a alteração dos tipos de dados específicos usados. Por exemplo, ao usar a declaração do tipo *Map* invés de *HashMap* permitimos que no futuro fosse fácil alterar esse tipo para um *TreeMap* por exemplo, mantendo no entanto a mesma intenção de permitir uma gestão eficiente e organizada da informação.

Pela mesma razão declaramos as variáveis de instância como *List*, permitindo depois a escolha do tipo específico de *List* pretendida - verificar, por exemplo, **Figura 2** .

```
public abstract class User implements Comparable<User>, Serializable {

    private String name;
    private Address address;
    private LocalDate birthday;
    private String email;
    private String password;
    private List<Integer> tripHistory;
    private double totalTripCost;
```

Figura 2. Variáveis da classe User

Observando a **Figura 3**, é visível a uniformização dos métodos das classes a esta metodologia de abstração do tipo de dados.

3 Métodos

Em cada classe existem métodos que permitem a interação da empresa com a mesma, permitindo a criação, de instâncias de classe, atribuição de valores ás várias variáveis através dos métodos *set<nome da variável>*, bem como obtenção dos valores contidos nas mesmas, através dos métodos *get<nome da variável>*.

```java
/**
 * Gets copy of this instance User's trip history
 * @return Returns List that's a copy of this instance of User's Trip History
 */
public List<Integer> getTripHistory(){
    List<Integer> aux = new LinkedList<Integer>();

    ((LinkedList<Integer>) this.tripHistory).forEach(trip -> {aux.add(trip);});

    return aux;
}
```

Figura 3. Métodos com abstração de dados assegurada

Cada classe tem em comum os métodos clone(), equals(), compareTo(), hashCode() e toString(). O método toString() e o equals() é definido na classe abstrata e redefinido nas subclasses. O método é definido na classe abstrata porque várias da variáveis são definidas nesta classe e, assim, evita-se a duplicação de código, no entanto, existem também variáveis declaradas na classe *client*, pelo que é necessário chamar o *super* da super classe para tratar da execução do método sobre as variáveis lá definidas e só depois é aplicado o código local ao resto das variáveis de instância. O método clone() é um método abstrato e, portanto, é totalmente definido em cada subclasse, uma vez que apenas a subclasse sabe como definir cada um dos parâmetros que compõe. O método compareTo() é o responsável pela "ordem natural"definida em cada classe para a ordenação das suas instâncias. Já o método hashCode() é um método que todas as classes têm apenas na sua superclasse abstrata, uma vez que é nela que se encontra o identificador único inerente a cada uma das subclasses que a ampliam.

Salienta-se o método *setPassword (String password)* e *public String getPassword ()* que na sua simplicidade, apenas guardam a password entregue pela empresa, sendo da responsabilidade e controlo da mesma garantir a segurança da mesma - nesta proposta tal método não foi implementado. Destacam-se também métodos construtores vazios da classe *User*, *Vehicle* e *Trip* que são privados, uma vez que não pretendido que estes sejam criadas instâncias destas classes sem dados concretos. No entanto, dada a filosofia do Java, foi necessário criar os construtores, para que o Java em si não desse automaticamente essa possibilidade aos utilizadores.

Para além destes métodos, cada classe dispõe dos seus métodos próprios, os quais podem ser consultados em anexo, necessários para efetuar as suas operações.

4 Expressão escrita

Ao longo do projeto foi usado um estilo restrito de regras para a criação dos seus ficheiros componentes. Para a declaração das variáveis foi definido que as mesmo deveriam ter nomes sugestivos que tornassem fácil compreender aquilo que as mesmas representavam. Mais ainda, para variáveis compostas por várias palavras, foi adotado o método lowerCamelCase, em que cada letra de cada palavra é iniciada por letra maiúscula.

5 Arquitetura

Nesta proposta foram utilizados vários tipos de classes com diferentes atributos. Desde classes concretas a classes abstratas, classes com e sem implementações de interfaces e subclasses de classes abstratas.
Por questões de necessidade de gravação do estado da aplicação, todas as classes tem em comum a implementação da interface *Serializable*, oferecendo, assim, um método simples de gravação dos objetos com todos os seus estados atuais em ficheiro, para posterior consulta.

5.1 Classe UMeR

Utiliza Maps, e tipos primitivos para guardar os seus dados. Possui um registo de toda a informação necessária para o bom funcionamento da empresa. Contém quatro estruturas de dados Map onde se encontram guardados os dados de: 1. User como *value* e email como *key*, para guardar todos os utilizadores registados na empresa, quer clientes, quer condutores ; 2. Vehicle como *value* e licensePlate como *key*, para guardar a informação da

viaturas ao serviço da empresa; 3. Trip como *value* e id de viagem como *key*, para manter um histórico de todas as viagens efetuadas através da empresa; 4. Email como *value* e licensePlate como *key*, para manter um registo do veículo associado a cada condutor. Existe ainda a variável isLogged, do tipo *boolean*, que indica se existe um utilizador "logado". Se sim, então o seu e-mail estará presente na variável loggedUserEmail, do tipo *String*. Por último, existe na classe uma variável tripNumber, do tipo *Integer*, responsável por garantir a sequencialização dos identificadores únicos da viagem, utilizados no Map das Trips.

Foram aglomerados todos os dados do tipo *User*, respetivamente, todos os dados de tipo *Trip* e *Vehicle*, num só *Map*, uma vez que eram compatíveis, de maneira a ser possível efetuar operações sobre todos os utilizadores de uma só vez, bem como facilitar a adição de novos tipo de utilizadores.

5.2 User

Não são permitidas instanciações desta classe, uma vez que se trata duma classe abstrata, que implementa comparadores. Define os dados dos utilizadores que são comuns entre as suas subclasses e pressupões a criação de subclasses que permitirão a instanciação de objetos deste tipo. Contém variáveis de tipos primitivos e uma *List* para guardar os registos da viagens efetuadas pelo utilizador.

Client A classe *Client* é uma subclasse de *User*, tem a particularidade de possuir uma variável do tipo personalizado *Coordinates* que permite guardar a localização do cliente num plano cartesiano. Esta é atualizada após ter efetuado uma viagem.

Driver A classe *Driver* é uma subclasse de *User*, tem a particularidade de possuir variáveis referentes ao seu performance como motorista bem como um total de km já percorridos ao serviço da empresa e a indicação da sua disponibilidade atual para efetuar um serviço. A sua disponibilidade é alterada conforme o mesmo está ou não a efetuar uma viagem.

A classificação do condutor é determinada pela média das suas classificações. A performance é um valor de 0 a 100. Este valor começa por ter valor 100, sendo que caso a sua primeira viagem tenha um desvio superior a 25% do tempo estimado, ele passa automaticamente para 50. Após estas situações concretas, este valor é alterado segundo a subtração ou soma ao valor atual de 50 / (<numero de viagens efetuadas anteriormente> + 1), dependendo de a viagem efetuada ter ou não excedido 25% do tempo estimado. Ou seja, a sua performance vai aumentado, ou diminuindo, um valor incrementalmente pequeno conforme faz mais viagens.

5.3 Vehicle

Não são permitidas instanciações desta classe, uma vez que se trata duma classe abstrata, que implementa comparadores. Define os dados de utilização dos veículos utilizados ao serviço da empresa, bem como o custo da viagem por quilómetro, *fare*, a fiabilidade, *reliability*, e ainda a sua localização, *location*, do tipo *Coordinates*, que é atualizada no fim de cada viagem. Possui os métodos abstratos clone() e setReliability().

Car, Motorcycle e Van Estas classes são subclasses de *Vehicle*, são idênticas e não possuem variáveis próprias. Cada uma das classes possui ume definição dos métodos abstratos da sua superclasse.

5.4 Trip

A classe *Trip* é a responsável pela definição do parâmetros das viagens efetuadas, incluindo: o id da viagem, o id do cliente e do condutor, a data em que foi efetuada, a matricula do carro que efetuou a viagem, a localização do veículo, do cliente e do destino, o custo/tempo estimado da viagem e o custo/tempo real da mesma. Esta classe possui também comparadores para que as viagens possam ser ordenadas segundo mais do que um critério.

5.5 Coordinates

A classe *Coordinates* é a responsável pela definição da localização de algum objeto e tratamento de do cálculo da distância entre dois pontos.

5.6 Address

A classe *Address* é a responsável pela definição da cidade e país de um dado utilizador, contendo apenas *Strings*. Implementa exceções.

5.7 EmailValidator

Classe responsável por analisar o formato de um dado email, e verificar se tem um formato válido.

5.8 Interface

A classe *Interface* é a responsável pela impressão dos menus no ecrã, contendo apenas métodos que imprimem conjuntos de *Strings* no ecrã. Cada conjunto de *Strings* imprimidas é diferente de acordo com o menu que se pretende imprimir.

5.9 Main

A classe *Main* é a principal responsável pelo funcionamento da aplicação, uma vez que é ela que continuamente recebe, interpreta e valida os comandos introduzidos pelo utilizador, imprimindo os menus solicitados e chamando as funções da UMeR necessárias para efetuar as opções solicitadas. A classe Main, a classe UMeR e o utilizador têm uma relação de repetida comunicação para solicitar, obter, introduzir e validar dados. Acima de tudo, esta classe é a responsável pelo tratamento de erros.

5.10 Comparadores

Para além das classes mencionadas, esta proposta dispõe também de um conjunto de comparadores, os quais podem ser usados para ordenar as listas e os maps por uma ordem diferente da ordem natural determinada pelo método compareTo().

6 Hierarquia

Podemos ver abaixo na **Figura 4** a hierarquia das classes do projeto e a maneira como todas elas estão ligadas entre si. Podemos também verificar na **Figura 5** uma representação gráfica das exceções previstas no código.

7 Manual de utilização

Quando abrimos a aplicação deparamo-nos com o **ecrã 4**

8 Conclusões

Neste trabalho...

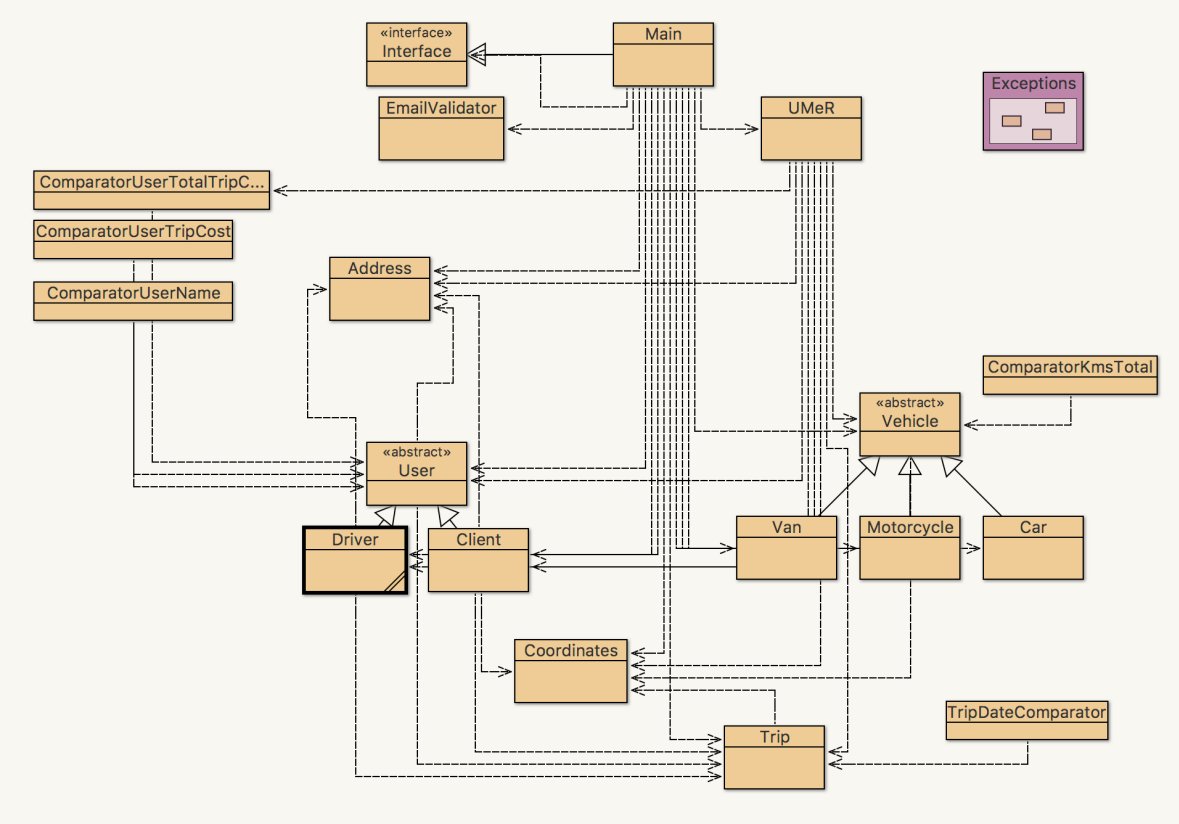

Figura 4. Hierarquia de classes na UMeR

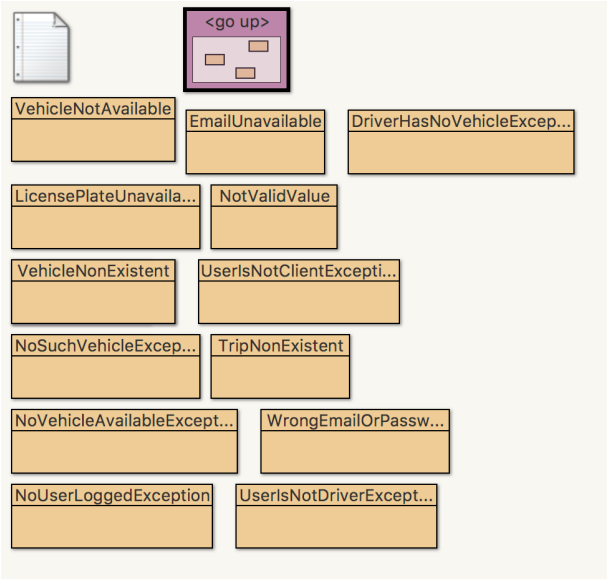

Figura 5. Exceções previstas no código