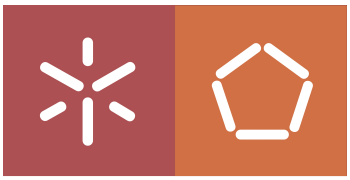

Universidade do Minho
Escola de Engenharia

UMCarroJá!

Relatório do projeto prático de POO

Grupo 32

José Magalhães,A85852

Luís Ramos,A83930

Luís Vila,A84439

Conteúdo

1 Introdução

No âmbito da unidade curricular Programação Orientada aos Objetos, foi proposto desenvolver um projeto em *Java*, mais concretamente um serviço de aluguer de veículos particulares, denominado *UMCarroJá!*.

Nesta aplicação será possível um proprietário registar um veículo na mesma podendo este ser utilizado por um cliente que possua um registo nessa mesmo *App*. De salientar que, tanto o cliente como o proprietário podem, respetivamente, aceitar,ou não, o veículo e o aluguer. Esta opção poderá ser tomada com base nas carateristicas que ambos possuem.

Um cliente poderá escolher um tipo de aluguer dependendo da sua conveniência. Este poderá procurar o veículo mais barato, o mais perto, um específico ou então um atribuído pela *App*. Por sua vez, um veículo tem na sua definição atributos como as suas coordenadas, o seu tipo de combustível, o seu preço cobrado por quilómetro, a sua autonomia, entre outros, que ajudam à decisão do cliente. Todos os movimentos desta aplicação são devidamente guardados, sendo que tanto o cliente como o proprietário poderão ter, a qualquer hora, acesso aos mesmos.

Os utilizadores desta *App* têm à sua disposição um menu com todas as funcionalidades disponíveis.

2 Descrição da arquitetura das classes

2.1 Carro

A classe **Carro** é, na verdade, uma *superclass*. Esta é abstrata sendo possível, assim, inserir a qualquer momento um novo veículo cujas especificações base serão os atributos definidos em *Carro*. Estes atributos passam não só pelas caraterísticas do veículo, mas também contém na sua definição uma estrutura de dados de modo a guardar o histórico de alugueres do mesmo.

```
public abstract class  Carro implements Serializable {
    private double velMedia;
    private double precoKm;
    private double consumo;
    private ArrayList<Aluguer> historico;
    private double classificacaoCarro;
    private double fatorFiabilidade;
    private String matricula;
    private String modelo;
    private Ponto2D coordenadas;
    private double autonomiaInicial;
    private double autonomiaAtual;
    private boolean disponivel;
    }
```

Este modelo passa para as subclasses *Eletirico*,*Gasolina* e *Hibrido*, que correspondem aos três tipos de veículos pedidos para a aplicação. Notar que o que distingue os três tipos é o seu tipo de combustível. Teremos, então, as seguintes subclasses:

```
public class Eletrico extends Carro

public class Gasolina extends Carro

public class Hibrido extends Carro
```

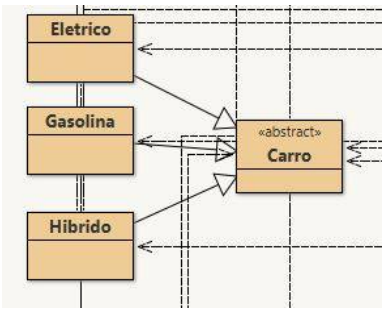

Figura 1: Estrutura de classes relativa aos veículos.

2.2 *Ator*

A classe **Ator** é, também, uma *superclass*. Sendo abstrata é possível, assim, inserir a qualquer momento um novo ator no sistema cujas especificações base serão os atributos definidos em *Carro*. Estes atributos correspondem a dados relativos ao ator. De notar que a *classificacao* é atualizada a cada aluguer.

```
public abstract class Ator implements Serializable {
    private String email;
    private String nome;
    private String password;
    private String morada;
    private GregorianCalendar dataNascimento;
    private int nif;
    private double classificacao;
    }
```

Este modelo passa para as subclasses *Cliente* e *Proprietario*, que correspondem aos dois tipos de utilizadores que a aplicação permite.

Além destas especificações bases, ambas têm ainda na sua definição novos atributos e novas estruturas necessárias à execução da *App*.

Cliente

```
public class Cliente extends Ator implements Serializable {
    private Ponto2D coordenadas;
    private ArrayList<Aluguer> historico;
    }
```

Um cliente, além dos seus dados padrão possui ainda uma estrutura de coordenadas e uma estrutura de dados de alugueres. A primeira indica a posição do cliente, sendo posteriormente necessária para a realização do aluguer. A segunda guarda na base de dados o historial de alugueres do mesmo.

Proprietario

```
public class Proprietario extends Ator implements Serializable {
    private ArrayList<Aluguer> historico;
    private HashMap<String,Carro> conjunto;
    private Queue<Aluguer> pendentes;
    }
```

Um proprietário, além dos seus dados padrão possui ainda três estruturas distintas. A primeira passa por guardar automaticamente os alugueres que as suas viaturas efetuaram. A segunda tem em sua possa o conjunto dos veículos que o mesmo possui. A terceira, por sua vez, terá os pedidos de alugueres efetuados por clientes para serem, ou não, aceites.

2.3 *Aluguer*

Esta classe é responsável pela realização do objetivo da *App*, o de efetuar alugueres. De referir que esta apresenta toda a informação de um aluguer: o *Carro* usado para a viagem, todas as coordenadas necessáras, distâncias, tempos e respetivos preços. Possui ainda a avaliação atribuida pelo cliente,o respetivo nif

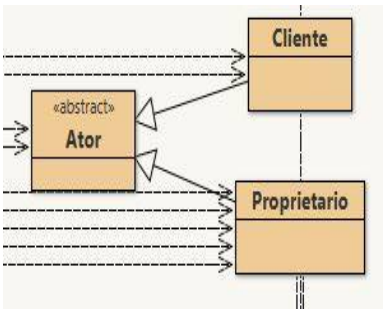

Figura 2: Estrutura de classes relativa aos atores.

e a data do aluguer. Realça-se que *coorPartida* correspondem às coordenadas iniciais do cliente, sendo que *coordChegada* correspondem às coordenadas finais do aluguer e, consequentemente, do veículo e do cliente. O *boolean autonomia* permite, após cálculo de distâncias, verificar se a viatura em questão possui autonomia suficente para efetuar a viagem.

```
public class Aluguer implements Serializable{
    private Carro carro;
    private Ponto2D coordPartida;
    private Ponto2D coordChegada;
    private boolean autonomia;
    private double distanciaInicial;
    private double distanciaAluguer;
    private boolean aceitaAlug;
    private double precoEstimado;
    private double precoReal;
    private double tempoCliente;
    private double tempoReal;
    private int avaliacao;
    private LocalDate dataAlug;
    private int nifCliente;
    }
```

2.4 *Ponto2D*

Uma classe com o intuito de facilitar o uso das mais variadas coordenadas ao longo da execução da aplicação

```
public class Ponto2D implements Serializable {
    private double x, y;
    }
```

2.5 *Menu*

Esta classe permite uma melhor interação com o utilizador, sendo fundamental para a construção dos menus interativos. Possui na sua definição uma estrutura de dados que guarda as escolhas do *user*, bem como a opção selecionada pelo mesmo.
É através dela que é possivel interpretar as intenções do utilizador.

```
public class Menu implements Serializable {
    private List<String> escolhas;
    private int nEscolhas;
    }
```

2.6 *UmCarroJa*

Esta classe contém as duas estruturas de dados mais importantes do programa. Estas contêm toda a informação referente à base de dados. Ao longo desta classe são elaborados métodos necessários para o total funcionamento da aplicação. De destacar que é responsável, também, pelos métodos de leitura e respetiva inserção dos *logs* na aplicação, criando assim a base de dados do programa.

```
public class UmCarroJa implements Serializable {
        private HashMap<Integer, Ator> map;
        private HashMap<String, Carro> carro;
        }
```

2.7 *UmCarroJaApp*

É o motor de toda a aplicação. A classe anteriormente falada, *UmCarroJa* é inicializada e então estabelece-se toda a base de dados do programa. É nesta mesma classe que são apresentados ao utilizador todos os menus existentes e, por isso, toda a interação entre o *user* e a base de dados só é possível através da mesma.

Consoante as várias opções do utilizador, este módulo está encarregue de interpretá-las e apresentar o passo seguinte, bem como guardar,automaticamente, todos os novos dados introduzidos.

De destacar que contém também os métodos de gravar e ler estados de modo a criar uma base contínua de informação após *restart* do programa.

```
public class UmCarroJaApp {
 private static UmCarroJa ucj;
 private static Menu menuInicial, menuCliente, menuProprietario,menuEscolhaAlug,menuTipoEs
    }
```

De um modo mais geral, podemos verificar a ligação entre todos os componentes acima referidos. Ligação esta fundamental para a boa execução do programa.

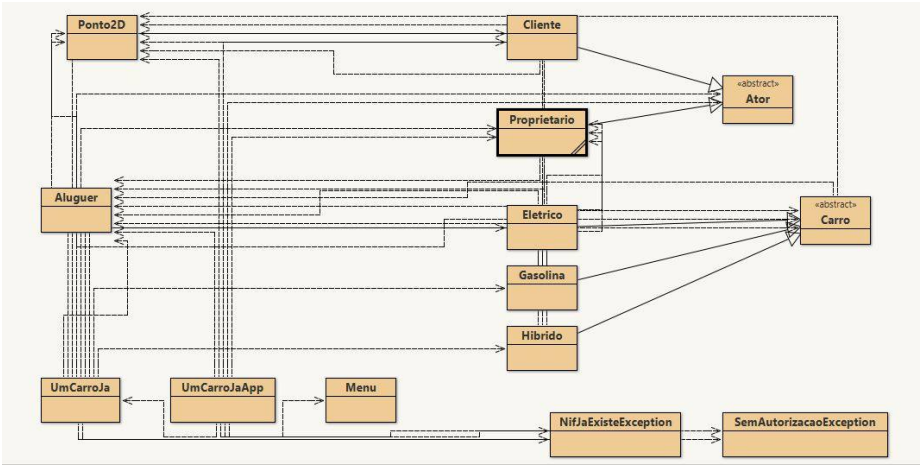

Figura 3: Ligação das diversas classes do programa.

3 Principais Estruturas de Dados

A aplicação possui diversas estruturas de dados fundamentais pelo que iremos abordar algumas delas de seguida.

Carro

Relativamente a toda a estrutura *Carro*, incluindo então as suas três subclasses, foi criado um *ArrayList* de modo a que este guardasse, em cada veículo, o histórico de alugueres que este tenha feito.

```
private ArrayList<Aluguer> historico;
```

Quanto às classes *Proprietario* e *Cliente* foram aplicadas diferentes estruturas de dados.

Proprietario

Relativamente ao proprietário foram criadas três estruturas distintas de modo a satisfazer melhor o que era pedido. Foi pensado, em primeiro lugar, um *ArrayList* que guardasse o histórico de alugueres proveninente dos veículos que este possui.

Para armazenar todos os veículos do mesmo proprietário foi criado um *HashMap* pois, tendo a matrícula como *key* torna muito eficiente a procura de um determinado veículo e consequentemente toda a informação referente ao mesmo. A inserção de novos veículos também aproveita esta mesma eficiência.

Por último, e devido à clausula de que o proprietário teria que aceitar um determinado aluguer, foi criado uma *LinkedList* que guarda, por ordem de mais recente, os pedidos feitos pelos clientes para,mais tarde serem, ou não, aceites.

```
private ArrayList<Aluguer> historico;
private HashMap<String,Carro> conjunto;
private Queue<Aluguer> pendentes;
```

Cliente

Para os clientes foi implementada uma única estrutura de dados,um *ArrayList*, como já visto anteriormente, responsável por guardar o historial de alugueres que estes efetuaram.

```
private ArrayList<Aluguer> historico;
```

UmCarroJa

As estruturas pertencentes a esta classe são as mais importantes do programa. Para todos os dados de utilizadores foi criado um *HashMap* cuja *key* corresponde ao NIF. Este é imútavel ao longo da *App* e permite,assim, distinguir o tipo de *Ator*, podendo ser proprietário ou cliente. Esta estrutura contém, então, todos os utilizadores do sistema, bem como todas as carateristicas dos mesmos. Esta escolha baseou-se no grande benefício de procura de um *value*, através do NIF, sendo esta praticamente instantânea. A sua inserção, bem como a alteração dos dados é feita de modo competente sendo, por isso, bastante conveniente ao nosso programa o uso deste mesmo *HashMap*.
A segunda estrutura contém como *key* a matrícula e como *value* o *Carro* correspondente. Foi idealizada como resposta a vários pontos pedidos pelo enunciado. Resposta essa bastante eficaz, sendo bastante útil na procura instantânea de caraterísticas de um determinado *Carro*.

```
private HashMap<Integer, Ator> map;
private HashMap<String, Carro> carro;
```

4 Guia de utilização

Para início da *App* é apresentado ao utilizador um menu onde este pode escolhar se pretende registar-se ou, caso já tenha efetuado registo se pretende iniciar sessão com o mesmo. São também mostradas as três *queries* pedidas pelo enunciado.

Caso escolha **Registar utilizador**, então serão pedidos os dados do utilizador. Caso opte por **Iniciar Sessão** então terá de inserir os dados correspondentes. Após ínicio, dependendo da sua categoria, apresentará dois menus distintos.

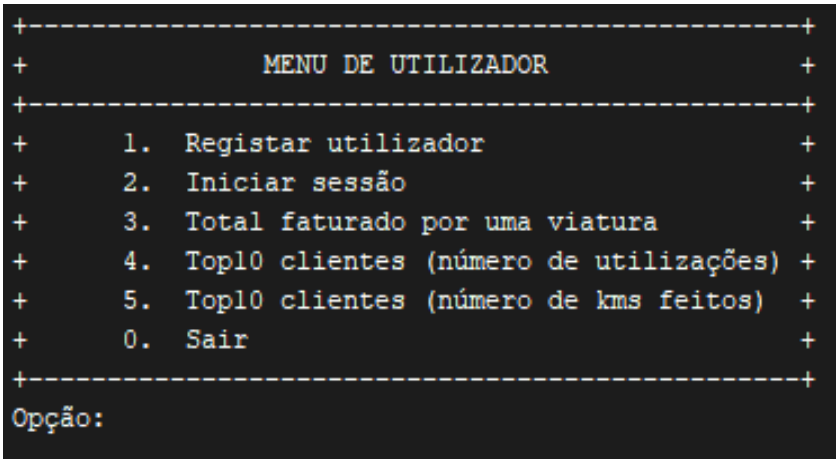

Figura 4: Menu inicial

Cliente

Se o utilizador com sessão iniciada for *Cliente*, então aparecerá um novo menu relativas às opções que ele pode tomar. Estas passam por iniciar um novo aluguer que, visualizar o histórico de alugueres ou classificar um aluguer já efetuado.

Se optar por efetuar um aluguer, então o programa pedirá as coordenadas

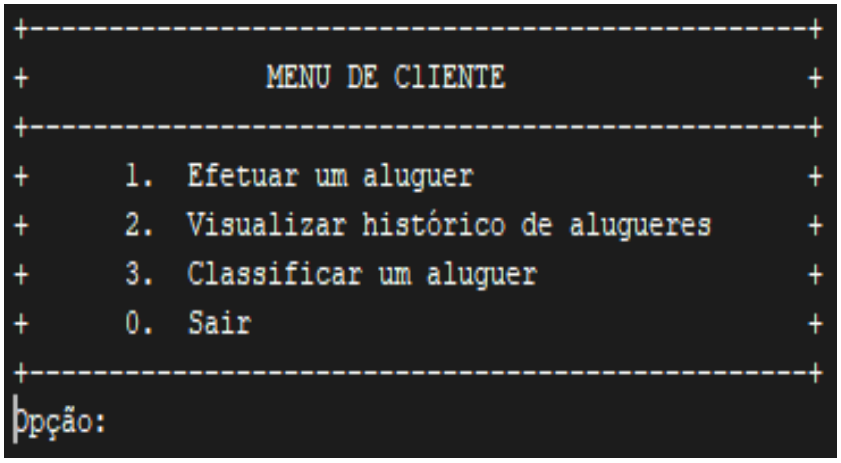

Figura 5: Menu correspondente ao *Cliente*

necessárias pelo que, de seguida, pede a preferência do *Cliente* quanto ao tipo de aluger. Independentemente da escolha, o menu seguinte será saber a preferência do utilizador quanto ao veículo a usar. Depois de tudo escolhido é ainda perguntado ao mesmo utilizador se aceita, ou não, o *Carro* que lhe é proposto, apresentando todas as caraterísticas do mesmo. Caso opte por não aceitar, então um novo carro é apresentado.

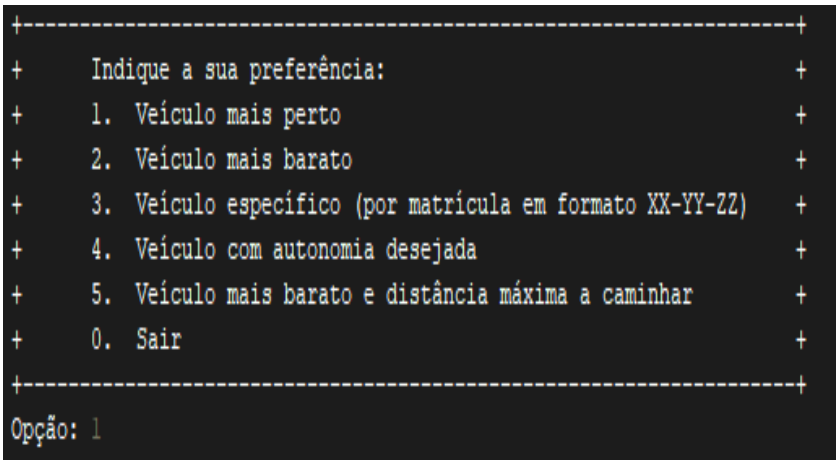

Figura 6: Menu correspondente ao tipo de aluguer

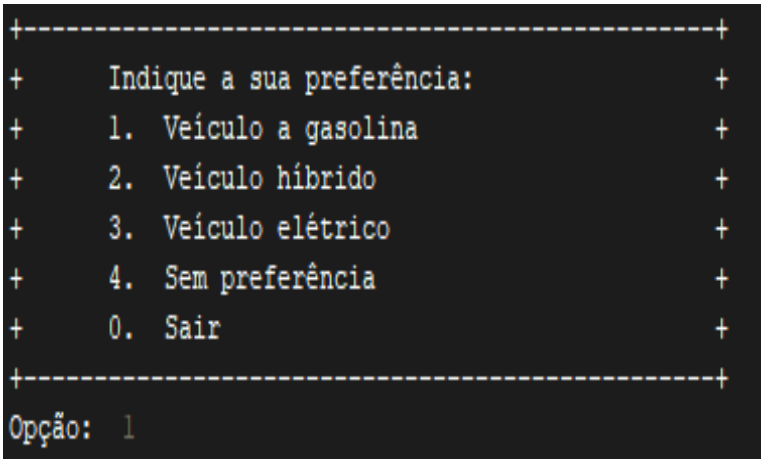

Figura 7: Menu correspondente à preferência do *Carro*

Proprietário

Se o utilizador com sessão iniciada for *Proprietario*, então aparecerá um novo menu relativas às opções que ele pode tomar.

No menu dedicado ao proprietario, então este poderá tomar várias iniciativas, entre elas adicionar uma viatura, alterar caraterísticas de uma já existente, abastecer um carro específico e visualizar um grande número de dados que o programa fornece, como presente na figura.

Independentemente da escolha pretendida, serão sempre, posteriormente, pedidos dados necessários para a sua realização.

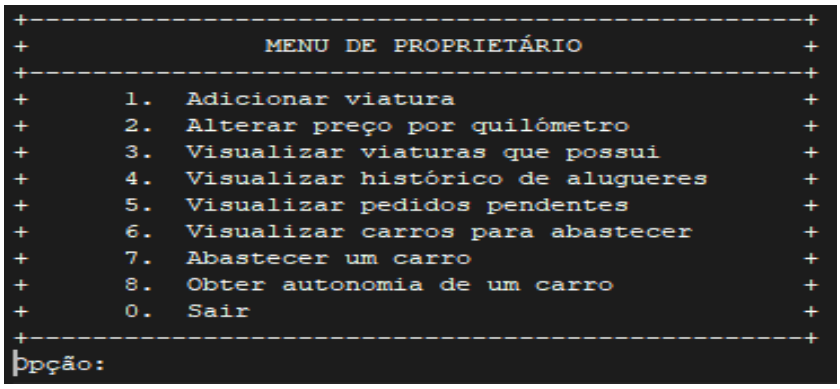

Figura 8: Menu correspondente ao *Proprietario*

5 Conclusão

A opinião do grupo é unânime no que toca à eficiência do projeto. Achamos que o nível desta é bastante bom pois a rapidez da *App* é uma prova sólida disso.

Quanto ao manuseamento do programa achamos que este é bastante fácil e claro sendo por isso bastante rápida a compreensão do utilizador.

Quanto à parte pedagócia, este projeto foi muito enriquecedor para o nosso coletivo, pois permitiu que adquirissemos agilidade numa linguagem até agora desconhecida, *Java*. De realçar a importa compreensão das estruturas de dados utilizadas.

Quanto à questão proposta pelo enunciado, sobre como seria possível incluir novos tipos de viaturas, chegamos à conclusão que não envolveria grandes alterações no código proposto.

A superclasse *Carro* pode ser extendida para qualquer novo tipo de viatura, pois, sendo abstrata, ao adicionar um novo tipo, este deverá implementar os atributos que a superclasse possui sendo apenas necessário declarar na classe *extends Carro*. É, também,necessário, retificar alguns métodos bastando apenas acrescentar o novo tipo.

Em suma, o esforço coletivo foi grande com o intuito de garantir boas soluções para o enunciado proposto deixando, assim uma janela aberta de novas ideias para trabalhos futuros.