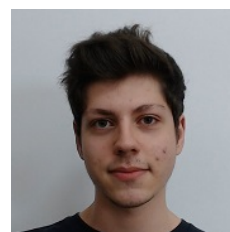
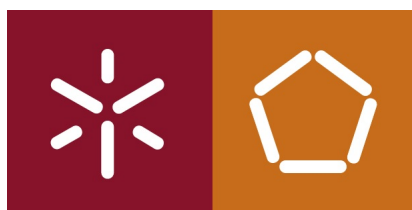


# Programação Orientada aos Objetos 2019

***UMCarroJá!***

José Pedro Silva (A84302)  
José Ricardo Cunha (A84577)  
Válter Carvalho (A84464)

26 de Maio de 2019



**Grupo nº 1**

Mestrado Integrado em Engenharia Informática  
Universidade do Minho

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Declaração de classes base</b>	<b>2</b>
2.1	Veículos . . . . .	2
2.1.1	Tipos de Veículo . . . . .	3
2.2	Atores . . . . .	3
2.2.1	Cliente . . . . .	3
2.2.2	Proprietário . . . . .	4
2.3	Aluguer . . . . .	5
<b>3</b>	<b>Estruturação principal</b>	<b>6</b>
3.1	Randomizador . . . . .	6
3.2	Aplicação . . . . .	8
3.3	Menu . . . . .	9
3.4	Lógica de Negócio . . . . .	9
<b>4</b>	<b>Aspetos a melhorar</b>	<b>11</b>
4.1	GUI (Graphical User Interface) . . . . .	11
4.2	Randomizador . . . . .	11
<b>5</b>	<b>Conclusão</b>	<b>12</b>

# 1 Introdução

Este relatório surge a propósito da elaboração do trabalho prático de *Programação Orientada aos Objetos* (POO) que é essencialmente um sistema de alugueres de veículos, dado o nome de **UMCarroJá!**, usando Java11 no IDE *IntelliJ*.

De forma resumida, trata-se de um encadeamento de tratamento de pedidos de *Aluguer* de um *Veículo*, no ponto de vista de *Cliente* e dum *Proprietário*. Os veículos estão definidos em 2.1, os clientes e proprietários são parte do mesmo subconjunto de atores, vistos em 2.2 e os alugueres são mencionados em 2.3.

## 2 Declaração de classes base

### 2.1 Veículos

```
1 public abstract class Veiculo implements Serializable {
2     private String ID; // Matricula
3     private String marca; // Marca do Veiculo
4     private String prop; // NIF do proprietario
5     private Ponto posicao; // Posicao (x,y) do veiculo
6     private double velocidade; // Velocidade media
7     private double priceKm; // Preco em Euro/km
8     private double consumoKm; // Consumo em L/km
9     private Set<Aluguer> historico; // Alugueres feitos
10    private List<Integer> classificacoes; // Ratings
11    private double depositoMax; // Autonomia maxima
12    private double depositoAtual; // Autonomia atual
13    (...)
14 }
```

O grupo implementou o veículo como se uma classe abstrata, tornando assim o processo de inserção na estrutura de dados principal (ver 3.4) muito mais simples, visto que não temos que dividir em 3 partes para cada tipo de veículo: *Elétrico*, *Híbrido* e *Gasolina*.

Para além disso, qualquer veículo, independentemente do tipo, é tratado exatamente da mesma forma, portanto não faria sentido o desdobramento da estrutura para alocar cada um dos três.

### 2.1.1 Tipos de Veículo

Como no enunciado pedia a divisão em três tipos, novamente, *Eléctrico*, *Híbrido* e *Gasolina*, simplesmente seguem todas a mesma estrutura:

```
1     public class [TIPO] extends Veiculo implements
      Serializable {
2         (...)
3     }
```

Desta forma forma, garantimos que inserir outro tipo qualquer de veículo apenas requer que seja feita uma nova classe que siga esta declaração hierárquica, simplificando muito o processo de inserção em estruturas.

Cada uma destas classes poderá, ainda, ter a sua própria gestão das variáveis como o consumo, etc, o que simplifica, também, a estruturação lógica do problema.

## 2.2 Atores

```
1     public class Ator implements Serializable {
2         private String email; // E-mail
3         private String name; // Nome
4         private String password; // Password
5         private String address; // Morada
6         private LocalDate birthday; // Aniversario
7         private Set<Aluguer> historico; // Alugueres feitos
8         private List<Integer> classificacao; // Ratings
9         (...)
10    }
```

Para um ator básico, implementamos desta forma, porém, há alguns pontos a considerar, nomeadamente acerca das definições hierárquicas.

A razão pela qual não é uma classe abstrata é porque faz sentido a divisão das 2 subclasses em 2 estruturas diferentes, uma para clientes e outra para proprietários.

Ao contrários dos veículos, um proprietário e um cliente são tratados de forma completamente distinta, pelo que mantê-los juntos não faz qualquer sentido do ponto de vista lógico.

### 2.2.1 Cliente

```
1     public class Cliente extends Ator implements Serializable
      {
2         private Ponto posicaoI; // Posicao Inicial
```

```

3         private Ponto posicaoF; // Posicao Final
4         private Set<Aluguer> queue; // Por classificar
5         (...)
6     }

```

O cliente é composto por esta estrutura, em que para além da informação básica dos atores tem também a informação da sua posição atual e o seu destino pretendido.

O cliente na lógica da aplicação segue os seguintes passos:

1. O cliente decide que carro quer alugar, segundo algum critério;
2. O cliente espera pela resposta do proprietário;
3. Se o proprietário aceitou, a viagem é efetuada;
4. O cliente pode dar uma classificação ao veículo e ao proprietário.

Tem, também informação acerca dos alugueres por classificar, pelo que são apenas adicionados aos alugueres efetivos os classificados pelo cliente, numa query específica dentro da aplicação, ordenados pelo seu ID.

### 2.2.2 Proprietário

```

1     public class Proprietario extends Ator implements
        Serializable {
2         private Map<String, Veiculo> frota; // Key=Matricula
3         private Set<Aluguer> queue; // Alugueres por aceitar
4         (...)
5     }

```

De forma semelhante ao cliente, este adquire toda a informação básica do ator. As diferenças estão, nas variáveis locais (como seria expectável), em que há informação acerca da frota de veículos e dos alugueres por aceitar do mesmo proprietário.

Todos os pedidos de aluguer seguem o mesmo esquema:

1. O cliente pede um aluguer de um veículo;
2. O proprietário desse veículo tem um aluguer adicionado à sua fila de espera;
3. O proprietário decide para cada aluguer da fila de espera se o quer aceitar ou não;

4. Caso seja aceite, todos os históricos das entidades são atualizados com o aluguer que o proprietário tratou.
5. O proprietário pode classificar o cliente.

A frota é indexada pelas matrículas dos carros (devido à garantia que são únicas) e os alugueres por aceitar são ordenados pelo ID inerente à classe.

## 2.3 Aluguer

```
1  public class Aluguer implements Comparable<Aluguer>,
   Serializable {
2      private int aluguerID; // ID do aluguer
3      private String veiculoID; // Matricula do veiculo
4      private String clienteID; // NIF cliente
5      private String propID; // NIF prop
6      private String tipo; // Tipo de Aluguer
7      private String tipoVeiculo; //Tipo de combustivel
8      private Ponto inicioPercurso; // Localizacao do
        cliente
9      private Ponto fimPercurso; // Destino do cliente
10     private Ponto posInicialVeiculo; // Localizacao do
        veiculo
11     private double preco; // Preco do percurso
12     private double tempo; // Tempo do percurso
13     private LocalDate date; // Data que foi realizado
14     private Randomizer randomizer; // Randomizador de
        percurso
15     (...)
16 }
```

O aluguer é simplesmente um acumulador de dados de percurso, que contém informação necessária para tanto o proprietário como o cliente. Para além desta informação base, tem também o preço da viagem, o tempo que demorou e a data em que foi realizado.

Tem como identificador um inteiro atribuído consoante a ordem de criação (ordem natural). Simultaneamente, tem: dados do carro, como a matrícula e combustível; dados de cliente, que são o NIF, posição inicial e final; dados de proprietário, que são o NIF, e a matrícula do carro.

Quanto ao randomizador, será visto mais à frente (em 3.1), pelo que serão explicitados os detalhes apropriadamente.

### 3 Estruturação principal

Para este projeto, conforme indicado pelas aulas teóricas, decidimos utilizar o modelo *MVC*, pelo que temos uma classe principal que apenas age como controlador, uma classe que gere o menu de interação e outra que gere toda a manipulação da estrutura de dados.

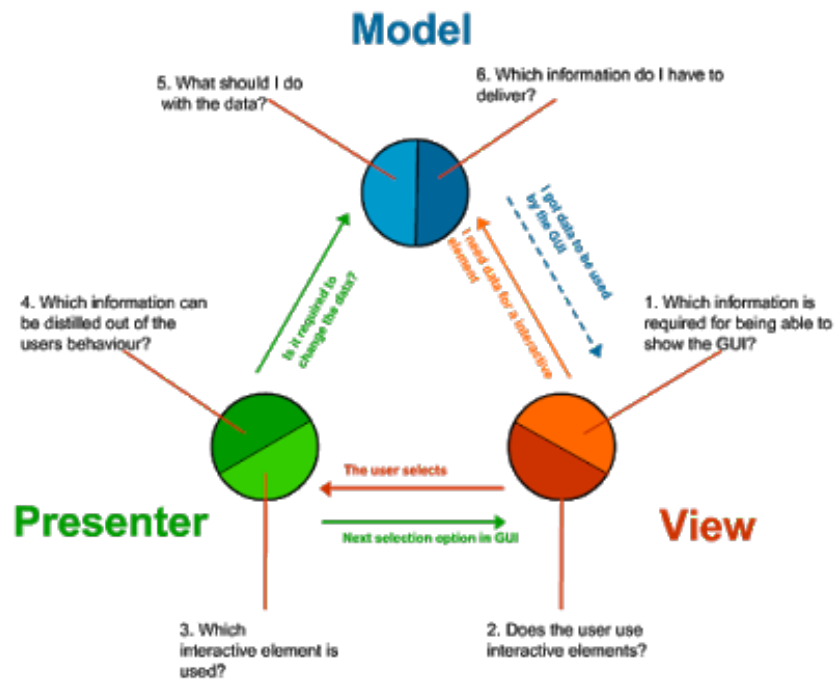


Figura 1: Organização de um MVC.

Para além disso e como fator de valorização, implementamos um *Randomizer*, que será visto já de seguida mais concretamente.

#### 3.1 Randomizador

```
1 public class Randomizer implements Serializable {
2     private String[] tempo = {"Sol", "Vento", "Chuva"};
3     private String[] condicoes = {"Completamente_
4         danificado", "Relativamente_danificado", "
        Relativamente_bem_cuidado", "Bem_cuidado"};
5     private String clima; // Escolha aleatoria de tempo
```

```

5         private String veiculo; // Escolha aleatoria de
           condicoes do veiculo
6         (...)
7     }

```

A nossa randomização funciona de forma a que tenha em conta várias situações:

- Clima atmosférico;
- Estado em que o carro foi deixado;
- Trânsito.

Ao arrancar a aplicação é de imediato detetado o clima, o estado do carro é único a cada aluguer e totalmente aleatório, assim como as influências do clima.

Em função das componentes do carro, temos três situações possíveis, quanto ao clima:

### 1. Sol

Neste caso apenas o clima não interfere nas "inconveniências" possíveis, pelo que não há desmultiplicador para a velocidade do carro, é a situação ideal (em termos de clima).

### 2. Vento

Quando há vento, há uma ligeira interferência na velocidade total do carro, devido à ação contrária ao movimento (porém não muito acentuada), levando a uma viagem relativamente mais longa e a um aumento relativamente pequeno da influência do trânsito.

### 3. Chuva

Em caso de chuva, os pneus dos carros têm péssimo atrito, levando a uma diminuição (por segurança) da velocidade média do veículo mais acentuada, consequentemente, leva a uma viagem ainda mais longa assim como uma maior influência do trânsito.

Em função destes três climas, o depósito do carro vai sofrer também alterações, que são calculadas consoante a diferença entre o tempo teórico e o tempo efetivo das viagens.

Quanto ao estado do veículo, é totalmente aleatório aquando a entrega ao proprietário, pelo que cabe ao proprietário classificar o cliente consoante o quão estragado foi deixado o seu veículo no final do aluguer.



## 3.2 Aplicação

```
1 public class App implements Serializable{
2     private MyLog logNegocio; // Logica de Negocio
3     private Menu menuCliente,menuProprietario,menuSign,
        menuSignUp; // Menus normais
4     private MenuLogin menuLogin; // Menu de login
5     private Randomizer estado = new Randomizer(); //
        Randomizador
6     (...)
7 }
```

A aplicação é essencialmente o controlador da aplicação. É o ponto de coerência entre os intervenientes da aplicação, como a leitura de dados, *IO*, etc.

É aqui que são inicializadas as instâncias de Menu's (3.3) e de Lógica de Negócio (3.4), pelo que as funcionalidades do programa são aqui criadas.

Primeiramente, permitimos na aplicação principal o seguinte:

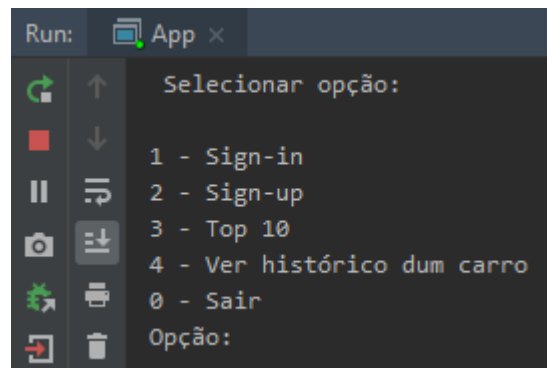


Figura 2: Menu principal da aplicação

Aqui é permitido ao utilizador decidir o que quer fazer:

1. Sign-in numa conta já existente;
2. Sign-up de um novo ator;
3. Top-10 clientes que mais utilizaram a aplicação;
4. Ver histórico de alugueres de um determinado carro.

Como funcionalidades específicas aos atores, admitimos a proposta do enunciado, como vemos nas duas próximas imagens.

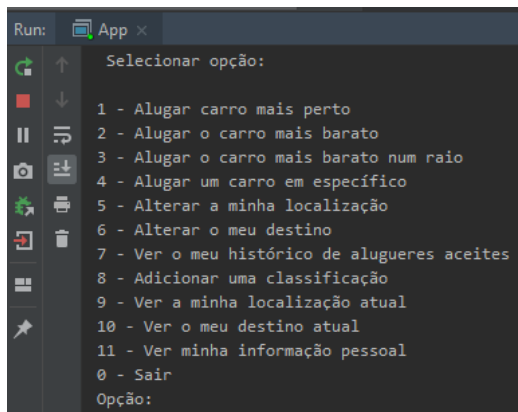


Figura 3: Para clientes

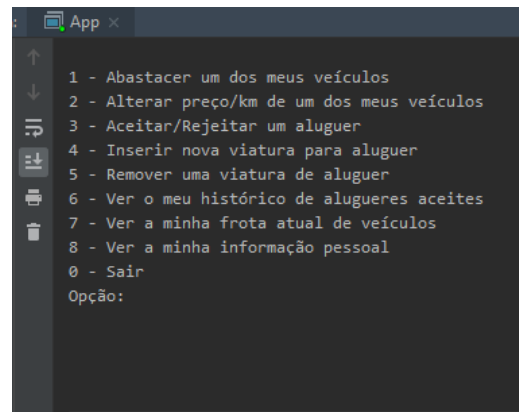


Figura 4: Para proprietários

### 3.3 Menu

```

1  public class Menu implements Serializable {
2      private List<String> opcoes;
3      private int op;
4      (...)
5  }
6  public class MenuLogin extends Menu implements
7      Serializable {
8      private String email;
9      private String password;
10     (...)
11 }

```

O grupo decidiu implementar dois tipos de Menus diferentes, como sugerem as definições acima. A razão pela qual o fizemos foi porque aquando o login, ajudava imenso manter o e-mail e a password guardadas algures facilmente acessíveis, pelo que o Menu de Login foi a melhor solução que encontramos.

Quando ao menu normal, assim como visto na aula teórica, é dado uma lista de strings que serão impressas como opções e espera *input* do utilizador, sendo o restante output resultado das interações com o utilizador.

É um menu relativamente simples, escrito totalmente em *output* de *strings*, que apesar de primitivo, é robusto e pouco sujeito a erros.

### 3.4 Lógica de Negócio

```

1  public class MyLog implements Serializable{

```

```

2      private Map<String,Cliente> clientes; // Mapeados
      pelo NIF
3      private Map<String,Proprietario> proprietarios; //
      Mapeados pelo NIF
4      private Map<String,Veiculo> listaVeiculos; //
      Mapeados pela matricula
5      private Set<Aluguer> alugueres; // Alugueres
      ordenados
6      private int counter; // ID do proximo aluguer

```

Esta classe introduz estruturas de dados que contêm a lógica toda do programa, permite efetivamente a *storage* de informação adequadamente.

Inicialmente, tal como imposto pelo enunciado, é lido do ficheiro de *logs* dado pela equipa docente e a partir daí criadas as estruturas de dados.

É também realizada, aquando o fim da interação do utilizador, uma salva-guarda de todo o estado interno da aplicação, para que possa ser retomada quando necessário e quando pretendido pelo utilizador.

Na figura que se segue mostramos a organização geral do nosso trabalho, como a hierarquia de classes, definições e lógica estrutural.

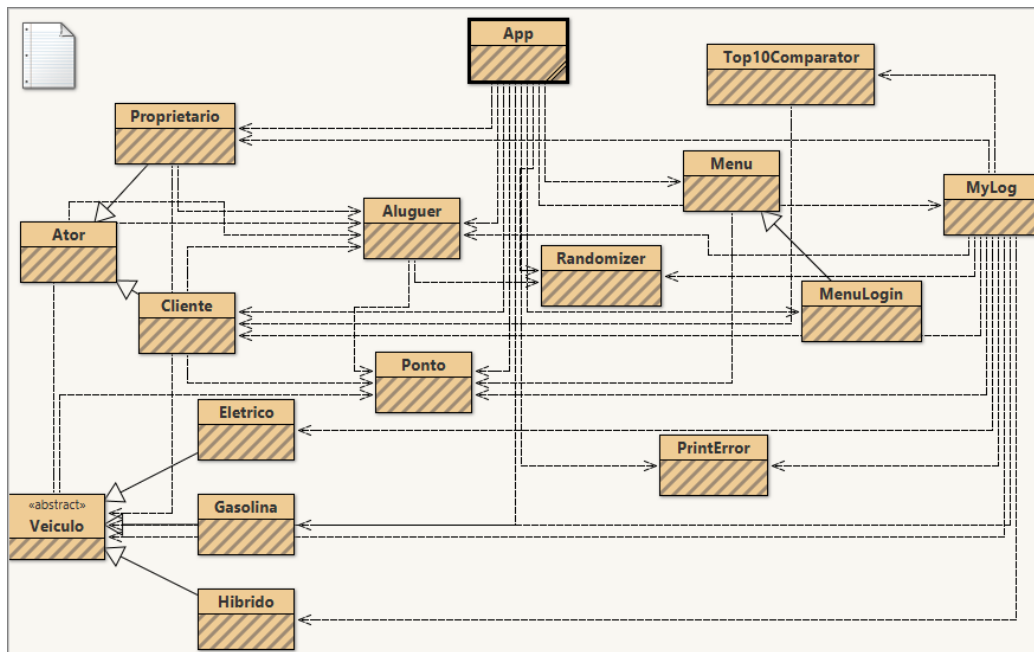


Figura 5: Diagrama classes (BlueJ).

## 4 Aspectos a melhorar

### 4.1 GUI (Graphical User Interface)

Como é evidente, ter tudo escrito em texto não é ideal no ponto de vista do utilizador.

Uma boa alternativa seria usar um dos inúmeros *framework's* de Java para criar uma interface visual, porém, além de não ser o objetivo da disciplina, era demasiado extenso para um trabalho com uma *deadline* tão curta.

Fica apenas para expectativa futura podermos trabalhar com esta parte do Java que realmente é muito conveniente e relativamente simples de utilizar, levando a que o trabalho tenha outro grau de qualidade e apresentação.

É realmente muito mau uma apresentação dum menu sem *clickables* (botões, *dialog boxes*, etc) pelo que esta seria fundamentalmente a principal alteração do programa que realmente traria uma mudança enorme.

Felizmente, devido ao encapsulamento e políticas de desenho do nosso trabalho era só alterar o método de *output* na classe *App*, pelo que caso queirássemos num futuro próximo alterar este Menu era facilmente exequível.

### 4.2 Randomizador

Como segundo aspeto mais importante a melhorar, há o Randomizador, descrito em 3.1.

É evidente que as viagens nunca correm como esperado, e considerando todos os fatores externos, é improvável que duas viagens com o mesmo destino tenham exatamente os mesmos cálculos quanto ao tempo, preço total, consumo do carro, entre outros.

Neste modo, estamos essencialmente a trabalhar num projeto de "simulação", pelo que seria ideal colocar estatística nas equações, como por exemplo:

- Introdução de **estações do ano**, isto é, **desvios-padrões** associados aos climas mais prováveis em cada uma;
- Introdução de **desvios-padrões** nos **consumos** dos veículos, porque nenhum veículo gasta sempre a mesma quantidade de combustível, independentemente de ser fóssil ou não;

- Introdução de **cálculos estatísticos** para o **estado de entrega** do veículo, tendo em conta a classificação do cliente (pode ser lida como reputação);

Infelizmente, não há tempo para aplicar algo tão complexo quanto este tipo de situações pelo que tivemos de aplicar algo relativamente mais simples, mas que serve como simulador também e o faz efetivamente como é esperado.

## 5 Conclusão

Em suma, o grupo no geral considera que aprofundou muito o seu conhecimento sobre o paradigma da Programação Orientada aos Objetos, nomeadamente de abstração, hierarquias, encapsulamento, tratamento de erros, *IO* e modelos de estruturação, neste projeto semestral.

Foi-nos dada a oportunidade de ver em primeira mão como se estruturam projetos em particular no Java, uma das línguas comercialmente mais procuradas, pelo que o seu valor enquanto aprendizagem é tomado pelo grupo inteiro como uma mais-valia.

Por fim, gostaríamos de agradecer à equipa docente não só pela disposição para com os alunos neste projeto e constante ajuda ao longo do semestre, mas também no ensino da linguagem Java nas aulas teóricas e práticas de forma concisa e eficaz, porque de facto foram utilizados conceitos de aula importantíssimos neste trabalho.