Uma imagem com texto

Descrição gerada automaticamente

**Relatório de Trabalho Prático**

**Programação Orientada a Objetos**

Docente: Luís Ferreira

Alunas:

Adriana Gomes 23151  
Joana Pereira 23153

Licenciatura em Engenharia de Sistemas Informáticos Pós-Laboral

**Índice**

[1. Introdução 4](#_Toc121006754)

[2. Descrição do Projeto 5](#_Toc121006755)

[2.1. Enquadramento do Projeto 5](#_Toc121006756)

[2.2. Diagrama de classes 5](#_Toc121006757)

[3. Implementação essencial das classes 6](#_Toc121006758)

[3.1. Utilizador 6](#_Toc121006759)

[3.2. TecnicoSAS 7](#_Toc121006760)

[3.3. Modelo 7](#_Toc121006761)

[3.4. MeioMobilidade 8](#_Toc121006762)

[3.5. MeiosMob 8](#_Toc121006763)

[3.6. Pedido 9](#_Toc121006764)

[3.7. Pedidos 10](#_Toc121006765)

[4. Conclusão 12](#_Toc121006766)

**Índice de Figuras**

[Figura 1 - Diagrama de classes. São apresentadas as classes desenvolvidas e os respetivos atributos, propriedades e métodos. 6](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386635)

[Figura 2 – Atributos e construtor da classe Utilizador. 7](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386636)

[Figura 3 - Redefinição do método Equals para a classe Utilizador. 7](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386637)

[Figura 4 - Função que permite criar utilizadores a partir de um ficheiro e adiciona à lista. 8](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386638)

[Figura 5 - Utilização de enum para definir os tipos de modelo possíveis. 8](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386639)

[Figura 6 - Função que devolve o valor de acordo com o modelo escolhido. 9](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386640)

[Figura 7 - Redefinição dos operadores de igualdade. 9](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386641)

[Figura 8 - Método desenvolvido para inserir um meio na lista de meios de mobilidade. 10](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386642)

[Figura 9 - Reescrita do método Equals que permite comprar pedidos. 10](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386643)

[Figura 10 - Reescrita da função ToString(). 10](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386644)

[Figura 11 - Operação que devolve o valor do pedido. 11](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386645)

[Figura 12 - Método que permite comparar dois pedidos. 11](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386646)

[Figura 13 - Construtor da classe Pedidos. 11](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386647)

[Figura 14 - Função que avalia se o meio de mobilidade está disponível nas datas pedidas. 12](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386648)

[Figura 15 - Método que permite verificar se um utilizador tem apenas um pedido em cada instante de tempo. 12](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386649)

[Figura 16 - Método que permite adicionar um pedido a uma lista de pedidos. 13](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386650)

[Figura 17 - Método que permite consultar os pedidos de um utilizador. 13](#_Toc123386651)

[Figura 18 - Método que mostra todos os pedidos de uma lista. 13](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386652)

[Figura 19 - Separação por camadas. 14](file:///C:\Users\ASUS\Desktop\IPCA\2ano\1semestre\POO\Projeto\Relatório_POO.docx#_Toc123386653)

# 1. Introdução

O presente relatório foi desenvolvido no âmbito da unidade curricular de Programação Orientada a Objetos (POO), que integra o plano de estudos do 1º semestre do 2º ano do curso de Licenciatura em Engenharia de Sistemas Informáticos. Este trabalho tem como objetivo o desenvolvimento de uma solução para um sistema de mobilidade sustentável que simplifique a acessibilidade entre o CAMPUS e zona envolvente por parte dos alunos, docentes e não docentes do IPCA. O sistema desenvolvido permite o aluguer de meios de mobilidade, tais como: bicicletas, bicicletas elétricas e trotinetes elétricas.

O desenvolvimento deste projeto divide-se em duas fases. Numa primeira fase, serão apresentados os seguintes pontos: diagrama de classes, implementação essencial das classes, definição e estruturas de dados a utilizar. Na segunda fase, realizar-se-á a implementação final das classes e serviços, fazendo a separação por camadas, incluindo uma aplicação demonstradora dos serviços implementados.

Este projeto será realizado com recurso à linguagem C#. Este relatório será acompanhado do projeto desenvolvido.

# 2. Programação Orientada a Objetos: C#

## 2.1. Pilares

Na programação orientada a objetos existem quatro pilares fundamentais:

* Abstração: Classes abstratas não são instanciadas e permite a reutilização de código.
* Encapsulamento: Consiste em ocultar detalhes da implementação, visto que o estado do objeto não pode ser alterado facilmente. O estado deve ser definido como *private*;
* Herança: Uma classe pode herdar os atributos e método de outra, designada “classe pai”. Representa uma relação generalização/especialização e é de “ser” e não “ter”.
* Polimorfismo: Classes derivadas da mesma “classe pai” podem invocar métodos que têm a mesma identificação, mas comportamento distinto.

## 2.2. Coleções *(collections)*

Em várias situações é necessário gerir objetos relacionados. Os objetos podem ser agrupados em *arrays* ou em coleções de objetos. Os *arrays* podem ser utilizados nos casos em que se conhece a quantidade de objetos. As coleções fornecem uma forma mais flexível de manipular grupos de objetos, visto que o seu tamanho pode aumentar ou diminuir de acordo com as necessidades. Uma coleção é uma classe, logo esta deve ser instanciada (*new*), para depois serem adicionados objetos. Para coleções do mesmo tipo de dados podem utilizar-se classes do *namespace System.Collection.Generic*. Neste caso, apenas objetos do tipo definido podem ser adicionados à coleção. Um exemplo de *Generic* são as listas, definidas como List<T>, em que T indica o tipo de dados. As classes no *namespace System.Collections* os dados não são de um tipo específico, sendo considerados do tipo *Object*, ou seja, na mesma coleção podem existir objetos de diferentes tipos. Um exemplo destes é ArrayList.

Neste trabalho, sempre que seja necessário agrupar objetos, serão agrupados com recurso a List<T>, de forma a permitir uma organização de acordo com o tipo de dados.

# 3. Descrição do Projeto

## 3.1. Enquadramento do Projeto

Caso um membro da academia do IPCA pretenda adquirir um meio de mobilidade, deve fazer um pedido, escolhendo a data de início e de fim, assim como, um meio de mobilidade que esteja disponível. Após avançar com o pedido, é calculado o preço a pagar com base na duração da utilização e no modelo do meio requisitado. O utilizador só pode realizar um pedido de cada vez e tem a possibilidade de consultar todos os pedidos efetuados. Os técnicos dos Serviços de Ação Social (SAS) são também utilizadores, mas como são também responsáveis pela gestão de pedidos e dos meios de mobilidade, pelo que podem consultar todos os pedidos realizados.

## 3.2. Diagrama de classes

Para o desenvolvimento do sistema de mobilidade foram criadas as classes apresentadas na figura 1. A classe Modelo foi desenvolvida para definir os diferentes modelos dos meios de mobilidade, que se encontram descritos na estrutura *enum* Tipo. A classe MeioMobilidade permite identificar univocamente cada meio e a classe MeiosMob permite guardar todos os meios criados numa lista. A classe Pedido define as características necessárias para criar um pedido, tendo também um estado (*enum*) associado. Na classe Pedidos são armazenados todos os pedidos válidos. A classe Utilizador permite identificar o autor dos pedidos. Os utilizadores são agrupados numa lista. Em todas as classes, os atributos são privados, tendo as respetivas propriedades públicas.

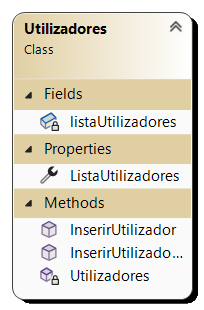
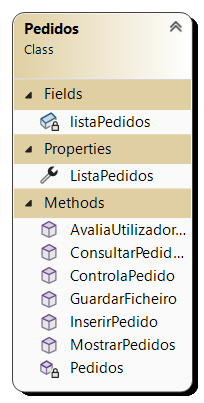
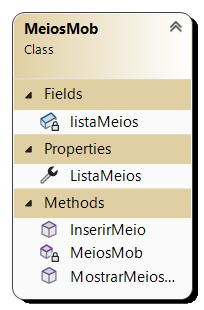
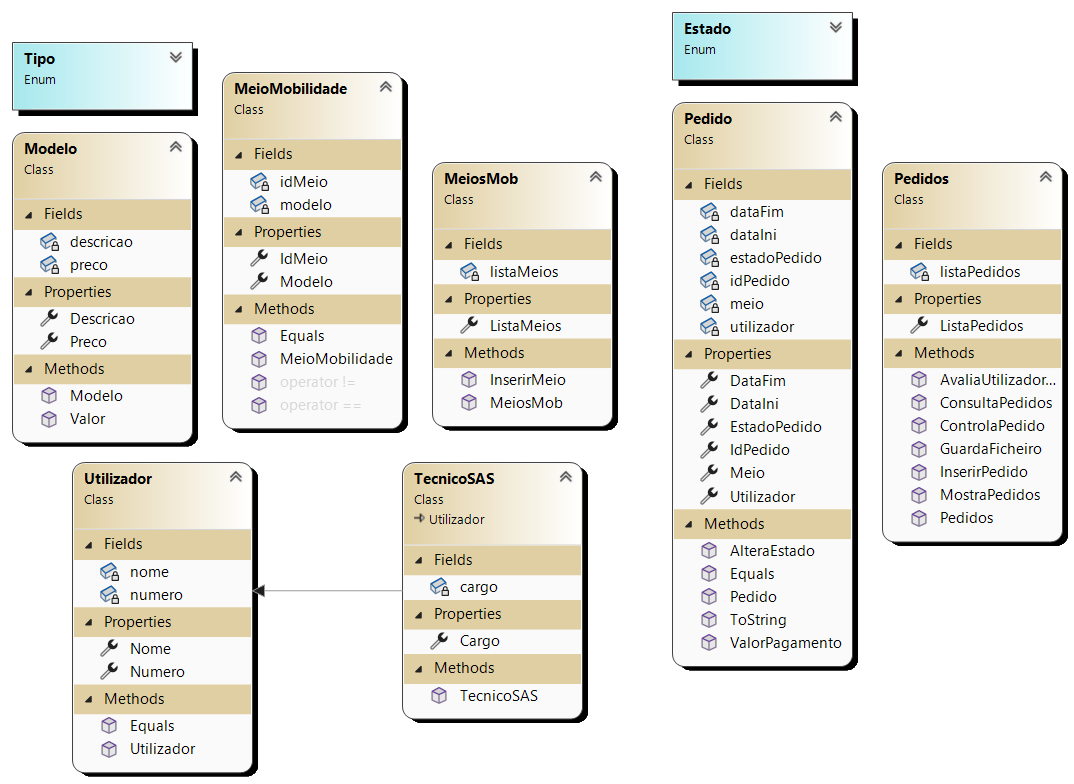


Figura 1 - Diagrama de classes. São apresentadas as classes desenvolvidas e os respetivos atributos, propriedades e métodos.

# 4. Implementação essencial das classes

## 4.1. Utilizador

Uma imagem com texto

Descrição gerada automaticamenteTal como se pode verificar na figura 1, a classe Utilizador tem os atributos nome e número, para os quais foram desenvolvidas as propriedades adequadas, assim como o construtor, apresentado na figura 2. O construtor recebe como parâmetros as variáveis necessárias para criar um objeto.

Figura 2 – Atributos e construtor da classe Utilizador.

Nesta classe foi também redefinido e método Equals, no qual é convencionado que dois utilizadores são iguais no caso de terem o mesmo número (figura 3).

Uma imagem com texto

Descrição gerada automaticamente

Figura 3 - Redefinição do método Equals para a classe Utilizador.

## 4.2. Utilizadores

Nesta classe são agrupados todos os utilizadores que forem criados. Os objetos do tipo Utilizador são adicionados à lista através do método InserirUtilizador(). Os utilizadores podem ainda ser adicionados através de um ficheiro, utilizando a operação apresentada na figura 4. Neste caso os dados são inicialmente guardados num *array* de strings, sendo distinguidos através do caracter “,”. De seguida, faz-se a correspondência entre a posição do *array* e o atributo.

Uma imagem com texto

Descrição gerada automaticamente

Figura 4 - Função que permite criar utilizadores a partir de um ficheiro e adiciona à lista.

## 4.3. Modelo

A classe Modelo define quais os modelos de meios de mobilidade que existem no sistema. Os diferentes modelos que podem existir foram definidos com recurso a enumerador, tal como se pode verificar na figura seguinte – bicicleta, bicicleta elétrica e trotinete elétrica.

Uma imagem com texto

Descrição gerada automaticamente

Figura 5 - Utilização de enum para definir os tipos de modelo possíveis.

Também nesta classe é definido o valor, visto que varia de acordo com o modelo. A função apresentada de seguida (figura 6) devolve o custo por hora de cada modelo. A partir desta será calculado o valor do pedido, de acordo com a sua duração.

Uma imagem com texto

Descrição gerada automaticamente

Figura 6 - Função que devolve o valor de acordo com o modelo escolhido.

## 4.4. MeioMobilidade

A classe do meio de mobilidade tem como atributos o id do meio de mobilidade e o modelo. Foi definido o construtor, que recebe como parâmero os dois atributos, e as respetivas propriedades. O método Equals foi aqui também reescrito, indicando que meios com o mesmo valor de idMeio são iguais. Nesta classe procedeu-se também à redefinição do operador de igualdade “==” e, consequentemente do de diferença “!=”. O primeiro foi construído com base no método Equals, tal como se observa na figura 6.

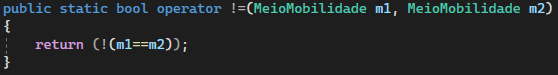
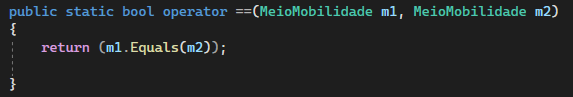


Figura 7 - Redefinição dos operadores de igualdade.

## 4.5. MeiosMob

A classe MeiosMob foi criada para possibilitar o armazenamento de todos os meios de mobilidade existentes no sistema. Esta classe tem apenas um atributo, uma lista (List) que recebe objetos do tipo MeioMobilidade. Foi desenvolvido o método InserirMeio(), apresentado na figura 7, que permite adicionar novos meios de mobilidade à lista. Nesta operação foi utilizado o método Contains() que verifica se o meio já existe através do Equals definido na classe MeioMobilidade.

Uma imagem com texto

Descrição gerada automaticamente

Figura 8 - Método desenvolvido para inserir um meio na lista de meios de mobilidade.

## 4.6. Pedido

Uma imagem com texto

Descrição gerada automaticamente Os atributos da classe Pedido são o idPedido, utilizador, meio, data de início (dataIni), data de fim (dataFim) e estado do pedido. O estado do pedido indica se está ativo ou cancelado, através de enumerador. Nesta classe foi reescrita a função Equals, definindo-se que dois pedidos são iguais se estiverem associados ao mesmo utilizador e ao mesmo meio com a mesma data de início (figura 9).

Figura 9 - Reescrita do método Equals que permite comprar pedidos.

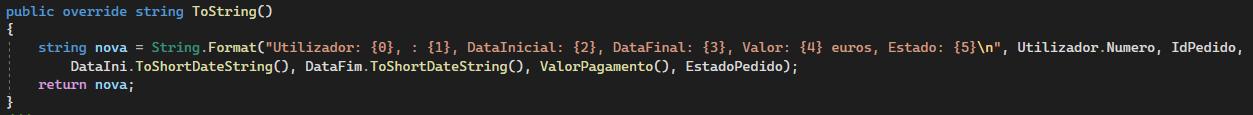
De forma a apresentar a informação de cada pedido de uma forma estruturada foi reescrito o método ToString, tal como mostra a figura 10. Neste caso, quando esta função for invocada serão apresentados na consola todos os dados referentes ao pedido.

Figura 10 - Reescrita da função ToString().

Ainda nesta classe foi desenvolvido um método para determinar o valor de cada pedido. Neste método, a variável “valor” guarda o custo que é devolvido pela função Valor() da classe Modelo (figura 8) e de seguida determina a quantidade de horas que o meio está requisitado. O custo multiplicado pela duração do pedido, resulta no valor de pagamento do pedido. Na figura anterior é possível observar um exemplo de utilização desta operação.

Uma imagem com texto

Descrição gerada automaticamente

Figura 11 - Operação que devolve o valor do pedido.

## 

Esta classe herda da interface IComparable, que é utilizado para comparar objetos. Esta interface implica a implementação do método CompareTo. Este método devolve um inteiro:

* 0 – os objetos são iguais;
* Maior que zero – se o objeto que entrar como parâmetro for menor do que o atual;
* Menor que zero – se o objeto que entrar como parâmetro for maior que o atual.

Na figura 12 está apresentado o método implementado em Pedido, onde se verifica que os pedidos são comparados pelo valor do ID.

## Uma imagem com texto Descrição gerada automaticamente

Figura 12 - Método que permite comparar dois pedidos.

## 4.7. Pedidos

À semelhança da classe MeiosMob, a classe Pedidos tem como atributo apenas uma lista. Esta lista permite armazenar objetos do tipo Pedido, ou seja, armazena todos os pedidos válidos efetuados. Na figura 13 está apresentado o construtor desta classe.

Uma imagem com texto

Descrição gerada automaticamente

Figura 13 - Construtor da classe Pedidos.

A função ControlaPedido() foi desenvolvida de forma a validar se o meio escolhido num novo pedido está disponível. Inicialmente é criada uma lista que guarda apenas os pedidos ativos que utilizam o mesmo meio que o novo pedido. De seguida, verifica-se se nesta lista existe algum pedido cujas datas coincidam com a do novo pedido. Caso se verifique alguma das condições, este método retorna *false*.

Uma imagem com texto

Descrição gerada automaticamente

Figura 14 - Função que avalia se o meio de mobilidade está disponível nas datas pedidas.

Além da validação anterior, é necessário garantir que um utilizador só tem um pedido num intervalo de tempo. Ou seja, apenas pode ter um meio de mobilidade associado a ele em cada momento. Desta forma, foi desenvolvido o método AvaliaUtilizadorPedido(). Também aqui é gerada uma lista com todos os pedidos ativos do utilizador. Posteriormente a data dos pedidos é comparada e caso ocorra sobreposição, a função devolve *false* (figura 15).

Uma imagem com texto

Descrição gerada automaticamente

Figura 15 - Método que permite verificar se um utilizador tem apenas um pedido em cada instante de tempo.

Uma imagem com texto

Descrição gerada automaticamenteO método InserirPedido(), apresentado na figura 16, permite adicionar pedidos à lista de Pedidos. Este método invoca as função ControlaPedido() e AvaliaUtilizadorPedido(). Caso estas devolvam *true* e o pedido ainda não exista, então é adicionado na lista.

Figura 16 - Método que permite adicionar um pedido a uma lista de pedidos.

O método ConsultarPedidos() permite mostrar na consola todos os pedidos de um utilizador. Este método retorna uma lista com todos os pedidos associados ao utilizador que estra como parâmetro.

Uma imagem com texto

Descrição gerada automaticamente

Figura 17 - Método que permite consultar os pedidos de um utilizador.

PorDe forma a ordenar a apresentar os pedidos ordenados, utilizou-se a função Sort(), que recorre ao método CompareTo() definido na classe Pedido. A lista é percorrida e a informação de cada objeto é apresentada.

Uma imagem com texto

Descrição gerada automaticamente

Figura 18 - Método que mostra todos os pedidos de uma lista.

# 5. Implementação final das classes

De forma a organizar adequadamente as classes foi tido como base o *Design Pattern* NTier. Este modelo permite estruturar o código por camadas:

* *Pesentation Layer* (PL) – interação com o utilizador;
* *Business Layer* (BL) – validação das regras de negócio;
* *Data Layer* (DL) – armazenamento e manipulação de dados;
* *Business Object (BO) –* conjunto de classes acessíveis às restantes camadas.

Seguindo a arquitetura anterior as classes foram então separadas de acordo com o seguinte esquema (figura 18).

Uma imagem com texto

Descrição gerada automaticamente

Figura 19 - Separação por camadas.

Na camada de dados (DL) ficam as classes que agrupam todos os dados existentes. Esta camada apenas pode ser acedida pela BL. Os atributos e métodos destas classes são estáticos, dado que se pretende que apenas seja criada uma lista de cada tipo e os objetos adicionados a essa mesma lista. Na camada BO estão todas as classes que permitem criar objetos, visto que estes têm que estar acessiveis para ser utilizados pelas restantes camadas. Na camada BL foram implementados métodos que permitem apresentar os dados da camada DL, solicitados pela camada PL. Na camada PL foi efetuados testes para verificar o correto funcionamento dos métodos desenvolvidos, nomeadamente a criação de utilizadores, meios de mobilidade e pedidos e inserção das respetivas listas de acordo com as regras estabelecidas (Pedidos).

# 6. Conclusão

A realização deste projeto permitiu aplicar os conhecimentos adquiridos na UC de POO. Além disso, possibilitou desenvolver as capacidades de programação em linguagem C# e um maior entendimento das suas funcionalidades.

Numa primeira fase de desenvolvimento do projeto, foram consolidados os conceitos de classes e objetos, assim como de construtores e propriedades. Os pilares de programação orientada a objetos foram igualmente explorados. A redefinição de operadores e a reescrita de métodos base também foram utilizados no desenvolvimento deste projeto. A utilização de ficheiros, assim como a definição de exceções foram brevemente exploradas.

A segunda fase do projeto prendeu-se com a organização do código, nomeadamente através da arquitetura NTier. Uma separação das classes por camadas, permitiu um maior entendimento da responsabilidade que compete a cada camada, assim como da utilização de membros estáticos.