
# Relatório de Projeto de Programação Avançada

**ÉPOCA RECURSO**
Licenciatura em Engenharia Informática

**ORIENTADOR**
Professor André Sanguinetti

**GRUPO DE TRABALHO**
Guilherme Bernardino Nº202001870
João Serra Nº 201400713
Inês Palet Nº 201701984
José Gonzalez Nº 201701770

fevereiro de 2023

# Índice

# Índice de Figuras



# Índice de Tabelas

# 1. Introdução

O objetivo deste projeto, em contexto de Programação Avançada e Orientada a Objetos, em linguagem JAVA, é a cração, visualização e interação de um rede de paragens de autocarro. A forma de mostrar esta rede é através da implementação de grafos (ou dígrafos). No caso deste projeto, foi pedido a implementação da rede com base num dígrafo (grafo orientado), em que cada vértice no grafo (Vertex) é colocado como vértices de saída, ou partida, e grupos de arestas (Edge) identificadas com os correspondetes vértices de chegada. A importação de datasets (grupos de dados) como, por exemplo, o dataset correspondente à Europa, consiste num grupo de Stops (paragens) e de Routes (rotas), que são respetivamente, os tipos de dados armazenados em vértices e arestas. Para a criação e visualização destes dados, foi pedido a iteração pelos ficheiros dataset (JSON), e a inserção dos vértices e arestas através das classes de leitura. Os dados de entrada, Stops e Routes, consistem de valores reais de paragens e rotas, e portanto, seria importante a colocação de mapas geográficos como plano de fundo para contextualização, algo também pedido e implementado.

Neste projeto foi utilizado a livraria JavaFXSmartGraph, útil para a visualização dos grafos, GSON para a importação e exportação de grafos, e Itext para a criação de bilhetes em forma de ficheiros PDF. O projeto em si é extenso, com várias partes a funcionar, mas consiste numa boa divisão de classes e funcionalidades.

Ao longo do projeto foi efetuado refactoring do código, de maneira a mantê-lo mais limpo e acessível e também envolveu muita documentação utilizando Javadoc.

# 2. Tipos Abstratos de Dados

## 2.1. ADT Digraph

O ADT Digraph, denomindado de grafo orientado, existe no projeto como forma de implementação de digrafos. Um grafo orientado, constituído de vértices e arestas, que no caso seria representados por paragens (stops) e rotas (rotas). Como é um grafo orientado, os as rotas são unidirecionais, isto é, entre dois vértices no grafo, numa rota existiram o vértice de partida, o que "saí" (outbound) e um vértice de chegada, o que "entra" (inbound). Na interface, ao oposto da implementação de grafos não orientados, é necessário estabelecer métodos de incindentEdges num vértice e outboundEdges, visto que a métrica sobre o que é uma aresta incindente muda neste tipo de ADT. Os métodos de inserção de ambos os tipos de dados é diferente, visto que é preciso ter atenção que uma rota vai depender do vértice de inicio e de partida corretos. Ainda a acrescentar, métodos como o areAdjacent, que retorna um boolean consoante a adjacência de vértices, é parametrizado de diferente forma, visto que um vértice adjacente num digrafo é contabilizado de forma diferente.

## 2.2. Digraph Adjacency Matrix

Para este projeto, foi nos pedido uma implementação de um ADT Digraph através da utilização de uma matriz de adjacência. Este tipo de coleção é caracterízado pela existência de uma matriz. Esta matriz é feita através da criação de um Map (HashMap) em que as chaves (keys) são os vértices de partida pertencentes ao grafo (outbound), e os valores associados a cada chave são mapas de pares vértices de chegada e correspondente aresta. Ou seja, cada vez que se faz, por exemplo, uma inserção, é tido em conta o vértice de partida, e colocado uma aresta correspondente ao vértice de chegada.

## 2.3. Stops e Route

Os dados extraídos através da funcionalidade da classe Reader seriam decompostos em dois tipos de dados a armazenar: os Stops e as Routes. Os stops são armazenados como vértices do grafo, constituídos por StopCode, StopName, Longitude, Latitude e coordenadas X e Y (para colocação visual do grafo no aplicação). As rotas são como arestas do grafo, e são constituidas por StopCodeStart (paragem de partida), StopCodeEnd (paragem de chegada), duração da rota e distância da rota. É importante assentuar que todos este dados pertencem um conjunto de dados (dataset), e que contêm os grupos de stops e routes para cada.

# 3. Padrões de Software

## 3.1. MVC

O padrão MVC, **M**odel-**V**iew-**C**ontroller, serve nomeadamente como comportamento organizacional entre classes que representem uma interface, um controlador de eventos e um modelo de dados. Este padrão serve para a separação e organizacão de funcionalidades correspondentes a cada um destes tipos, o que ajuda na leitura de código, e na compreensão das relações de cada funcionalidade, e também na manutenção e identificação de erros. A criação deste padrão é justificado pois a aplicação consiste numa interface gráfica com o utilizador e o tratamento de dados.

O model corresponde com os dados persistentes a serem apresentados, e preocupa-se com operações que possam manipular estes dados neste objeto. Ao modelo nada lhe interessa o que existe na view, pois este apenas se preocupa que alteração de dados. No projeto, a classe BusModel, é o model em questão, e resolve coisas como criar um grafo através da injeção dados dos datasets.

A view corresponde à interface gráfica do utilizador, algo que se conseque visualmente interagir, em que dispara ações do Controlador e define dados do Model, em que em run-time, ou em tempo real, alterar a visualização de dados consoante alterações pelo utilizador.

O controller é a correspondência entre a view e o model, um middle man por assim dizer, que se responsabiliza pela transferência de dados entre ambas as partes, e apenas usa sinais como método de passagem de dados, como botões na interface gráfica, objetos que lhe interessam para disparar estes sinais, porém os dados no modelo não utiliza para estas ações. No projeto, o BusController pega em dados passados pelo utilizador e realiza eventos consoante o que lhe pedem.

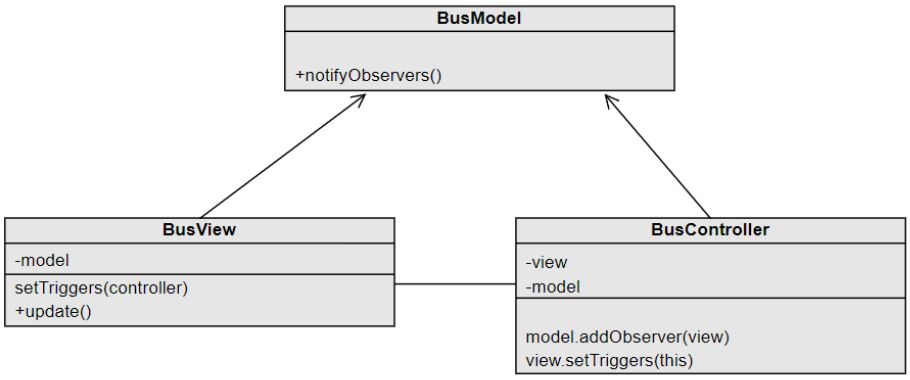


*Figura 1 – MVC Pattern Diagram*

## 3.2. Observer

O padrão Observer serve para atualização de objetos, estes mesmos alertados automaticamente cada vez que existe uma mudança de dados, ou seja, em termos informáticos, o Observador fica à escuta de possiveis mudanças no Observável, objeto que tem acesso a dados e muda o seu estado ou sujeito. No contexto deste projeto, este padrão é utilizado em conjunção com o padrão MVC, em que a view, ou por outras palavras, a interface do utilizador, atualiza o seu contéudo consoante os eventos lançados pelo utilizador, em que estes eventos cada vez que há uma alteração no grafo, o objeto observável, neste caso o model, atualiza os conteúdos, por exemplo, número de stops da interface se for adicionado / removido um vértice.

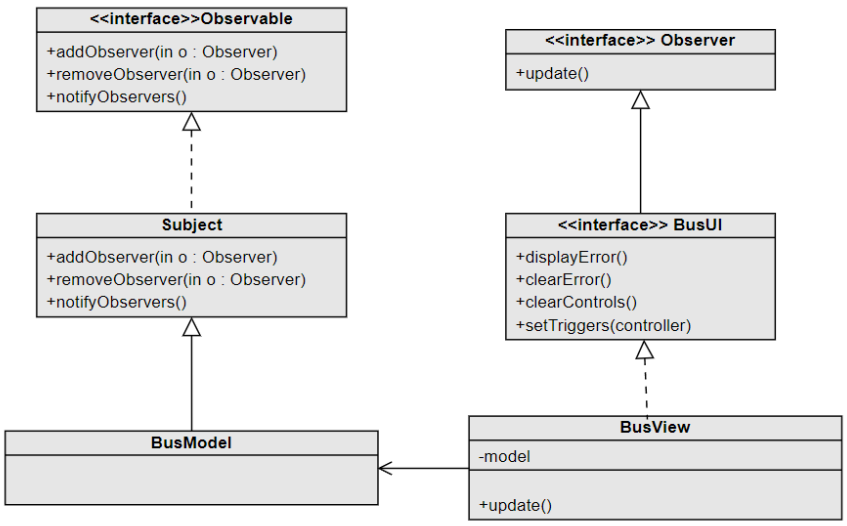


*Figura 2 – Observer Pattern Diagram*

## 3.3. Strategy

O padrão Strategy serve para a utilização de várias estratégias de comportamento que um objeto possa ter, ou seja, que possa ter para o mesmo tipo de funcionalidade, dois ou mais tipos de comportamento. No caso deste projeto, foi utilizado na seleção dos ficheiros de dados, o caso da região Ibérica e Europeia, dois grupos distintos que seriam lidos consoante a seleção de cada um. Foi também utilizado como estratégia para calcular um caminho mais curto entre um par de stops, em que utilizava duas métricas diferentes: a distância e a duração de rotas.

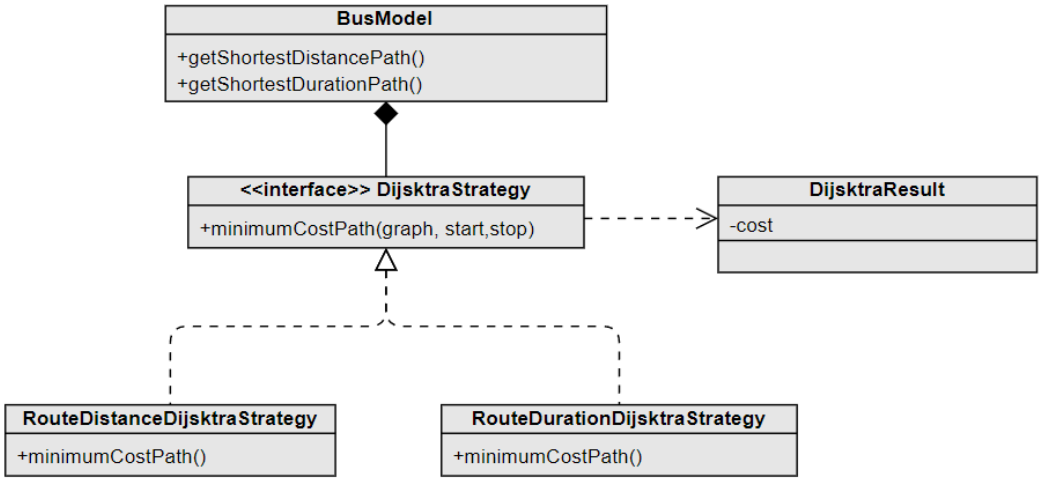


*Figura 3 - Strategy Pattern Diagram*

## 3.4. Memento

O padrão Memento, um padrão utilizado para o armazenamento de vários estados de um objeto, guardados através de mementos, objetos que guardam estes estados, e podem ser utilizados, através de Caretaker, para serem inseridos novos estados ou selecionar estados anteriores, numa forma de poder fazer rollback ao objeto em questão. No projeto é utilizado para poder fazer Undo ao algo que o utilizador faça como alteração no grafo, por exemplo, alterar a distância e duração para o estado anterior.

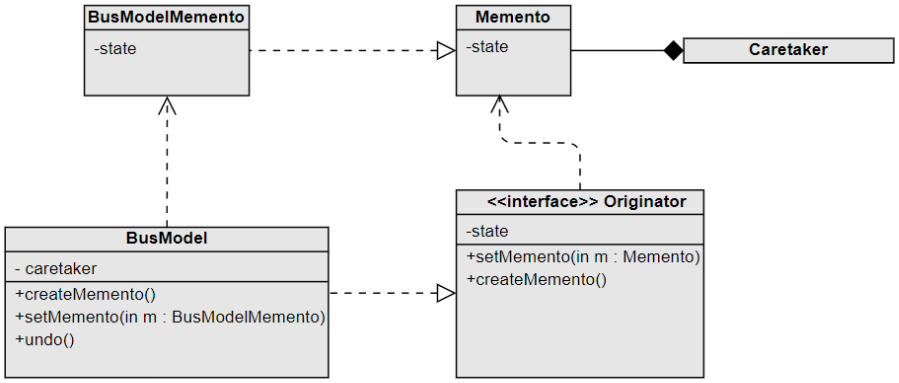


*Figura 4 - Memento Pattern Diagram*

## 3.5. Factory Method

O padrão Factory Method, um padrão criacional, que permite a uma super classe numa interface criar objetos, e que permite a sub classes alterar o tipo de objetos criados. Neste projeto é proposto criar um gerador de bilhetes, em que estes bilhetes têm dois tipos: de ida ou de ida-volta. Dentro destes tipos podemos criar dois formatos de bilhetes: basico ou detalhado.

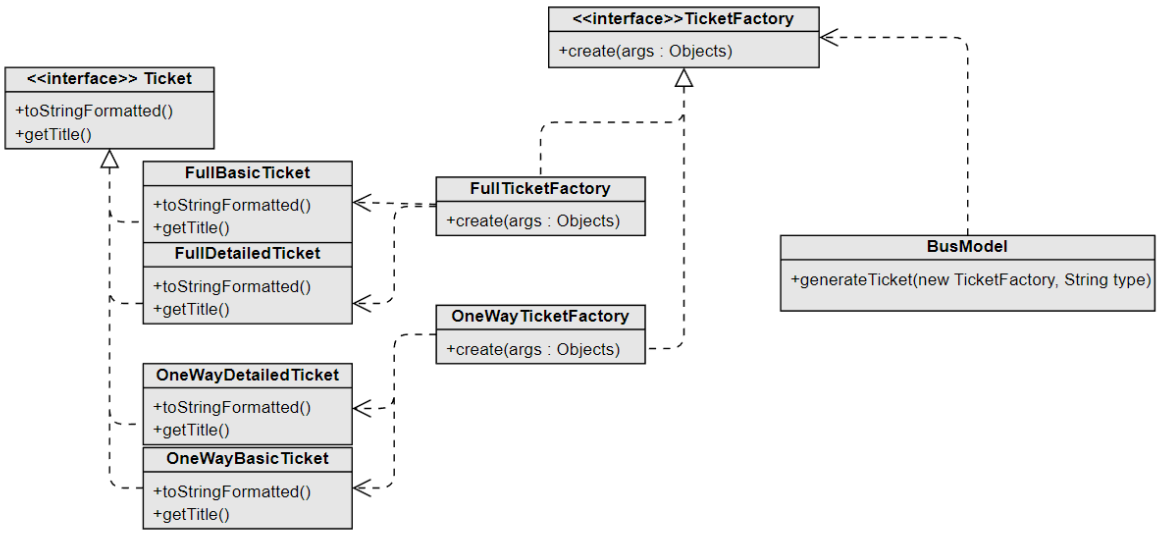


*Figura 5 - Factory Method Pattern Diagram*

# 4. Refactoring

## 4.1. Tabela de Code Smells

| Code Smell | Ocorrências | Refactoring |
|---|---|---|
| **Duplicate Code** | 9 | Extract Method |
| **Magic Number** | 3 | Creation of Constants |
| **Long Method** | 7 | Extract Method |
| **Dead Code** | 22 | Code Deletion |
| **Large Class** | 2 | Extract Class |
| **Data Class** | 3 | Move Method or Encapsulate Field |

*Tabela 1 - Code Smells*

## 4.2. Duplicate Code

Este code smell corresponde à existência de código duplicado, ou seja, código que está escrito múltiplas vezes na mesma classe. Para resolver isto basta usar o método Extract Method, que cria um novo método com esse pedaço de código, que depois é só evocar.

No exemplo do projeto, dentro de um método criava-mos labels para a interface, porém estas mesmas tinham o mesmo código de colocar o estilo e a fonte repetido várias vezes.

```
Label lblStats = new Label ( text: "Statistics");
lblStats.setStyle("-fx-font-weight: bold;");
lblStats.setFont(new Font(FONT_SIZE));

Label lblActions = new Label( text: "Actions");
lblActions.setStyle("-fx-font-weight: bold;");
lblActions.setFont(new Font(FONT_SIZE));

Label lblDatasets = new Label( text: "Maps");
lblDatasets.setStyle("-fx-font-weight: bold;");
lblDatasets.setFont(new Font(FONT_SIZE));
```

*Figura 6 - Exemplo Duplicate Code*

Portanto o que se fez foi criar um novo método extraindo o código que existe e colocado a label desejada como parâmetro.

```
Label lblStats = new Label ( text: "Statistics");
setLabelStyles(lblStats);

Label lblActions = new Label( text: "Actions");
setLabelStyles(lblActions);

Label lblDatasets = new Label( text: "Maps");
setLabelStyles(lblDatasets);
```

*Figura 7 - Exemplo Refactoring do Duplicate Code*

```
private void setLabelStyles(Label label){
    label.setStyle("-fx-font-weight: bold;");
    label.setFont(new Font(FONT_SIZE));
}
```

*Figura 8 - Exemplo 2 Refactoring do Duplicate Code*

## 4.3. Magic Number

O Magic Number ocorre quando existe um valor numérico no código, excepto 0, que tenha um significado, por exemplo, 3.14, o valor de Pi, porém não está identificado por uma constante, pois pode se desconhecer pela parte do programador.

No exemplo abaixo, é criado um método para iterador sobre a lista de tuplos das paragens, as top 10 mais centrais.

```
public List<Map.Entry<Stop, Integer>> getTop10Tuples(){
    List<Map.Entry<Stop, Integer>> list = new ArrayList<>();
    for(int i = 0; i < 10; i++){
        list.add(getTuples().get(i));
    }

    return list;
}
```

*Figura 9 - Exemplo Magic Number*

Para resolver bastou criar-se a constante TOP10, e atribuir o valor 10.

```java
public List<Map.Entry<Stop, Integer>> getTop10Tuples(){
    List<Map.Entry<Stop, Integer>> list = new ArrayList<>();
    for(int i = 0; i < TOP10; i++){
        list.add(getTuples().get(i));
    }

    return list;
}
```

*Figura 10 - Exemplo Refactoring do Magic Number*

## 4.4. Long Method

Este code smell ocorre quando temos um método que faz muitas coisa dentro, ou seja, um método deve manter-se curto e direto ao assunto. Para isto basta utilizar o Extract Method, pois ajuda a separar e criar um novo método que pode vir a ser utilizado em separado.

No exemplo do projeto, na classe BusView, tinha-mos um método para inicializar o grafo, e este mesmo utilizava bastantes partes da criação da interface desnecessariamente acumulados.

```java
public void initGraphDisplay() {

    currentBackGroundImage = model.changeBackgroundImage(0);

    this.scene = new Scene(createStackPane(), width: 1680, height: 850);
    this.stage = new Stage(StageStyle.DECORATED);
    this.stage.setTitle("Bus Network");
    this.stage.setResizable(false);
    this.stage.setScene(this.scene);
    this.stage.show();

    BackgroundImage bImg = new BackgroundImage(currentBackGroundImage,
            BackgroundRepeat.NO_REPEAT,
            BackgroundRepeat.NO_REPEAT,
            BackgroundPosition.DEFAULT,
            BackgroundSize.DEFAULT);
    Background bGround = new Background(bImg);

    graphPanel.backgroundProperty().set(bGround);

    graphPanel.init();
    System.out.println(model.getMap());
    graphPanel.updateAndWait();

    for (Vertex<Stop> h : model.getMap().vertices()) {
        graphPanel.setVertexPosition(h, h.element().getX(), h.element().getY());
    }

    update(model, arg: null);
}
```

*Figura 11 - Exemplo de Long Method*

Como existem partes deste código que poderiam vir a ser utilizados noutra parte do código, que por acaso foram, e como este método em si já seria demasiado extenso, foram criados os métodos updateBackgroundImage(), que resolve a parte de colocar o uma imagem de fundo na scene, e setVertexPosition(), que resolve o a parte do código para iterar a lista de vértices e colocados nas coordenadas específicas.

```
public void initGraphDisplay() {

    currentBackGroundImage = model.changeBackgroundImage(0);

    this.scene = new Scene(createStackPane(), width: 1680, height: 850);
    this.stage = new Stage(StageStyle.DECORATED);
    this.stage.setTitle("Bus Network");
    this.stage.setResizable(false);
    this.stage.setScene(this.scene);
    this.stage.show();

    updateBackgroundImage();

    graphPanel.init();
    System.out.println(model.getMap());
    graphPanel.updateAndWait();

    setVertexPosition();
    setStyles();

    update(model, arg: null);
}
```

*Figura 12 - Exemplo de Refactoring de Long Method*

## 4.5. Dead Code

Este bad smell ocorre quando temos um pedaço de código que não é utilizado, ou seja, código dispensável. Para resolver basta eliminar o código. No projeto, por exemplo, um método pertencente ao BusModel, para obter a lista de imagens, porém este código entrou rapidamente em desuso, devido à desnecessidade de obter as imagens desta forma.

```
public List<Image> getImages(){
    return images;
}
```

*Figura 13 - Exemplo de Dead Code*

## 4.6. Large Class

Um code smell que corresponde à existência de uma classe demasiado extensa, em que a probabilidade de existir uma subclasse separada é melhor. Para resolver, como dito, basta fazer o Extract Class, para extrair um pedaço de código para uma nova classe.

Por exemplo, no projeto é criado as classes Reader e Writer, em que possuem uma subclasse de DataList, uma subclasse que serve para guardar os dados extraidos e importados e transformá-los em Strings e vice-versa.

```
private static class DataStops {
    1 usage
    private List<StopData> stops = new ArrayList<>();
    1 usage
    private List<RouteData> routes = new ArrayList<>();
    new *
    public List<StopData> getStops() { return stops; }
    new *
    public List<RouteData> getRoutes() { return routes; }
```

*Figura 14 - Exemplo de Large Class*

Para resolver, e visto que é extenso, foi criada uma classe nova, DataList, que agora aparece em separada, e extraiu-se o código das subclasses para esta nova classe.

```
public class DataList {
    3 usages
    private List<StopData> stops;

    3 usages
    private List<RouteData> routes;

    6 usages  Guilherme-Bernardino
    public DataList(){
        this.stops = new ArrayList<>();
        this.routes = new ArrayList<>();
    }

    Guilherme-Bernardino
    public List<StopData> getStops() { return stops; }

    Guilherme-Bernardino
    public void setStops(List<StopData> stops) { this.stops = stops; }

    Guilherme-Bernardino
    public List<RouteData> getRoutes() { return routes; }

    Guilherme-Bernardino
    public void setRoutes(List<RouteData> routes) { this.routes = routes; }
```

*Figura 15 - Exemplo de Refactoring de Large Class*

## 4.7. Data Class

Uma Data Class é quando uma classe é unicamente composta de atributos e getters e setters. Para resolver basta utilizar o Encapsulate Field.

No projeto, pegando no exemplo do DataList, este mesmo servia para criar StopData e RouteData, porém estas subclasses estavam a ser acedidas diretamente, portanto criou-se um método que retorna uma nova StopData ou um novo RouteData object, consoante os parâmetros passados.

```
public StopData createStopData(String stopCode, String stopName, String lat, String lon,String x, String y){
    return new StopData( stopCode,  stopName,  lat,  lon, x,  y);
}
```

*Figura 16 - Exemplo de Refactoring de Data Class*

```
public RouteData createRouteData(String stopCodeStart, String stopCodeEnd, String distance, String duration){
    return new RouteData( stopCodeStart,  stopCodeEnd,  distance,  duration);
}
```

*Figura 17 - Exemplo 2 de Refactoring de Data Class*

Após a criação destes métodos, a classe DataList passa a não ser uma Data Class, e pode delegar novos objetos consoante o que lhe for pedido.

# 5. Conclusão

Em suma, o projeto serviu como uma boa ferramenta de aprendizagem, com uma forte incidência na compreensão de padrões de software, algo muito útil para o estabelecimento de funcionalidades atomicamente implementadas, e que servem nomeadamente para uma forte divisão de tarefas por parte dos membros do grupo de trabalho. Este projeto poderia prefeitamente ser associado a uma aplicação em contexto real, algo que junta o útil ao agradável, pois ajuda a compreender as estruturação interna e externa deste tipo de trabalho, e a melhorar o projeto em si. Uma aplicação deste género possibilitou também a conhecimento entre interação e manipulação de dados, visto que o que o utilizador vê não são os dados em si, mas sim aquilo que utiliza para manipular esses dados, a interface gráfica. O grupo porém não teve muita facilidade com a parte gráfica, pois demonstrou-se algo um pouco díficil, devido à falta de conhecimentos adquiridos previamente. Ou seja, este projeto foi um desafio interessante, criou-se e arranjou-se alternativas para poder se criar esta aplicação, e houve muitos problemas no caminho.

O conhecimento de grafos e algoritmia ajudou na criação da rede de autocarros, mais uma vez, demonstrando a importância e relevância deste tipo de tecnologia. Como este projeto teve uma abordagem real, os dados inseridos seriam paragens e rotas de autocarro, porém ao invés se implementar um grafo onde uma rota seria bidirecional, tinhas a tarefa extra de resolver este projeto com um grafo orientado, algo que provou ser desafiante, porém acessível.

Também foi interessante a implementação dos padrões de software, como MVC, que constava da separação do modelo de dados com a interface de utilizador, e o padrão Strategy, que permitia criar estratégias de funcionalidades, algo importante para a separação de aplicações deste tipo, pois vários objetos podem ter várias interações. Os padrões mostraram-se complicados, uma vez que obrigam ao raciocínio de uma organização de classes acertiva e funcional.

Ainda mais, como parte deste projeto, e como tarefa extra, a limpeza e refaturação do código, algo que pode parecer indiferente, porém ajuda imensamente na leitura do código, na eficiência do mesmo, e vêm com as suas vantagens e relativamente poucas desvantagens. É uma mais valia, e como foi realizado ao longo da criação deste projeto, serviu como um resumo daquilo trabalhado e futuramente pode vir a criar hábitos de boa programação. A documentação do código, também efetuada, ajudou a estabelecer melhor e a descrever melhor o que cada classe, método e atributo faz.

O projeto teve momentos altos e baixos, como este documento pode ajudar a demonstrar, mas serviu como uma base de conhecimentos para o grupo.

# 6. Bibliografia / Webgrafia

**Abstract Data Types – Wikipedia – 15/02/2023**

https://en.wikipedia.org/wiki/Abstract_data_type

**Refactoring e Code Smells – Refactoring Guru – 14/02/2023**

https://refactoring.guru/refactoring/smells

**Software Patterns – Refactoring Guru – 14/02/2023**

https://refactoring.guru/design-patterns/catalog