TECNOLOGIA SETÚBAL
ESCOLA SUPERIOR POLITÉCNICO SETÚBAL

Programação Avançada
2024/2025
Licenciatura em Engª. Informática

Professor: Luís Damas
Grupo da PL6
Nº 202300310, Rafael Quintas
Nº 202300311, Rafael Pato
Nº 202300306, Guilherme Pereira

Indíce

1. Introdução

Este projeto, desenvolvido para a disciplina de Programação Avançada, do curso de Engenharia Informática no Instituto Politécnico de Setúbal, tem como objetivo criar uma aplicação gráfica interativa para representar e explorar um sistema integrado de transportes.

A aplicação será construída em Java, seguindo os princípios fundamentais da programação e utilizando padrões de design de software para garantir a coerência, extensibilidade e clareza no código. O sistema será modelado através de um grafo, onde os vértices representam as paragens (stops) e as arestas representam as rotas (routes) que os conectam.

A aplicação permitirá a visualização interativa do grafo, cálculo de métricas relevantes e análise de percursos otimizados baseados em critérios como distância, duração e sustentabilidade. Além disso, contará com funcionalidades como seleção personalizada de percursos, visualização de custos detalhados e registro das operações realizadas pelo utilizador por meio de um Logger.

O projeto busca não apenas criar uma ferramenta útil para a análise de redes de transporte, mas também proporcionar uma experiência de utilizador intuitiva e informativa, utilizando abordagens modernas de desenvolvimento de software.

2. Tipos Abstratos de Dados e Interface Gráfica

A nossa interface gráfica foi desenvolvida utilizando a classe MapView, que, por sua vez, faz uso de um SmartGraphPanel para apresentar e localizar o grafo de forma interativa na interface.

Para que essa interface funcione corretamente, foi necessário implementar algumas classes-chave:

- **TransportMap**:

  Representa e utiliza um ADT Graph. Esta classe é responsável pela estrutura principal de dados e contém as funcionalidades disponibilizadas pela interface. Complementa o grafo, realizando verificações para garantir a consistência dos dados e a integridade das operações realizadas sobre ele.

- **Stop**:

  Representa uma paragem, que corresponde a um vértice no grafo. É responsável por armazenar todas as informações relacionadas à paragem, como código, nome, latitude e longitude.

- **Route:**

  Representa uma rota, correspondente a uma aresta no grafo. É implementada como uma lista de rotas e armazena todas as informações associadas a cada rota, como tipo de transporte, distância, duração, sustentabilidade e estado.

Além disso, foram criadas algumas classes auxiliares:

- **GenericRoute**

  Uma classe auxiliar utilizada durante a importação de dados. Armazena a lista de rotas entre dois vértices para posterior adição ao grafo.

- **DataImporter:**

  Responsável por importar os dados dos ficheiros CSV. Esta classe assegura a importação e organização de grandes volumes de dados que serão utilizados para construir e atualizar o grafo.

3. Diagrama de classes

Figura 1 – Diagrama de classes

Nota: A Figura 1, que representa o diagrama de classes, está disponível como anexo na pasta 'Ficheiros Auxiliares', pois a sua visualização não é suficientemente clara quando inserida no Word.

4. Padrões de Software

- **Model View Controller (MVC):**

  O padrão MVC tem como objetivo separar as responsabilidades de uma aplicação em três componentes principais: Model, View e Controller. Essa separação visa organizar o código, facilitar a manutenção e promover a sua reutilização.

  Neste projeto, utilizamos o padrão MVC para dividir as diferentes responsabilidades da aplicação:

  1. Model: Representado pela classe TransportMap. É responsável por armazenar os dados que serão apresentados e pelas operações que os manipulam ou transformam.
  2. View: Representado pela classe MapView. Esta classe conhece o modelo, mas não o manipula diretamente, apenas o consulta. Define como os dados serão apresentados ao utilizador e não implementa lógica de negócios.
  3. Controller: Representado pela classe TransportMapController. Esta classe sincroniza as ações do utilizador com as alterações no modelo e interage diretamente com a View para garantir que os dados sejam exibidos de forma consistente.

- **Memento**

  O padrão Memento tem como objetivo capturar e restaurar o estado interno de um objeto sem violar o princípio de encapsulamento. O seu propósito é permitir que o estado de um objeto seja salvo em um momento específico e restaurado posteriormente, sem expor os detalhes internos do objeto.

  Neste projeto, utilizamos o padrão Memento para capturar o estado do grafo da classe TransportMap, possibilitando a desativação e ativação (undo) de rotas, bem como a modificação e restauração (undo) da duração de rotas de bicicleta.

  Para isso, foram implementadas as seguintes classes:

  1. Memento: Representado pela interface Memento, que abstrai os detalhes da classe concreta de memento.
  2. Originator: Representado pela interface Originator, que define o contrato para a classe TransportMap, permitindo a criação e restauração de estados dessa classe.

3. ConcreteMemento: Representado pela classe TransportMapMemento, que guarda o estado do grafo da classe TransportMap por meio de uma deep copy.
4. Caretaker: Representado pela classe Caretaker, responsável por armazenar os mementos (estados dos objetos de TransportMap) em uma stack, de forma a possiblitar salvar e restaurar o estado do grafo conforme necessário.

- **Strategy**

  O padrão Strategy tem como objetivo definir uma família de algoritmos, encapsulá-los em classes separadas e torná-los intercambiáveis. Esse padrão é especialmente útil quando é necessário escolher, em tempo de execução, diferentes comportamentos ou estratégias, sem alterar o código da função principal.

  Neste projeto, utilizamos o padrão Strategy no cálculo do custo de uma rota. Contudo, acreditamos que sua aplicação não seja ideal para o contexto do projeto. Isso porque, embora a ideia inicial fosse aplicar diferentes algoritmos para diferentes critérios de custo (por exemplo, usar Dijkstra para os critérios de distância e duração, e Bellman-Ford para o critério de sustentabilidade), concluímos que o algoritmo de Bellman-Ford é capaz de resolver os três critérios. Assim, o uso de Dijkstra tornou-se desnecessário.

  Ainda assim, decidimos manter as diferentes estratégias representando os diferentes critérios. Essas estratégias servem principalmente para retornar o valor desejado (distância, duração ou sustentabilidade) e, ao mesmo tempo, facilitar a organização e a legibilidade do código.

  Classes implementadas para este padrão:

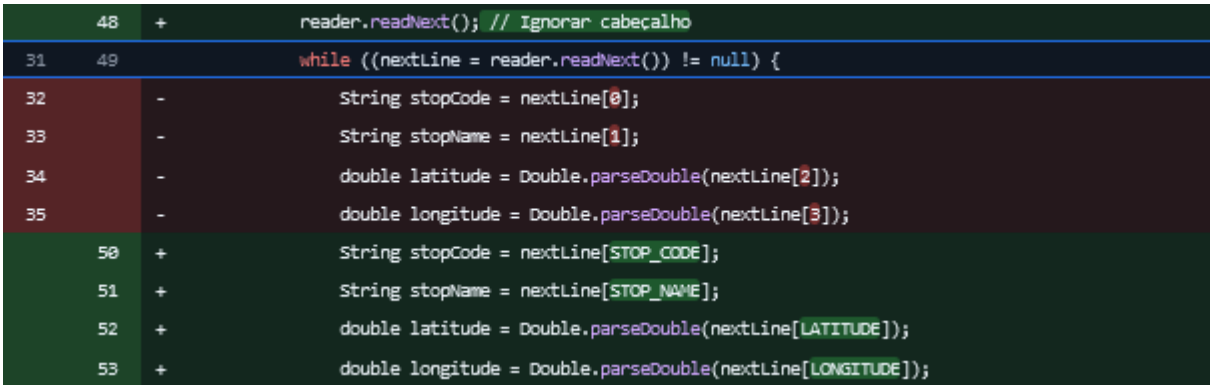
  1. Strategy: Representada pela interface WeightCalculationStrategy, que define um contrato para as diferentes estratégias, garantindo a implementação do método calculateWeight.
  2. Concrete Strategies: Representada pelas classes DistanceStrategy, DurationStrategy, SustainabilityStrategy. Estas têm a função de delegar o que cada cálculo de custo irá utilizar na sua avaliação, com a particularidade de, no caso da sustentabilidade, conter ainda uma constante (offset) que é utilizada de forma a garantir que o algoritmo seja processado de maneira desejada.

5. Refactoring

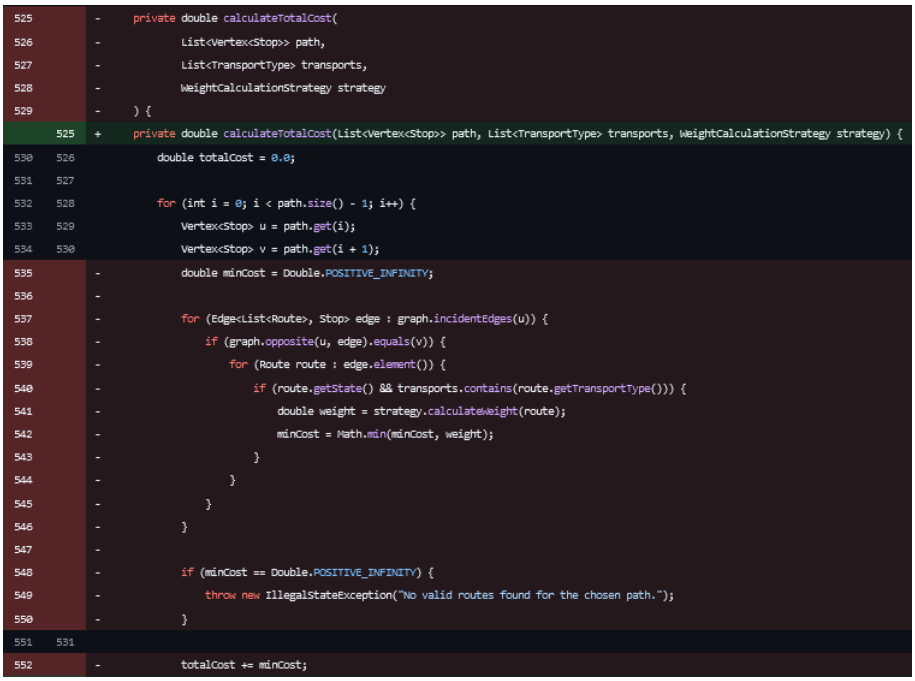
REFACTORING		
Code-Smell	**Número de Situações**	**Técnica de Refactoring**
Magic Numbers	2	Replace Magic Number with Symbolic Constant
Long Method	1	Extract Method
Dead Code	1	Remove Parameter
Inappropriate Intimacy	Ocorreu por vezes na View	Extract Class / Hide Delegate
Duplicate Code	Impossível Recordar	Extract Method
Switch Statements	1	Replace Type Code with Strategy
Large Class	1	Extract Class

Exemplos do refactoring efetuado:

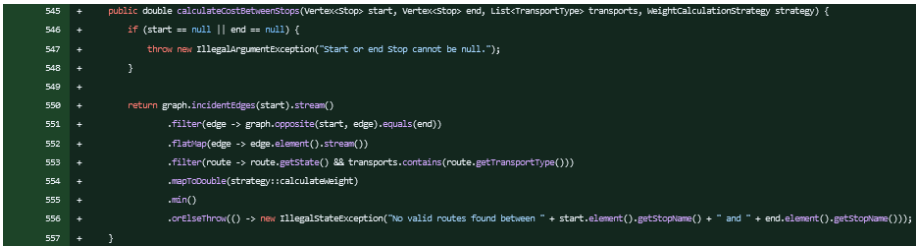
1. Code Smell: Magic Numbers



2. Code Smell: Long Method



```
525   -    private double calculateTotalCost(
526   -            List<Vertex<Stop>> path,
527   -            List<TransportType> transports,
528   -            WeightCalculationStrategy strategy
529   -    ) {
      525 +    private double calculateTotalCost(List<Vertex<Stop>> path, List<TransportType> transports, WeightCalculationStrategy strategy) {
530   526          double totalCost = 0.0;
531   527
532   528          for (int i = 0; i < path.size() - 1; i++) {
533   529              Vertex<Stop> u = path.get(i);
534   530              Vertex<Stop> v = path.get(i + 1);
535   -              double minCost = Double.POSITIVE_INFINITY;
536   -
537   -              for (Edge<List<Route>, Stop> edge : graph.incidentEdges(u)) {
538   -                  if (graph.opposite(u, edge).equals(v)) {
539   -                      for (Route route : edge.element()) {
540   -                          if (route.getState() && transports.contains(route.getTransportType())) {
541   -                              double weight = strategy.calculateWeight(route);
542   -                              minCost = Math.min(minCost, weight);
543   -                          }
544   -                      }
545   -                  }
546   -              }
547   -
548   -              if (minCost == Double.POSITIVE_INFINITY) {
549   -                  throw new IllegalStateException("No valid routes found for the chosen path.");
550   -              }
551   531
552   -              totalCost += minCost;
```



```
545 +    public double calculateCostBetweenStops(Vertex<Stop> start, Vertex<Stop> end, List<TransportType> transports, WeightCalculationStrategy strategy) {
546 +        if (start == null || end == null) {
547 +            throw new IllegalArgumentException("Start or end Stop cannot be null.");
548 +        }
549 +
550 +        return graph.incidentEdges(start).stream()
551 +                .filter(edge -> graph.opposite(start, edge).equals(end))
552 +                .flatMap(edge -> edge.element().stream())
553 +                .filter(route -> route.getState() && transports.contains(route.getTransportType()))
554 +                .mapToDouble(strategy::calculateWeight)
555 +                .min()
556 +                .orElseThrow(() -> new IllegalStateException("No valid routes found between " + start.element().getStopName() + " and " + end.element().getStopName()));
557 +    }
```

3. Code Smell: Dead Code

```java
/**
 * Desativa as rotas especificadas de uma aresta, alterando o estado das rotas para falso.
 *
 * @param edge            a aresta que contém as rotas a serem desativadas.
 * @param routesToDisable lista de rotas que serão desativadas.
 */
public void disableRoute(Edge<List<Route>, Stop> edge, List<Route> routesToDisable) {  1 usage
    for (Route route : routesToDisable) {
        route.setState(false);
    }
}
```

```java
/**
 * Desativa as rotas especificadas de uma aresta, alterando o estado das rotas para falso.
 *
 * @param routesToDisable lista de rotas que serão desativadas.
 */
public void disableRoute(List<Route> routesToDisable) {  1 usage  Rafael-Quintas *  1 related problem
    for (Route route : routesToDisable) {
        route.setState(false);
    }
}
```

4. Code Smell: Inappropriate Intimacy, Duplicate Code, Large Class

   Não possuímos exemplos deste código antes e após o refactoring, pois assim que nos apercebemos do problema, começamos a resolvê-lo. No entanto, é bastante claro que, antes da implementação do padrão MVC, havia código duplicado na View para realizar tarefas simples, além de inappropriate intimacy, pois a View acabava por vezes a manipular o TransportMap. Devido a isso, a View também se comportava como uma large class, já que estava a fazer mais do que o esperado, pois o Controller passou a ser responsável pelas interações do user.

5. Code Smell: Switch Statements

   A implementação do padrão Strategy simplifica a lógica de seleção de um critério, de forma a evitar uma grande quantidade de lógica dentro da seleção do mesmo.

6. Observações

A tabela de refactoring não apresentará os resultados mais precisos, pois, na realidade, é impossível saber exatamente quantos problemas ocorreram ao longo das três fases. Sempre procurámos melhorar o código das versões anteriores quando identificávamos possíveis problemas, e isso, obviamente, influenciou as nossas decisões durante o desenvolvimento do código, especialmente quando começámos a aprender sobre padrões e refactoring. Assim, contabilizamos apenas os code smells que nos lembrávamos e que se destacaram durante a terceira fase.

É também inegável o auxílio de ferramentas como StackOverflow e ChatGPT. Em alguns casos, como na View ou no algoritmo de cálculo de custos, seria impossível não contar com essas ferramentas, pois não conhecíamos o algoritmo de Bellman-Ford o que levou a dificuldades no seu desenvolvimento, tal como não dominamos o JavaFX, que, para nós, muitas vezes torna-se complexo.

7. Conclusão

Este projeto foi crucial para o nosso desenvolvimento como programadores. O uso de padrões de software e de boas práticas de programação fez nos refletir e explorar maneiras de otimizar o nosso código, tornando-o mais eficaz. Além disso, garantimos que o desenvolvimento do projeto fosse melhor estruturado, o que facilita a manutenção e o trabalho futuro de outras pessoas que possam interagir com o código, permitindo que aprendam também com as boas práticas adotadas.

Aprendemos a criar uma interface gráfica interativa para o utilizador, com o uso de conceitos fundamentais da programação. Isso não só proporciona uma experiência mais agradável para o utilizador, mas também facilita o trabalho dos programadores. Concluímos que este projeto teve um grande impacto na maneira como, no início, víamos a programação, e como, ao final, adquirimos uma nova visão sobre como programar.