Relatório

Projeto - Programação Avançada

Licenciatura em Engenharia Informática

Docente Orientador: Luís Damas

Grupo: Diogo Letras - Nº 202002529 - Turma: 2ºL_EI-SW-06
Miguel Vicente - Nº 202000563 - Turma: 2ºL_EI-SW-06
Pedro Cunha - Nº 202000757 - Turma: 2ºL_EI-SW-02
Jorge Mimoso - Nº 202000695 - Turma: 2ºL_EI-SW-03

Janeiro, 2021

Índice

Lista de figuras

Lista de tabelas

Introdução

Este projeto consiste na implementação de uma rede logística de transporte baseada num grafo não-orientado.

Para representar o problema, foi criada uma aplicação em JAVA, de uma rede logística de transporte, com recurso à teoria dos grafos, leccionada na unidade curricular de Programação Avançada.

A aplicação simula o funcionamento de uma rede logística de transporte com base no mapa dos Estados Unidos da América.

Esta rede é composta por cidades (hubs), que são vértices, e por rotas (routes), que são arestas.

Cada cidade (hub), é composta por um nº identificador, nome, população e coordenadas.

Cada rota (route), é composta por um nº identificador e pela distância.

De forma resumida, a aplicação apresenta várias métricas sobre a rede, permite calcular o caminho mais curto entre duas cidades, ver as cidades mais distantes entre si, adicionar e remover rotas.

Tipos Abstratos de Dados implementados

Neste projeto, optámos pela utilização de apenas um ADT, o ADT Graph, pois a rede é representada através de um grafo não-orientado.

O ADT Graph (junto com uma pequena aplicação) foi disponibilizado, de modo a ser usado com base numa implementação com lista de adjacências. Isto fez com que houvesse uma adaptação no tipo de implementação, pois o exemplo inicial era feito com base numa lista de arestas.

ADT Graph - Representa uma implementação de um grafo não-orientado. Este tipo abstrato de dados é constituído por vértices (Vertex) e arestas (Edge).

Este ADT contém várias operações sobre o grafo (ou neste caso, a rede logística), como por exemplo a inserção de vértices ou de arestas, a remoção de vértices ou de arestas, a substituição de vértices ou arestas, a contagem de número de arestas e de vértices, e também a consulta de informação sobre os vértices e arestas em forma de coleção.

Diagrama de classes

Imagens do diagrama de classes podem ser consultadas na pasta ‘relatorio’, para uma consulta com maior pormenor e detalhe.

Diagrama do ADT Graph (fig.1):

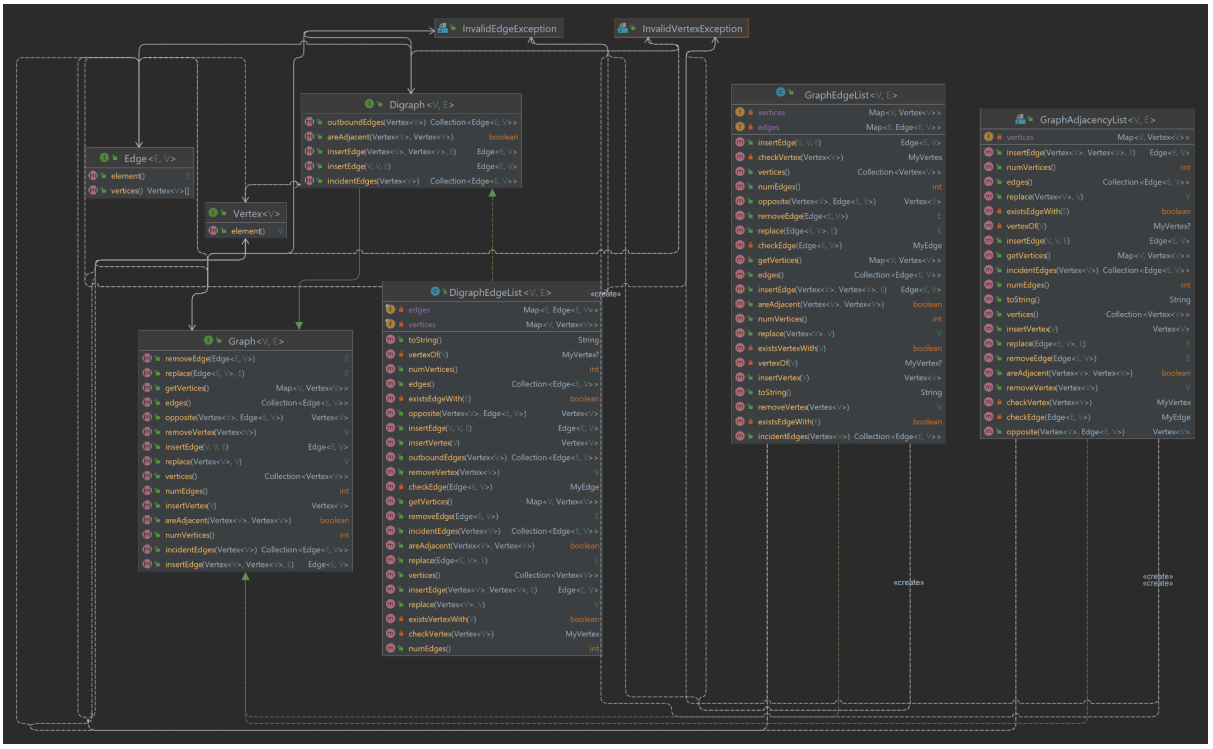
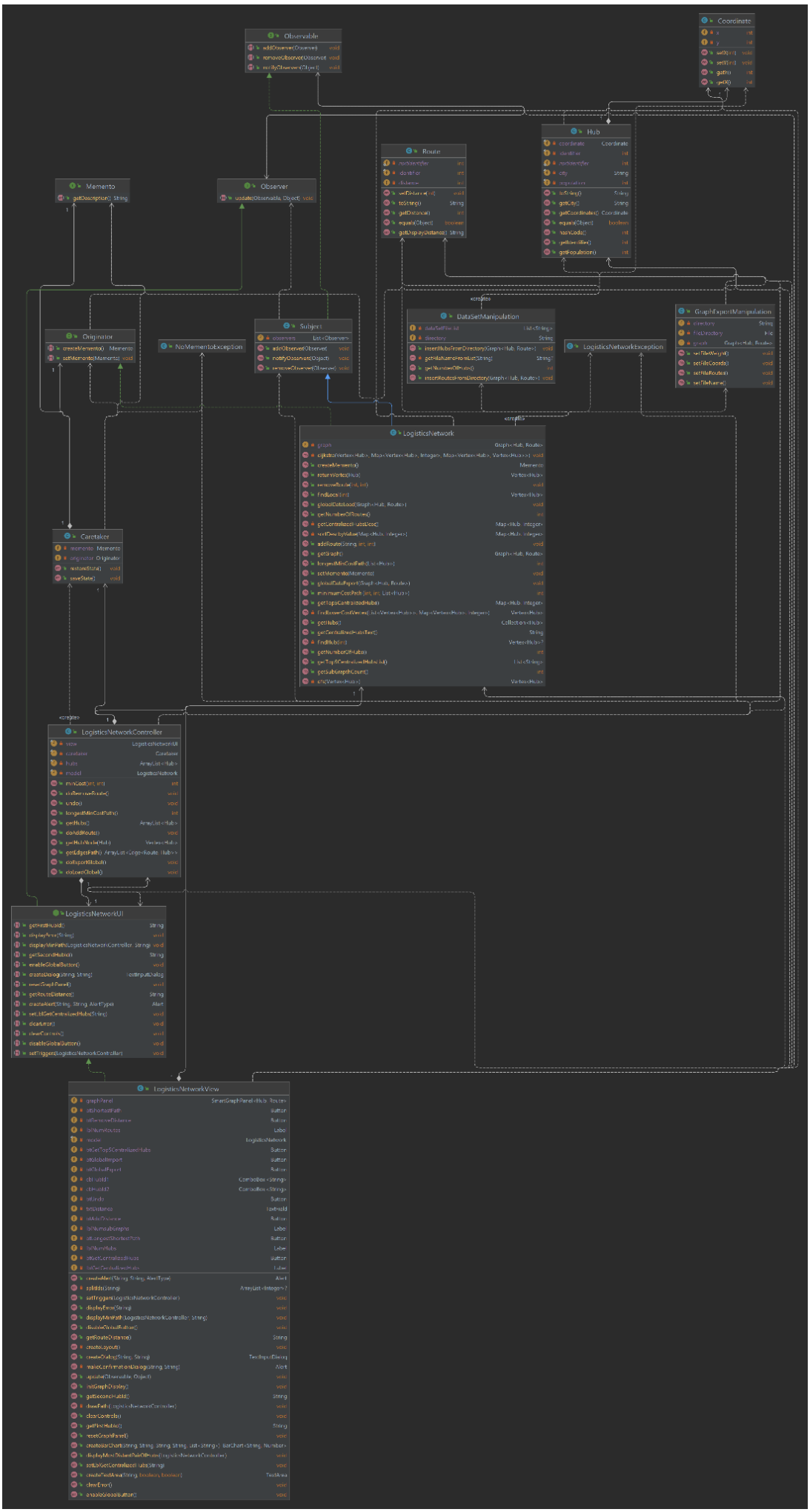


Diagrama de classes após refactoring (fig.2):


Observable
Coordinate
Hub
Route
Memento
Observer
Originator
NoMementoException
Subject
DataSetManipulation
GraphExportManipulation
LogisticsNetworkException
LogisticsNetwork
Caretaker
LogisticsNetworkController
LogisticsNetworkUI
LogisticsNetworkView

Documentação de classes e métodos

A documentação de classes e métodos foi feita através do JAVADOC. A mesma encontra-se disponível para consulta na pasta ***docs***, incluída no projeto.

Padrões de Software implementados

MVC - O padrão Model-View-Controller (MVC) tem como propósito separar as camadas da aplicação. Esta separação permite partilhar os mesmos dados pelas diferentes interfaces gráficas, tornando assim, mais fácil o desenvolvimento, testes e manutenção do código.

Participantes do padrão MVC:

Model: É responsável pela lógica de negócio. Preocupa-se com os dados persistentes que devem ser apresentados e com as operações que são aplicadas para transformar os objetos.
É representado através da classe 'LogisticsNetwork'.

View: Faz uso das operações do Model (através do Controller) para manipular e obter dados. Define como os dados são visualizados pelo utilizador e mantém a consistência na apresentação dos dados quando o Model se altera.
É representada através da classe 'LogisticsNetworkView'.

Controller: Sincroniza as ações da View com as ações realizadas pelo Model. Funciona essencialmente como intermediário entre os dois.
É representado através da classe 'LogisticsNetworkController'.

Este padrão foi portanto utilizado para criar uma divisão clara entre a interface gráfica (front-end) e o back-end da aplicação, desta forma tornando o desenvolvimento de código mais fácil e eficiente.

Observer - O padrão é utilizado para que alguns objetos consigam observar outros por um determinado tempo.
No projeto, este padrão atua essencialmente como auxílio ao padrão MVC, de modo a manter o conteúdo da View sempre atualizado.
Portanto, sempre que houver alguma alteração no model (LogisticsNetwork), a view (LogisticsNetworkView) será notificada e estará sempre atualizada.

Memento - É um padrão de design comportamental que permite salvar e restaurar o estado anterior de um objeto, sem revelar os detalhes de sua implementação.
É composto pela classe 'LogisticsNetwork' implementa a interface Originator e contém a inner class 'LogisticsNetworkMemento', que implementa a interface Memento. A classe 'Caretaker' guarda o histórico de mementos (snapshots).

Refactoring

Tabela com code smells encontrados no código do projeto.

Code smell	Nº de ocorrências	Técnica de refactoring aplicadas
Comments	1	Rename Method
Data Class	1	Não se fez nada
Dead Code	8	Delete the code
Temporary Field	3	Inline Temp
Magic Number	1	Replace Magic Number with Symbolic Constant

Tabela.1 - Code smells encontrados no projeto.

Comments - exemplo

Considera-se um code smell do tipo ‘Comments’, quando existem comentários dentro de um método, neste caso no método update da classe LogisticsNetworkView. Para tratar este code smell, é recomendado apagar os comentários e dar um nome mais sugestivo ao método. No entanto, o nome do método já é bastante sugestivo, por isso simplesmente apagou-se os comentários.

```java
public void update(Observable subject, Object arg) {
    if (subject == model) {
        /* update graph panel */
        graphPanel.updateAndWait();

        /* metrics */
        lblNumHubs.setText(model.getNumberOfHubs() + "");
        lblNumRoutes.setText(model.getNumberOfRoutes() + "");
        lblGetCentralizedHubs.setText(model.getCentralizedHubsText() + "");
        this.lblNumsubGraphs.setText(String.valueOf(model.getSubGrapthCount()));

        if (this.model.getGraph() == null) {
            this.btGetTop5CentralizedHubs.setDisable(true);
            this.btGetCentralizedHubs.setDisable(true);
            this.lblGetCentralizedHubs.setDisable(true);
            this.btShortestPath.setDisable(true);
        }

        /* update ID comboboxes */
        cbHubId1.getItems().clear();
        cbHubId2.getItems().clear();
        List<Hub> hubList = new ArrayList<>(model.getHubs());
        hubList.sort(Comparator.comparingInt(Hub::getIdentifier));
        for (Hub p : hubList) {
            cbHubId1.getItems().add(String.valueOf(p.getIdentifier()));
            cbHubId2.getItems().add(String.valueOf(p.getIdentifier()));
        }
    }
}
```

Fig.3 - Método com exemplo de code smell ‘comments’.

```java
public void update(Observable subject, Object arg) {
    if (subject == model) {
        graphPanel.updateAndWait();

        lblNumHubs.setText(model.getNumberOfHubs() + "");
        lblNumRoutes.setText(model.getNumberOfRoutes() + "");
        lblGetCentralizedHubs.setText(model.getCentralizedHubsText() + "");
        this.lblNumsubGraphs.setText(String.valueOf(model.getSubGrapthCount()));

        if (this.model.getGraph() == null) {
            this.btGetTop5CentralizedHubs.setDisable(true);
            this.btGetCentralizedHubs.setDisable(true);
            this.lblGetCentralizedHubs.setDisable(true);
            this.btShortestPath.setDisable(true);
        }

        cbHubId1.getItems().clear();
        cbHubId2.getItems().clear();
        List<Hub> hubList = new ArrayList<>(model.getHubs());
        hubList.sort(Comparator.comparingInt(Hub::getIdentifier));
        for (Hub p : hubList) {
            cbHubId1.getItems().add(String.valueOf(p.getIdentifier()));
            cbHubId2.getItems().add(String.valueOf(p.getIdentifier()));
        }
    }
}
```

Fig.4 - Método depois de efetuado o refactoring.

Data Class - exemplo

Este code smell ocorre quando uma classe basicamente apenas possui atributos, getters e setters. A classe 'Coordinate' é um exemplo.

```java
public class Coordinate {
    private int x, y;

    public Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }

    public void setX(int x) { this.x = x; }

    public int getY() { return y; }

    public void setY(int y) { this.y = y; }
}
```

Fig.5 - Exemplo de uma 'Data Class'.

Como esta classe não representa perigo, não se fez nada.

Dead Code - exemplo

Dead Code refere-se a código não utilizado. Pode acontecer no início do desenvolvimento, o programador supor que um determinado método seja útil, mas quando se chega ao fim do desenvolvimento, verifica-se que afinal esse método não era necessário.

```java
/**
 * Checks if the Graph vertices are empty.
 *
 * @return true if they are empty, otherwise false
 */
public boolean isHubsEmpty() {
    return this.graph.vertices().isEmpty();
}
```

Fig.6 - Exemplo de 'Dead Code'.

Para resolver este code smell, simplesmente apagou-se o método.

Temporary Field - exemplo

Este code smell foi encontrado na classe 'LogisticsNetworkView'. Representa um campo (neste caso três labels) vazio fora de um método.

```
private Label lblNumHubsDescription = new Label( text: "0");
private Label lblNumRoutesDescription;
private Label lblNumsubGraphsDescription;
```

Fig.7 - Exemplo de 'Temporary Fields'.

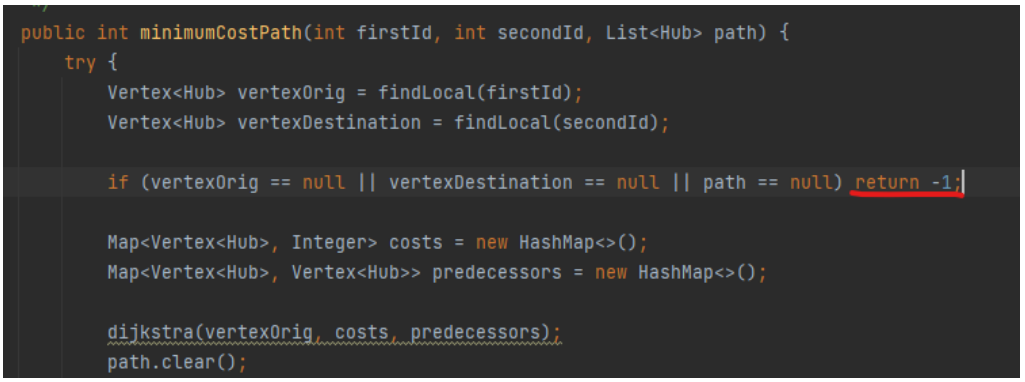
Para o resolver, usou-se a técnica Inline Temp, ou seja tanto a declaração como a inicialização das labels passaram a estar dentro do método createLayout().

```
Label lblNumHubsDescription = new Label( text: "Number of Hubs:");
lblNumHubs = new Label();
lblNumHubs.setStyle("-fx-font-weight: bold;");

Label lblNumRoutesDescription = new Label( text: "Number of Routes:");
lblNumRoutes = new Label();
lblNumRoutes.setStyle("-fx-font-weight: bold;");

Label lblNumsubGraphsDescription = new Label( text: "Number of Sub-Graphs:");
lblNumsubGraphs = new Label();
lblNumsubGraphs.setStyle("-fx-font-weight: bold;");
```

Fig.8 - Declaração e inicialização das labels dentro do método createLayout().

Magic Number - exemplo

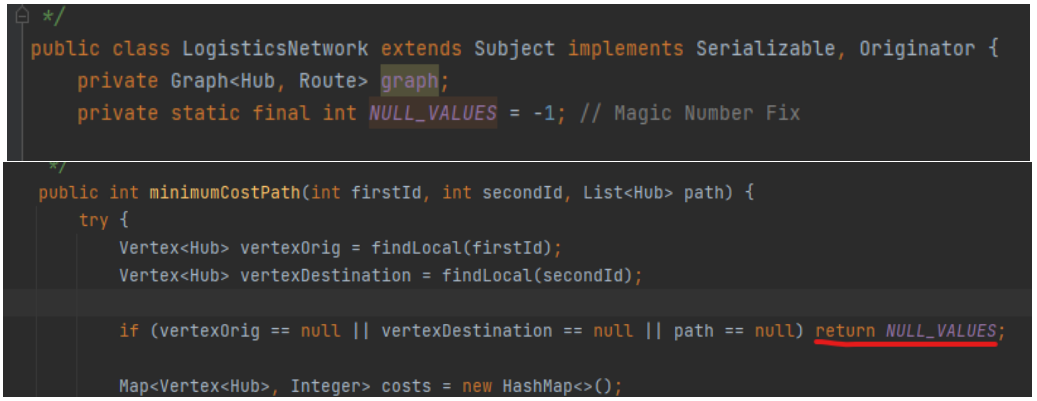
Este code smell foi encontrado na classe 'LogisticsNetwork'. Refere-se a um valor que está no código sem um significado óbvio.



```
public int minimumCostPath(int firstId, int secondId, List<Hub> path) {
    try {
        Vertex<Hub> vertexOrig = findLocal(firstId);
        Vertex<Hub> vertexDestination = findLocal(secondId);

        if (vertexOrig == null || vertexDestination == null || path == null) return -1;

        Map<Vertex<Hub>, Integer> costs = new HashMap<>();
        Map<Vertex<Hub>, Vertex<Hub>> predecessors = new HashMap<>();

        dijkstra(vertexOrig, costs, predecessors);
        path.clear();
```

Fig.9 - Code smell 'Magic Number'.

Para corrigir o problema, foi criada uma constante com o valor.



```
*/
public class LogisticsNetwork extends Subject implements Serializable, Originator {
    private Graph<Hub, Route> graph;
    private static final int NULL_VALUES = -1; // Magic Number Fix
```



```
*/
public int minimumCostPath(int firstId, int secondId, List<Hub> path) {
    try {
        Vertex<Hub> vertexOrig = findLocal(firstId);
        Vertex<Hub> vertexDestination = findLocal(secondId);

        if (vertexOrig == null || vertexDestination == null || path == null) return NULL_VALUES;

        Map<Vertex<Hub>, Integer> costs = new HashMap<>();
```

Fig.10 - Criação de uma constante com o valor. Substituição do valor pela constante no método.

Conclusão

Através da realização deste projeto, foi possível adquirir conhecimentos avançados de programação. A aplicação baseou-se numa rede logística de transportes, porém as mesmas técnicas se aplicam a outros problemas semelhantes.
Portanto, a partir de agora, os elementos do grupo estão preparados para modelar e desenvolver aplicações que envolvam a teoria dos grafos.

A aplicação de padrões de software, nomeadamente do padrão MVC, foi bastante importante, pois permitiu separar claramente a interface gráfica da lógica de negócio, o que facilitou significativamente o desenvolvimento de código

Foi feito o refactoring do código, que era um conceito que pouco ou nada conhecíamos, e que é extremamente importante no desenvolvimento de código.

Também foi deduzido que a documentação é algo essencial durante o desenvolvimento de código, pois permite aumentar a legibilidade do código e explicar de forma sucinta cada seção do código.

Portanto, foi importante aplicar estes conceitos durante o projeto, pois são conceitos fundamentais no desenvolvimento de software e sem dúvida que a experiência adquirida será útil em futuros projetos.

Bibliografia

Code Smells (refactoring.guru) - Informação sobre code smells.

Refactoring Techniques - Informação sobre técnicas de refactoring.

The Catalog of Design Patterns (refactoring.guru) - Catálogo de padrões de software.