



Relatório de Projeto de Programação Avançada

ÉPOCA NORMAL

Licenciatura em Engenharia Informática

ORIENTADOR

Professora Patrícia Macedo

GRUPO DE TRABALHO

Miguel Calha, 201902037

João Dâmaso, 201901629

Diogo Soares, 201900765

Marcos Jesus, 201900268

Índice

Introdução.....	4
2. Implementação da Aplicação.....	5
2.1. Funcionalidades da Aplicação	5
2.2. ADTS	6
2.3. Padrões Implementados.....	7
3. Interface do Utilizador.....	8
.....	11
4. Refactoring.....	12
4.1. Bad smells encontrados	12
Dead Code	13
Comments	13
Inline Temp.....	14
Long Method.....	15
Duplicate Code	16
Large Class	17
Conclusão.....	18

Introdução

Este relatório detalha o desenvolvimento de uma aplicação em JavaFX, visando a gestão eficiente e intuitiva de um sistema de ficheiros personalizado, com o título *Personal File System* (PFS). Integrando o ADT Tree, esta aplicação reflete uma abordagem moderna e robusta na organização e manipulação de ficheiros e diretórios. Através da implementação de funcionalidades fundamentais como criação, edição, e navegação, juntamente com a persistência de dados, o projeto almeja oferecer uma solução prática e versátil para a gestão de ficheiros. Adicionalmente, a aplicação é um testemunho da aplicação dos princípios da orientação a objetos e padrões de software avançados, marcando um ponto de viragem na interação com sistemas de ficheiros personalizados.

2. Implementação da Aplicação

2.1. Funcionalidades da Aplicação

O PFS é uma aplicação avançada em JavaFX que oferece várias funcionalidades para a gestão de ficheiros e diretórias. As principais funcionalidades incorporadas no sistema são:

Navegação e Visualização Hierárquica: Foi implementada uma `TreeView` para permitir a navegação intuitiva pela estrutura de ficheiros e diretórias. A `TableView` fornece uma visualização detalhada dos ficheiros, incluindo informações como nome, data de criação, tamanho e status de bloqueio.

Gestão Completa de Ficheiros e Pastas: O sistema permite criar, editar, renomear e excluir ficheiros e pastas, oferecendo um controlo total sobre a gestão de dados.

Movimentação e Cópia: Implementadas funcionalidades para mover e copiar ficheiros e pastas, facilitando a organização e gestão dos dados.

Funcionalidade de Zip: Foi adicionada a capacidade de compactar ficheiros e pastas em formatos `.zip`, otimizando o armazenamento e compartilhamento de dados.

Undo: Foi utilizado `Command` para possibilitar operações de desfazer e refazer, mantendo um histórico de modificações para rastreamento e controlo.

Bloqueio de Ficheiros: O sistema inclui um mecanismo de bloqueio para proteger ficheiros contra modificações ou exclusões não autorizadas.

Pesquisa e Filtragem: A funcionalidade de pesquisa permite localizar rapidamente ficheiros e diretórios específicos dentro da estrutura do sistema.

Estatísticas e Métricas: O sistema apresenta estatísticas detalhadas e métricas, como a contagem de modificações e visualizações gráficas das atividades dos ficheiros.

Funcionalidades Avançadas de Gestão de Ficheiros: Inclui operações avançadas como a visualização de conteúdo de ficheiros, gestão do espaço ocupado e identificação de diretórios com grande número de descendentes.

2.2. ADTS

A TreeLinked é uma implementação concreta de uma árvore genérica, onde cada nó pode ter um número variável de filhos. Esta classe é genérica, indicada pela notação <E>, o que significa que ela pode armazenar qualquer tipo de objeto, aumentando assim a sua flexibilidade e reutilização. Cada elemento da árvore é armazenado num nó (TreeNode). Cada TreeNode contém o valor do elemento e uma lista dos seus filhos, permitindo assim uma estrutura hierárquica.

A árvore possui uma única raiz, que é o ponto de partida para todos os outros nós. A raiz pode ser inicializada vazia ou com um elemento específico.

Operações Fundamentais:

Inserção: A árvore permite a inserção de novos elementos. Estes elementos são adicionados como filhos de um nó existente.

Remoção: Elementos (nós) podem ser removidos da árvore. Esta operação pode exigir reajustes na estrutura para manter a integridade da árvore.

Travessia: A classe fornece métodos para percorrer a árvore. Isto é útil para operações como busca, exibição ou processamento de cada elemento da árvore.

Serialização: A implementação suporta serialização, permitindo que a estrutura da árvore seja facilmente salva e recuperada, um aspecto vital para persistência de dados no PFS.

Flexibilidade e Uso no PFS: No contexto do PFS, a TreeLinked é utilizada para representar a estrutura hierárquica de ficheiros e diretorias. Cada nó na árvore pode representar um ficheiro ou uma diretoria, permitindo uma representação clara e manipulável da estrutura de ficheiros.

2.3. Padrões Implementados

- **MVC e Command**

O padrão *Model-View-Controller* (MVC) serve para separar as funcionalidades principais do negócio. Esta separação permite partilhar os mesmos dados pelas diferentes vistas, tornando assim, mais fácil as tarefas de desenvolvimento, teste e manutenção. Este é um padrão de arquitetura de *software* amplamente utilizado, especialmente em aplicações com interfaces gráficas. No MVC, a aplicação é dividida em três componentes principais:

Model: Representa a lógica de negócios e os dados. No caso do projeto é a classe PFS;

View: É responsável por exibir os dados ao utilizador, geralmente através de uma interface gráfica. PFSView.

Controller: Atua como um intermediário entre o Model e a View, controlando o fluxo de dados e as interações do utilizador. Controller

O MVC ajuda a separar a lógica de negócios da interface do utilizador, facilitando a manutenção e a expansão da aplicação

O padrão Command é utilizado em programação orientada a objetos. É útil para implementar funcionalidades como operações de desfazer (undo) e refazer (redo).

A interface UndoableCommand define um método undo(), que é a essência do padrão Command para operações de desfazer.

Cada ação específica, como renomear, criar ou excluir arquivos, é implementada em uma classe separada que implementa UndoableCommand, por exemplo, RenameCommand, CreateFolderCommand, DeleteFileCommand, etc.

Funcionamento dos Comandos:

Cada uma dessas classes concretas encapsula todos os detalhes necessários para realizar e desfazer a ação. Por exemplo, RenameCommand armazena os ficheiros original e renomeado, CreateFolderCommand armazena a pasta criada, e assim por diante.

Quando uma ação é realizada na interface do utilizador, um objeto de comando correspondente é criado e armazenado numa stack de comandos (undoStack).

Para desfazer uma ação, o comando no topo da pilha undoStack é desempilhado o método undo() é chamado. Revertendo a ação realizada.

3. Interface do Utilizador

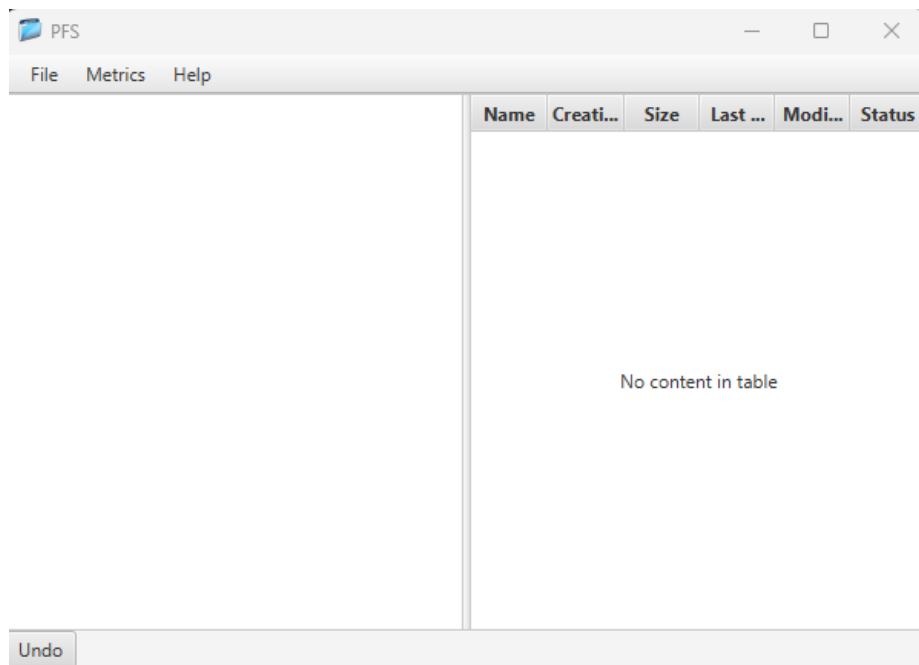


Fig.1- Interface principal

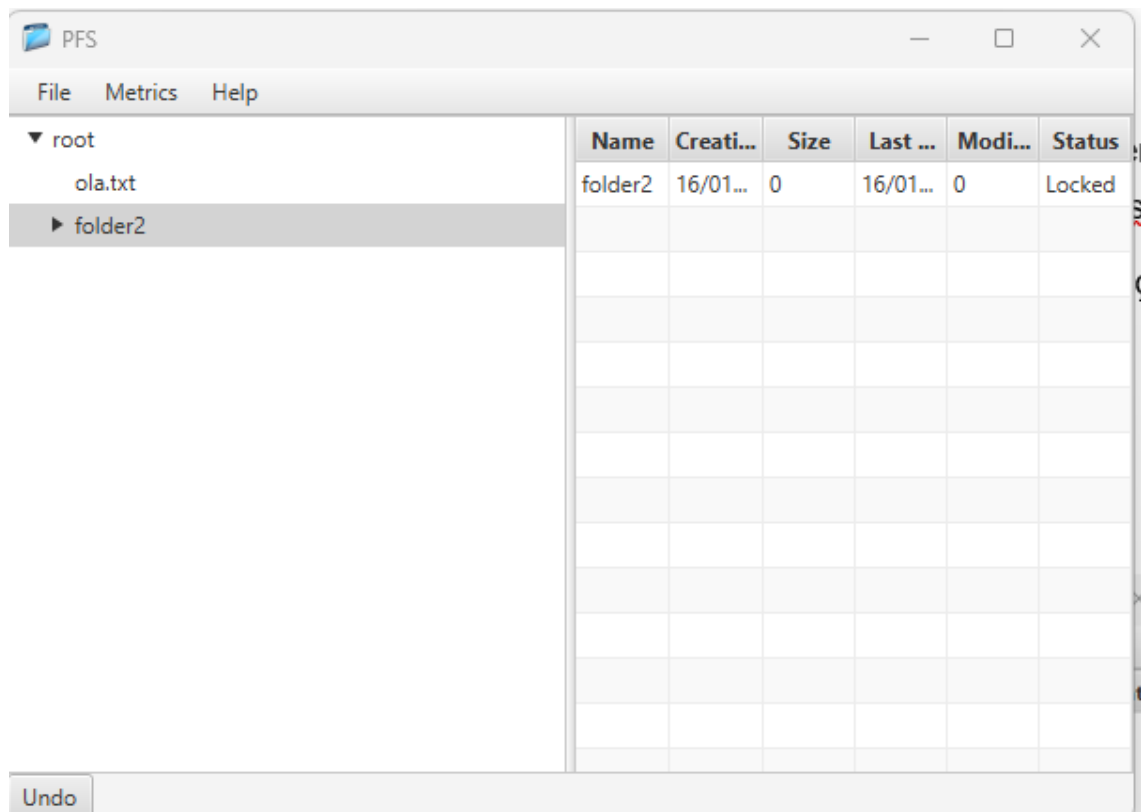


Fig.2- Interface principal depois de carregamento de ficheiros

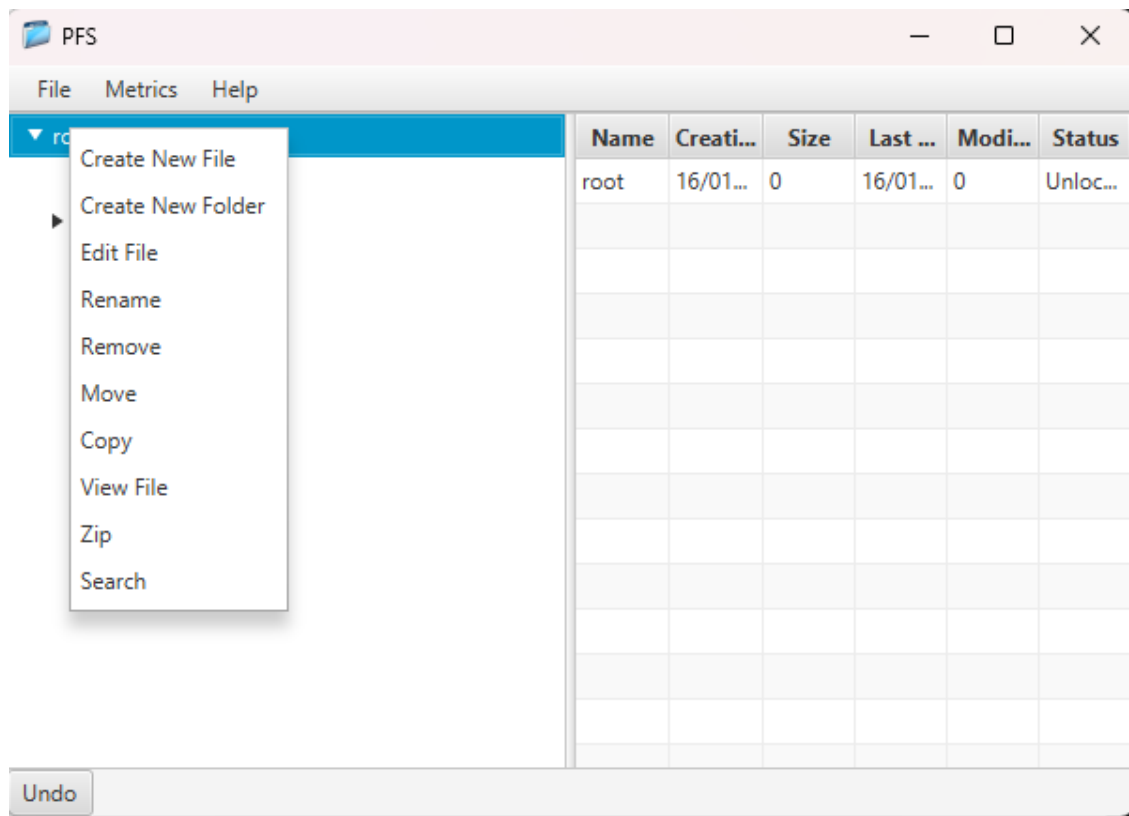


Fig.3- Interface principal – opções de manipulação

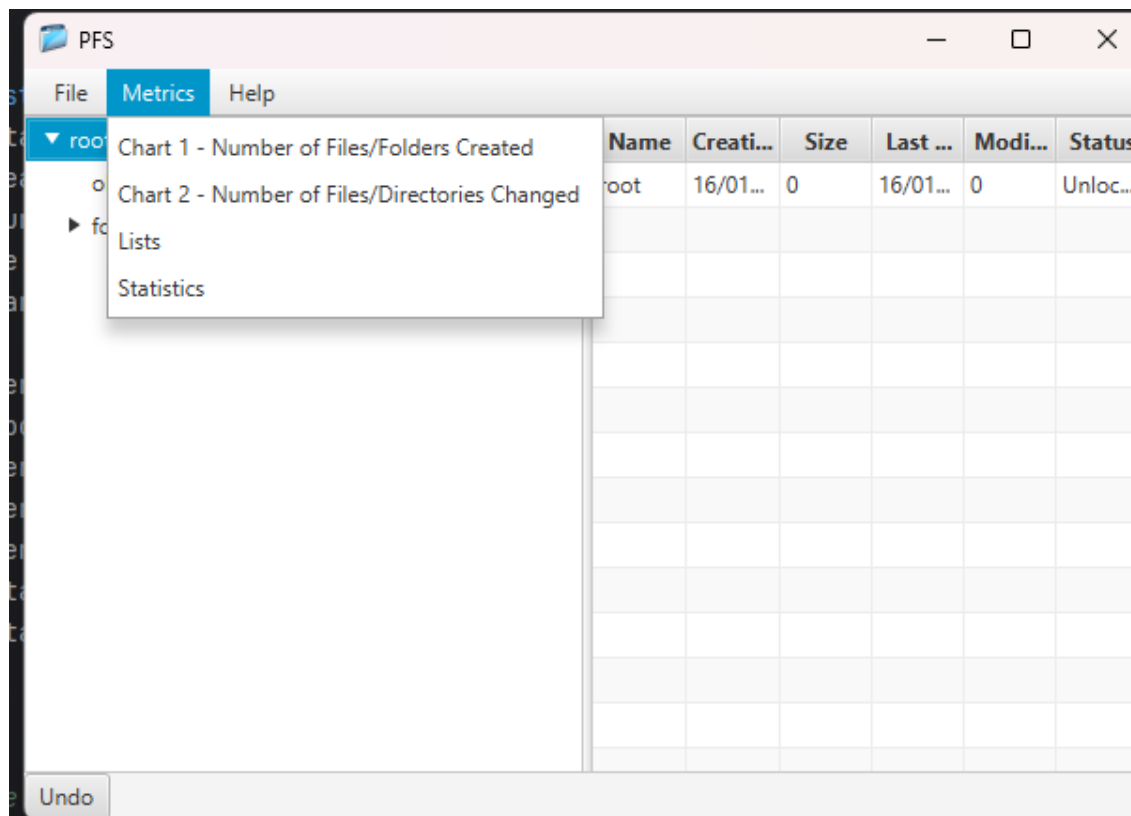


Fig.4- Interface principal – Métricas

4. Refactoring

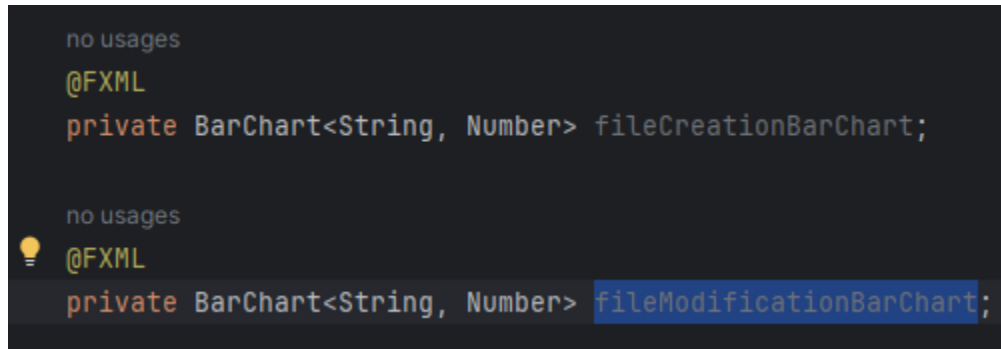
4.1. Bad smells encontrados

Tabela 1 – Tabela com os bad smells encontrados no projeto

Bad smell	Nº de ocorrências	<u>Técnica</u> refactoring aplicada
Duplicate Code	4	Extract Method
Dead Code/Speculative Generality	7	Delete the code
Long Method	1	Extract Method
Comments	83	Rename Method
Inappropriate Intimacy	0	-
Data class	0	-
Large class	1	Extract class
Temporary Field	2	Inline Temp
Dead Code	0	-
Data Clumps	0	-
Refused Bequest	0	-
Switch Statements	0	-

Dead Code

A existência do *bad smell Duplicate Code* indica que dois ou mais fragmentos de código aparentam ser semelhantes ou até mesmo iguais. No nosso caso, foi detetada uma repetição na criação de alertas visuais em casos específicos após cliques nos vários botões existentes.



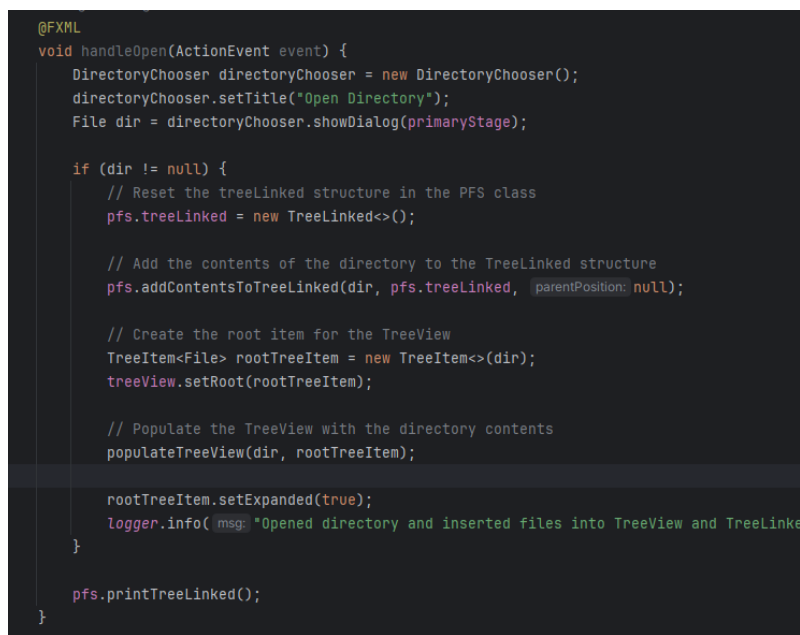
```
no usages
@FXML
private BarChart<String, Number> fileCreationBarChart;

no usages
@FXML
private BarChart<String, Number> fileModificationBarChart;
```

Figura 5 – Exemplo de Dead Code

Comments

Dizemos que existe um *bad smell de Comments* quando existem diversos comentários sobre a explicação do método. Encontramos este *bad smell* na nossa classe *StrategyJson*.



```
@FXML
void handleOpen(ActionEvent event) {
    DirectoryChooser directoryChooser = new DirectoryChooser();
    directoryChooser.setTitle("Open Directory");
    File dir = directoryChooser.showDialog(primaryStage);

    if (dir != null) {
        // Reset the treeLinked structure in the PFS class
        pfs.treeLinked = new TreeLinked<>();

        // Add the contents of the directory to the TreeLinked structure
        pfs.addContentsToTreeLinked(dir, pfs.treeLinked, parentPosition: null);

        // Create the root item for the TreeView
        TreeItem<File> rootTreeItem = new TreeItem<>(dir);
        treeView.setRoot(rootTreeItem);

        // Populate the TreeView with the directory contents
        populateTreeView(dir, rootTreeItem);

        rootTreeItem.setExpanded(true);
        logger.info(msg: "Opened directory and inserted files into TreeView and TreeLink
    }

    pfs.printTreeLinked();
}
```

Figura 6 – Exemplo de código de com *Comments*

Neste caso específico, apenas decidimos remover os comentários.

Inline Temp

O inline temp é um bad smell que pode ser identificado como uma declaração de variável dentro de um método que o seu intuito era ser uma declaração temporária. Para retirar este bad smell os alunos apenas retiraram a mesma dado que não estava a ser utilizada.

```
public void renameElement(Position<File> elementPosition, String newName) {
    try {
        checkPosition(elementPosition);

        File currentFile = elementPosition.element();
        String currentName = currentFile.getName();

        Position<File> existingPosition = findPositionWithName(elementPosition, newName);
        if (existingPosition != null) {
            throw new IllegalArgumentException("An element with the provided name already ex
        }

        treeLinked.replace(elementPosition, new File(currentFile.getParent(), newName));
        lockStatusMap.put(new File(currentFile.getParent(), newName), lockStatusMap.remove(c
        fileContentsMap.put(new File(currentFile.getParent(), newName), fileContentsMap.remo

        logger.info(msg: "Node Renamed--->" + newName);
        File file = elementPosition.element();
        modificationCountMap.put(file, modificationCountMap.getOrDefault(file, defaultValue: 0)
        modificationDateMap.put(file, new Date());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figura8 – Exemplo de código de com *Inline temp*

Long Method

Este é um exemplo de um Bad smell, chamado Long method, que pode ser facilmente identificado como um método demasiado longo que está a fazer demasiadas coisas.

```
/**
 * Método para adicionar conteúdos à TreeLinked
 * @param directory
 * @param treeLinked
 * @param parentPosition
 */
2 usages 1 MiguelCaiha
public void addContentsToTreeLinked(File directory, TreeLinked<File> treeLinked, Position<File> parentPosition) {
    if (directory == null || !directory.exists()) {
        return;
    }
    Position<File> currentPosition;
    if (parentPosition == null) {
        currentPosition = treeLinked.insert( parent: null, directory);
    } else {
        currentPosition = treeLinked.insert(parentPosition, directory);
    }
    if (directory.isDirectory()) {
        File[] files = directory.listFiles();
        if (files != null) {
            for (File file : files) {
                addContentsToTreeLinked(file, treeLinked, currentPosition);
            }
        }
    }
}
```

Figura9 – Exemplo de Long Method

```
2 usages 1 MiguelCaiha *
public void addContentsToTreeLinked(File directory, TreeLinked<File> treeLinked, Position<File> parentPosition) {
    if (directory == null || !directory.exists()) {
        return;
    }
    Position<File> currentPosition = insertInTreeLinked(directory, treeLinked, parentPosition);
    addDirectoryContents(directory, treeLinked, currentPosition);
}

1 usage 1 MiguelCaiha *
private void addDirectoryContents(File directory, TreeLinked<File> treeLinked, Position<File> parentPosition) {
    if (directory.isDirectory()) {
        File[] files = directory.listFiles();
        if (files != null) {
            for (File file : files) {
                addContentsToTreeLinked(file, treeLinked, parentPosition);
            }
        }
    }
}

1 usage new *
private Position<File> insertInTreeLinked(File file, TreeLinked<File> treeLinked, Position<File> parentPosition) {
    if (parentPosition == null) {
        return treeLinked.insert( parent: null, file);
    } else {
        return treeLinked.insert(parentPosition, file);
    }
}
```

Figura10 – Depois de refactoring

Duplicate Code

Este bad smell ocorre no âmbito em que estamos a chamar o insert na TreeLinked sempre diretamente e da mesma maneira para criação de ficheiros e pastas.

```
public void createFolder(Position<File> parent, String folderName) {
    try {
        if (parent == null) {
            if (treeLinked.isEmpty()) {
                treeLinked = new TreeLinked<>(new File(folderName));
            }
        } else {
            File parentFile = parent.element();
            File newFolder = new File(parentFile, folderName);

            if (!parentFile.isDirectory()) {
                throw new IllegalArgumentException("Parent does not exist or is not a folder.");
            }
            if (!newFolder.mkdir()) {
                throw new IllegalArgumentException("Cannot create folder.");
            }
            treeLinked.insert(parent, newFolder);
            logger.info(msg: "Created a node in TreeLinked" + " -> " + folderName);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    File newFolder = new File(parent.element(), folderName);
    creationCountMap.put(newFolder, creationCountMap.getOrDefault(newFolder, defaultValue: 0) + 1);
    printTreeLinked();
}
```

Figura11 – Duplicate Code

Podemos corrigir isto, criando um método auxiliar que permite inserir elementos na TreeLinked de maneira dinâmica, centralizando assim a inserção de elementos num único local, o que ajuda na manutenção da aplicação.

```
private Position<File> insertInTreeLinked(Position<File> parent, File file) {
    if (parent == null && treeLinked.isEmpty()) {
        treeLinked = new TreeLinked<>(file);
        return treeLinked.root();
    } else if (parent != null) {
        return treeLinked.insert(parent, file);
    } else {
        throw new IllegalArgumentException("Parent is null but TreeLinked is not empty.");
    }
}
```

Figura12– Método criado

Large Class

Os alunos identificaram uma classe que tinha demasiadas funcionalidades, algumas delas que saíam um pouco do espectro do que a classe deveria conter. A classe referida é a classe PFS, que deveria apenas ter funções atómicas e no entanto tinha também o cálculo das métricas. Para resolver isto, os alunos aplicaram a técnica de refactoring: extract method e criaram uma classe chamada “Metrics” só para colocar as funcionalidades das métricas.



Figura13– Classe criada

Conclusão

É possível concluir que a aplicação desenvolvida oferece uma solução robusta e eficaz para a gestão de um sistema ficheiros. A estrutura da aplicação foi cuidadosamente planeada e implementada, respeitando os princípios da orientação a objetos e a utilização de padrões de software, o que resultou numa arquitetura clara e modular.

O processo de refactoring foi uma etapa crucial para a melhoria da qualidade do código. A identificação e correção de bad smells, bem como a aplicação de técnicas de refactoring adequadas, contribuíram significativamente para a eficiência, legibilidade e manutenibilidade do código.

Em conclusão, o projeto cumpriu com sucesso os requisitos estabelecidos e demonstrou uma aplicação prática eficiente dos conceitos teóricos aprendidos. O uso de padrões de design, juntamente com uma abordagem de refactoring cuidadosa, resultou numa aplicação robusta e fácil de manter.