DOCUMENT TECHNIQUE

Création d'un Système de Fichiers Simplifié

Réalisées par :

- AIT ATMANE FATIMA-EZZAHRA
- AKOURI Yassine
- EL ALAMI AICHA
- EL HAIDAOUI ZAYNAB

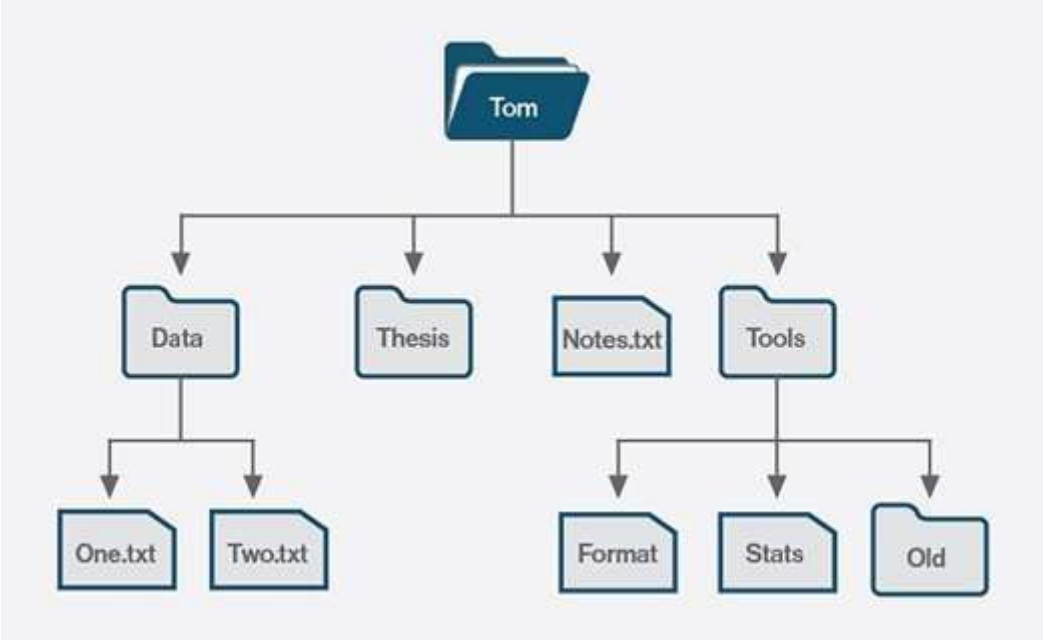
Table des matières

Introduction

Le projet de création d'un Système de Fichiers Simplifié en mémoire représente une exploration approfondie des fondamentaux de la gestion des données informatiques. En simulant les interactions clés entre un système d'exploitation et un système de fichiers, ce projet offre une plateforme pour développer une compréhension pratique des structures de données essentielles, telles que les fichiers, les répertoires et leurs métadonnées associées. En se concentrant sur des opérations fondamentales comme la création, la suppression, la lecture et l'écriture de fichiers, ainsi que la gestion efficace de l'espace disque en mémoire, les participants auront l'occasion d'explorer des concepts avancés tels que la fragmentation et la défragmentation. En intégrant éventuellement des fonctionnalités comme la journalisation et une interface utilisateur graphique, ce projet permettra également de développer des compétences pratiques en conception logicielle et en développement d'applications système.

Architecture de travail

Le système de fichiers en mémoire repose sur plusieurs composants clés interagissant de manière cohérente pour assurer la gestion efficace des fichiers et des répertoires. Au centre de cette architecture se trouve le **Gestionnaire de Système de Fichiers (FSM)**, qui orchestre les opérations fondamentales telles que la création, la suppression, la lecture et l'écriture de fichiers. Les données sont structurées à l'aide de deux principales entités : les **Fichiers** et les **Répertoires**.



Classes principales :

Les principales classes du système de gestion de fichiers en mémoire incluent DirectoryItem pour la gestion des répertoires et de leur contenu, DiskManager pour l'allocation efficace de l'espace disque virtuel, FileMetaData pour stocker les métadonnées des fichiers, et FileOperations pour les opérations fondamentales comme la création, la lecture, l'écriture et la suppression de fichiers.

- **DirectoryItem :** Cette classe représente un répertoire dans le système de fichiers. Elle contient une liste de fichiers et de sous-répertoires (instances de FileSystemItem). Le DirectoryItem gère également les opérations liées à la navigation dans la hiérarchie des répertoires et la gestion des fichiers qu'il contient.

- **DiskManager :** Responsable de l'allocation et de la libération de l'espace disque en mémoire. Le DiskManager simule les opérations de gestion de l'espace disque pour assurer une utilisation efficace des ressources disponibles. Il peut intégrer des algorithmes d'allocation d'espace pour optimiser la gestion des blocs de données alloués aux fichiers.

- **FileMetaData :** Cette classe représente les métadonnées associées à un fichier. Les métadonnées peuvent inclure des informations telles que le nom du fichier, la taille, les dates de création et de modification, les autorisations d'accès et d'autres attributs pertinents nécessaires pour gérer et organiser les fichiers.

- **FileOperations :** Fournit des méthodes pour effectuer des opérations de base sur les fichiers telles que la création, la lecture, l'écriture, la suppression et la gestion des permissions. Cette classe encapsule le logique métier pour manipuler les données des fichiers et maintenir la cohérence du système de fichiers.

- **Journal :** Journalise les opérations effectuées sur les fichiers et le disque. Cette classe permet de garder une trace des actions effectuées par les utilisateurs ou par le système lui-même, facilitant ainsi la récupération en cas de panne ou la résolution de problèmes.

- **FileSystemItem :** Classe abstraite qui sert de base pour les éléments du système de fichiers (File et Directory). Elle définit les propriétés et les méthodes communes aux fichiers et aux répertoires, permettant ainsi une gestion unifiée et polymorphique des éléments du système de fichiers.

Implémentation :

1. La classe FileSystemItem :

- Déclaration d’une classe abstraite publique nommée FileSystemItem.il contient un attribut privé nommé name de type String. Cet attribut stocke le nom de l'élément du système de fichiers.

```
public abstract class FileSystemItem {
    private String name;
```

- Déclaration d’un constructeur public pour la classe FileSystemItem qui prend un argument name de type String.

```
    public FileSystemItem(String name) {
        this.name = name;
    }
```

- Définition des méthodes getName et setName.

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

2. La classe FileOperations :

- Déclaration d’une méthode publique qui prend un chemin de répertoire en argument et retourne un booléen indiquant si la suppression a réussi.

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.PosixFilePermission;
import java.util.Set;

public class FileOperations {
    public boolean deleteDirectory(String dirPath) {
        if (dirPath == null || dirPath.trim().isEmpty()) {
            Journal.Log("Chemin de répertoire invalide pour la suppression.");
            return false;
        }

        File dir = new File(dirPath);
        if (!dir.isDirectory()) {
            Journal.Log("Not a valid directory: " + dirPath);
            return false;
        }

        return deleteRecursive(dir);
    }
```

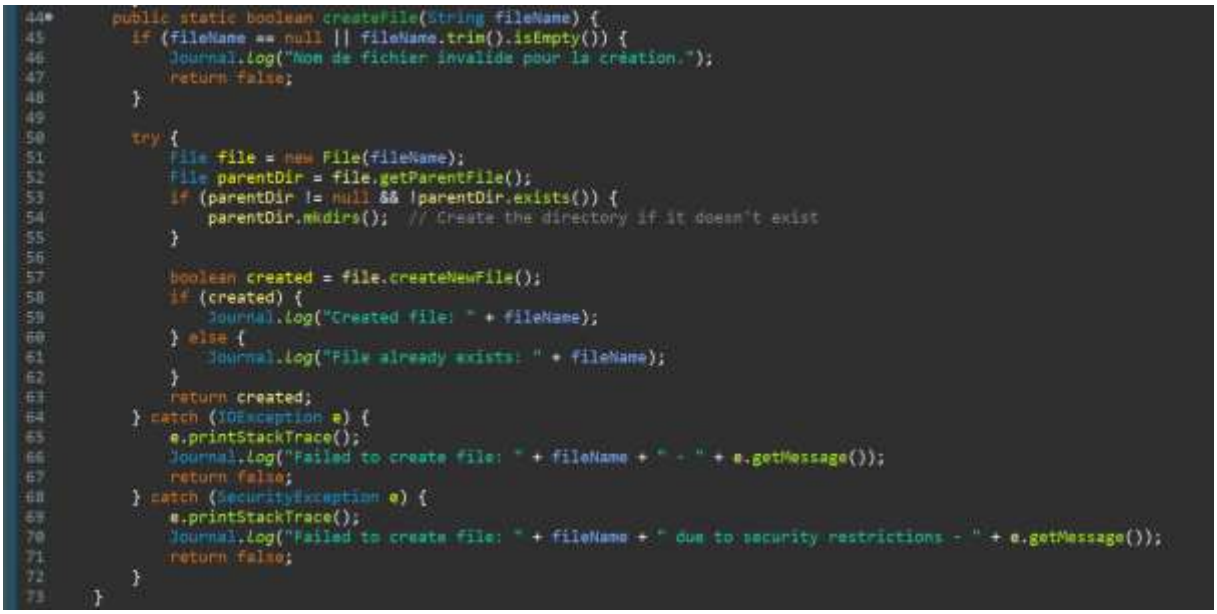
- Déclaration d’une méthode Privée pour Supprimer Récursivement qui prend un objet File et retourne un booléen indiquant si la suppression a réussi.

```
    private boolean deleteRecursive(File file) {
        if (!file.exists()) {
            return false;
        }

        if (file.isDirectory()) {
            File[] files = file.listFiles();
            if (files != null) {
                for (File f : files) {
                    deleteRecursive(f);
                }
            }
        }

        // Delete the directory or file
        return file.delete();
    }
```

- Déclaration d’une méthode statique publique qui prend un nom de fichier et retourne un booléen indiquant si la création a réussi.



```java
public static boolean createFile(String fileName) {
    if (fileName == null || fileName.trim().isEmpty()) {
        Journal.Log("Nom de fichier invalide pour la création.");
        return false;
    }

    try {
        File file = new File(fileName);
        File parentDir = file.getParentFile();
        if (parentDir != null && !parentDir.exists()) {
            parentDir.mkdirs();  // Create the directory if it doesn't exist
        }

        boolean created = file.createNewFile();
        if (created) {
            Journal.Log("Created file: " + fileName);
        } else {
            Journal.Log("File already exists: " + fileName);
        }
        return created;
    } catch (IOException e) {
        e.printStackTrace();
        Journal.Log("Failed to create file: " + fileName + " - " + e.getMessage());
        return false;
    } catch (SecurityException e) {
        e.printStackTrace();
        Journal.Log("Failed to create file: " + fileName + " due to security restrictions - " + e.getMessage());
        return false;
    }
}
```

- Déclaration d'une méthode statique publique qui prend un nom de fichier et retourne un booléen indiquant si la suppression a réussi.

```java
public static boolean deleteFile(String fileName) {
    if (fileName == null || fileName.trim().isEmpty()) {
        Journal.Log("Nom de fichier invalide pour la suppression.");
        return false;
    }

    File file = new File(fileName);
    boolean deleted = file.delete();
    if (deleted) {
        Journal.Log("Deleted file: " + fileName);
    } else {
        Journal.Log("Failed to delete file: " + fileName);
    }
    return deleted;
}
```

- Déclaration d'une méthode statique publique qui prend un nom de fichier et un contenu, et retourne un booléen indiquant si l'écriture a réussi :

  - Vérifie si le nom de fichier ou le contenu est nul. Si oui, enregistre un message dans le journal et retourne false.
  - Tente d'écrire le contenu dans le fichier en utilisant FileWriter, enregistre le résultat dans le journal et gère les exceptions IOException.

```
91  public static boolean writeFile(String fileName, String content) {
92      if (fileName == null || content == null) {
93          Journal.Log("Nom de fichier ou contenu invalide pour l'écriture.");
94          return false;
95      }
96
97      try (FileWriter writer = new FileWriter(fileName)) {
98          writer.write(content);
99          Journal.Log("Wrote to file: " + fileName);
100         return true;
101     } catch (IOException e) {
102         e.printStackTrace();
103         Journal.Log("Failed to write to file: " + fileName);
104         return false;
105     }
106 }
```

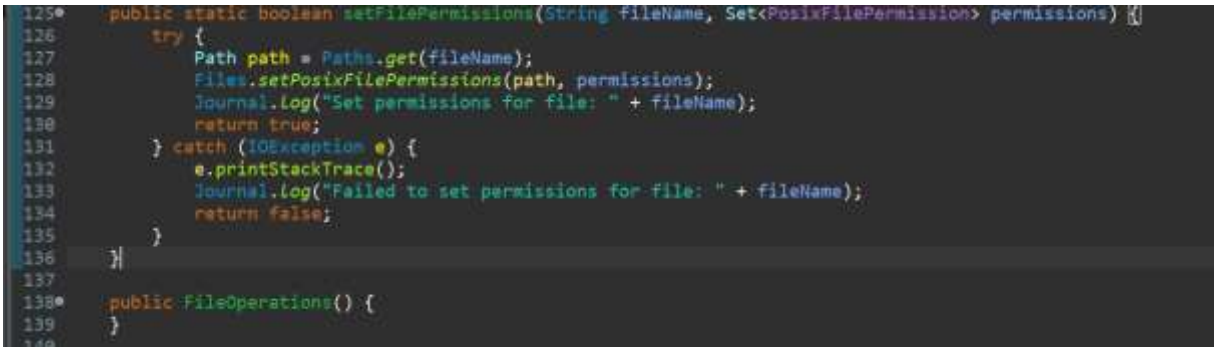
- Déclaration d'une méthode statique publique qui prend un nom de fichier et retourne le contenu du fichier sous forme de chaîne de caractères :

  - Vérifie si le fichier existe, enregistre un message et retourne une erreur si ce n'est pas le cas.
  - Tente de lire le contenu du fichier en utilisant Files.readAllBytes, enregistre le résultat dans le journal et gère les exceptions IOException.

```
100 public static String readFile(String fileName) {
109     Path path = Paths.get(fileName);
110     if (!Files.exists(path)) {
111         Journal.Log("File not found: " + fileName);
112         return "Error: File not found.";
113     }
114     try {
115         String content = new String(Files.readAllBytes(path));
116         Journal.Log("Read file: " + fileName);
117         return content;
118     } catch (IOException e) {
119         e.printStackTrace();
120         Journal.Log("Failed to read file: " + fileName);
121         return "Error: Could not read file.";
122     }
123 }
```

- Déclare une méthode statique publique qui prend un nom de fichier et un ensemble de permissions POSIX, et retourne un booléen indiquant si l'opération a réussi :

  - Tente de définir les permissions du fichier en utilisant Files.setPosixFilePermissions, enregistre le résultat dans le journal et gère les exceptions IOException.



```
125    public static boolean setFilePermissions(String fileName, Set<PosixFilePermission> permissions) {
126        try {
127            Path path = Paths.get(fileName);
128            Files.setPosixFilePermissions(path, permissions);
129            Journal.log("Set permissions for file: " + fileName);
130            return true;
131        } catch (IOException e) {
132            e.printStackTrace();
133            Journal.log("Failed to set permissions for file: " + fileName);
134            return false;
135        }
136    }
137
138    public FileOperations() {
139    }
```

3. La classe DirectoryItem :

- Déclare une classe publique DirectoryItem qui hérite de la classe FileSystemItem. Cela signifie que DirectoryItem est un type spécialisé de FileSystemItem.
- Déclare un attribut privé files, qui est une liste d'objets FileMetaData. Cette liste contient les métadonnées des fichiers dans le répertoire.
- Déclare un constructeur public qui prend un nom en paramètre.

```
import java.util.ArrayList;
import java.util.List;

public class DirectoryItem extends FileSystemItem {
    private List<FileMetaData> files;

    public DirectoryItem(String name) {
        super(name);
        this.files = new ArrayList<>();
    }
```

- Déclare une méthode publique addFile qui prend un objet FileMetaData en paramètre et l'ajoute à la liste files.

```
    public void addFile(FileMetaData file) {
        files.add(file);
    }
```

- Déclare une méthode publique removeFile qui prend un objet FileMetaData en paramètre et le supprime de la liste files.

```
    public void removeFile(FileMetaData file) {
        files.remove(file);
    }
```

- Déclare une méthode publique getFiles qui retourne la liste files. Cela permet d'accéder à tous les fichiers du répertoire.

```
    public List<FileMetaData> getFiles(){
        return files;
    }
}
```

4. La classe FileMetaData :

- Déclare une classe publique FileMetaData qui représente les métadonnées d'un fichier. Ce derniers contient les attributs suivantes : le nom du fichier, size : la taille du fichier en octets, permissions : les permissions du fichier sous forme de chaîne, creationDate : la date de création du fichier, modificationDate : la date de dernière modification du fichier.
- Déclare un constructeur public qui initialise les attributs name, size et permissions avec les valeurs fournies en paramètre.

```java
import java.util.Date;

public class FileMetaData {
    private String name;
    private long size;
    private String permissions;
    private Date creationDate;
    private Date modificationDate;

    public FileMetaData(String name, long size, String permissions) {
        this.name = name;
        this.size = size;
        this.permissions = permissions;
        this.creationDate = new Date();
        this.modificationDate = new Date();
    }
```

- Définition des méthodes getName, setName, getSize, setSize, getPermissions, setPermissions, getCreationDate, getModificationDate.

```java
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
        this.modificationDate = new Date();
    }
     public long getSize() {
        return size;
    }
    public void setSize(long size) {
        this.size = size;
        this.modificationDate = new Date();
    }
    public String getPermissions() {
        return permissions;
    }
    public void setPermissions(String permissions) {
        this.permissions = permissions;
        this.modificationDate = new Date();
    }
    public Date getCreationDate() {
        return creationDate;
    }
    public Date getModificationDate() {
        return modificationDate;
    }
```

- Définir la méthode toString pour fournir une représentation sous forme de chaîne de l'objet FileMetaData.

```
47    public String toString() {
48        return "FileMetaData{" +name + '\'' +
49                "name='" +name + '\'' +
50                ", size=" + size +
51                ", permissions='" +permissions + '\'' +
52                ", creationDate=" + creationDate +
53                ", modificationDate=" +modificationDate +
54                '}';
55    }
56 }
```

5. La classe DiskManager :

- Déclare une classe publique DiskManager qui gère l'allocation et la libération de l'espace disque ainsi que la gestion des fichiers. Ce dernier contient les attributs suivants :
    - TotalSpace : l'espace disque total en octets.
    - UsedSpace : l'espace disque utilisé en octets.
    - files : une liste de métadonnées des fichiers.
    - journal : un journal pour enregistrer les opérations.

- Déclare un constructeur public qui initialise totalSpace, usedSpace, files et journal avec les valeurs fournies en paramètres. journal.log enregistre un message de journalisation pour indiquer l'initialisation de DiskManager.

- Déclare un autre constructeur public qui initialise totalSpace, usedSpace et files, mais n'inclut pas de journal fourni. Utilise un appel statique à Journal.log.

```java

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class DiskManager {
    private long totalSpace;
    private long usedSpace;
    private List<FileMetaData> files;
    private Journal journal;

    public DiskManager(long totalSpace, Journal journal) {
        this.totalSpace = totalSpace;
        this.usedSpace = 0;
        this.files = new ArrayList<>();
        this.journal = journal;
        journal.Log("Initialized DiskManager with total space: " + totalSpace + " bytes");
    }

    public DiskManager(long totalSpace) {
        this.totalSpace = totalSpace;
        this.usedSpace = 0;
        this.files = new ArrayList<>();
        Journal.Log("Initialized DiskManager with total space: " + totalSpace + " bytes");
    }
```

- Méthodes d'Accès et de Gestion de l'Espace :
    - **GetFreeSpace :** Retourne l'espace disque libre en soustrayant usedSpace de totalSpace.

```
    public long getFreeSpace() {
        return totalSpace - usedSpace;
    }
```

    - **AllocateSpace :** Tente d'allouer un espace disque de size octets.
        - Si l'espace libre est suffisant, incrémente usedSpace de size et journalise l'opération.
        - Sinon, journalise l'échec et retourne false.

```
    public boolean allocateSpace(long size) {
        if (size <= getFreeSpace()) {
            usedSpace += size;
            journal.log("Allocated space: " + size + " bytes");
            return true;
        }
        journal.log("Failed to allocate space: " + size + " bytes");
        return false;
    }
```

    - **ReleaseSpace :** Libère un espace disque de size octets en décrémentant usedSpace et journalise l'opération.

```
    public void releaseSpace(long size) {
        usedSpace -= size;
        journal.log("Released space: " + size + " bytes");
    }
```


```
    public void releaseSpace(long size) {
        usedSpace -= size;
        journal.log("Released space: " + size + " bytes");
    }
```


- Méthodes de Gestion des Fichiers :
    - **AddFile :** Tente d'ajouter un fichier.
        - Si l'allocation d'espace est réussie, ajoute le fichier à la liste files et journalise l'ajout.

- Sinon, affiche un message d'erreur et journalise l'échec.

```
public void addFile(FileMetaData file) {
    if (allocateSpace(file.getSize())) {
        files.add(file);
        journal.log("Added file: " + file.getName());
    } else {
        System.out.println("Not enough space to add file: " + file.getName());
        journal.log("Failed to add file: " + file.getName() + " due to insufficient space");
    }
}
```

- **RemoveFile :** Supprime un fichier de la liste files, libère l'espace qu'il occupait et journalise l'opération.

```
public void removeFile(FileMetaData file) {
    files.remove(file);
    releaseSpace(file.getSize());
    journal.log("Removed file: " + file.getName());
}
```

- **Defragment :** Définit les fichiers en fonction de leur date de création pour améliorer l'efficacité et journalise l'opération.

```
public void defragment() {
    Collections.sort(files, (f1, f2) -> Long.compare(f1.getCreationDate().getTime(), f2.getCreationDate().getTime()));
    journal.log("Defragmented disk");
}
```

- Méthode d'Accès aux Fichiers :
  - **GetFiles :** Retourne la liste des fichiers gérés par DiskManager

```
public List<FileMetaData> getFiles(){
    return files;
}
}
```

6. La classe journal :

- Déclare une classe publique Journal pour gérer un journal de log.
- Déclare un attribut statique privé logEntries qui est une liste de chaînes de caractères, initialisée avec une nouvelle instance de arrayList. Cet attribut est utilisé pour stocker les entrées de log.

- **log :** une méthode statique publique log qui ajoute une nouvelle entrée de log à la liste logEntries.

- **GetLogEntries :** une méthode statique publique getLogEntries qui retourne la liste des entrées de log.

Interface utilisateur :

FileManagementGUI est une classe qui hérite de JFrame, ce qui signifie que c'est une fenêtre d'application Swing.

1. Variables d’instance :

- **diskManager :** Gère les opérations de disque (création, suppression de fichiers, etc.).
- **fileOperations** : Gère les opérations de fichier (lire, écrire, etc.).
- **fileTree :** Arborescence des fichiers affichée à l'utilisateur.
- **treeModel :** Modèle de données pour fileTree.

```
1  import javax.swing.*;
2  import javax.swing.tree.DefaultMutableTreeNode;
3  import javax.swing.tree.DefaultTreeModel;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.io.File;
7  import javax.swing.tree.TreeNode;
8
9
10 public class FileManagementGUI extends JFrame {
11     private DiskManager diskManager;
12     private FileOperations fileOperations;
13     private JTree fileTree;
14     private DefaultTreeModel treeModel;
```

2. Initialisation des gestionnaires :

- **diskManager :** Initialisé avec une taille de disque de 1 Go et un journal pour les opérations de fichier.
- **fileOperations :** Crée une nouvelle instance de FileOperations.

```
16    public FileManagementGUI() {
17        diskManager = new DiskManager(1024L * 1024L * 1024L, new Journal());
18        fileOperations = new FileOperations();
19
```

3. Configuration de la fenêtre :

- **setTitle :** Définit le titre de la fenêtre.
- **setSize :** Définit la taille de la fenêtre (600x400 pixels).
- **setDefaultCloseOperation :** Définit l'action de fermeture (quitter l'application).
- **setLocationRelativeTo (null) :** Centre la fenêtre sur l'écran.

```
        setTitle("File Management System");
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
```

4. Création et configuration du panneau principal :

```
        JPanel panel = new JPanel();
        getContentPane().add(panel);
        panel.setLayout(null);
```

5. Création de l'arborescence des fichiers :

```
        DefaultMutableTreeNode root = new DefaultMutableTreeNode("Root");
        treeModel = new DefaultTreeModel(root);
        fileTree = new JTree(treeModel);
        JScrollPane treeScroll = new JScrollPane(fileTree);
        treeScroll.setBounds(10, 10, 200, 350);
        panel.add(treeScroll);
```

6. Bouton "Créer un fichier" :

```
        JButton createButton = new JButton("Créer un fichier");
        createButton.setBounds(220, 30, 200, 30);
        panel.add(createButton);

        createButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                createFile();
            }
        });
```

7. Bouton "lire un fichier" , Écrire dans un fichier, Supprimer un fichier, Créer un répertoire, Supprimer un répertoire, Lire un répertoire, Défragmenter :

```
        JButton readButton = new JButton("Lire un fichier");
        readButton.setBounds(220, 70, 200, 30);
        panel.add(readButton);

        readButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                readFile();
            }
        });


        JButton writeButton = new JButton("Écrire dans un fichier");
        writeButton.setBounds(220, 110, 200, 30);
        panel.add(writeButton);
        writeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                writeFile();
            }
        });

        JButton deleteButton = new JButton("Supprimer un fichier");
        deleteButton.setBounds(220, 150, 200, 30);
        panel.add(deleteButton);

        deleteButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                deleteFile();
            }
        });
```

```
        JButton createDirButton = new JButton("Créer un répertoire");
        createDirButton.setBounds(220, 190, 200, 30);
        panel.add(createDirButton);

        createDirButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                createDirectory();
            }
        });

        JButton deleteDirButton = new JButton("Supprimer un répertoire");
        deleteDirButton.setBounds(220, 230, 200, 30);
        panel.add(deleteDirButton);

        deleteDirButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                deleteDirectory();
            }
        });

        JButton readDirButton = new JButton("Lire un répertoire");
        readDirButton.setBounds(220, 270, 200, 30);
        panel.add(readDirButton);

        readDirButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                readDirectory();
            }
        });
```

```
        JButton defragButton = new JButton("Défragmenter");
        defragButton.setBounds(220, 310, 200, 30);
        panel.add(defragButton);

        defragButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                defragment();
            }
        });
    }
```

8. Méthode createFile :

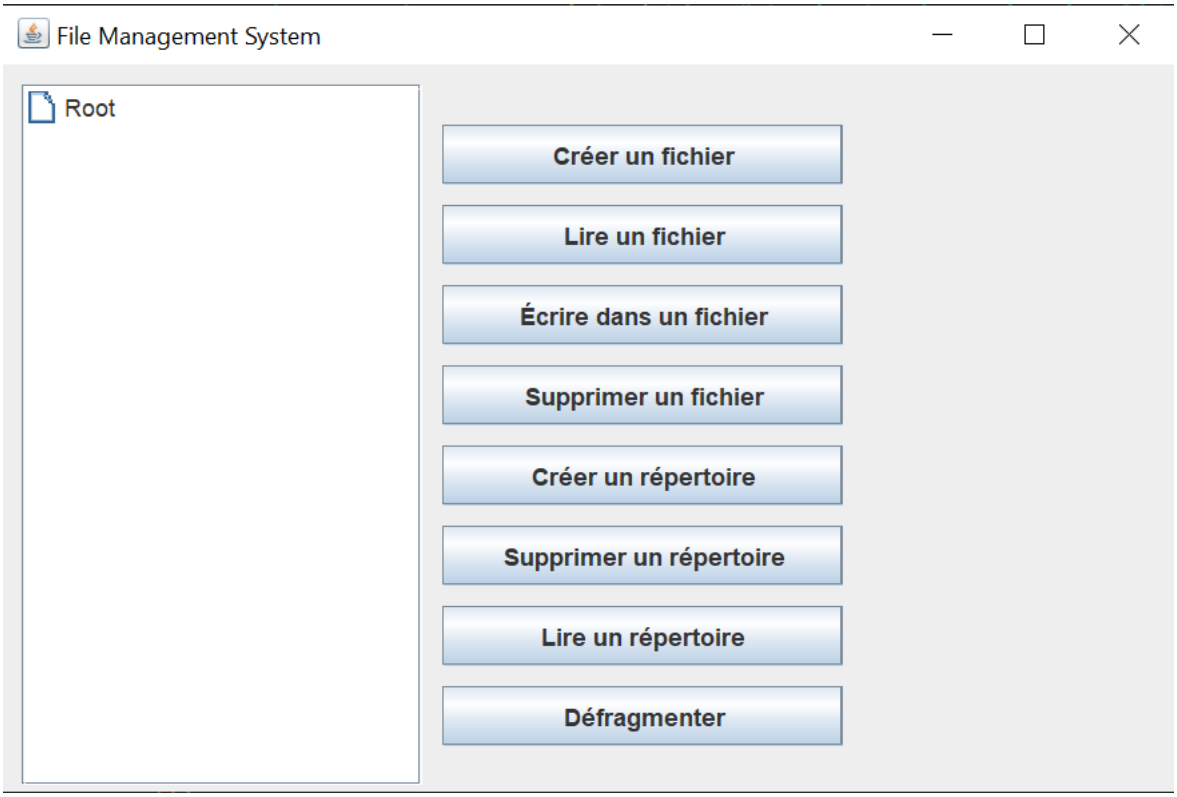
- **String fileName :** Affiche une boîte de dialogue pour demander le nom du fichier.
- Vérifie si le nom est valide et non vide.
- **fileOperations.createFile :** Crée le fichier.
- **diskManager.addFile :** Ajoute les métadonnées du fichier au gestionnaire de disque.
- **refreshFileTree () :** Rafraîchit l'arborescence des fichiers.
- Affiche un message d'erreur si la création échoue.

```
117    private void createFile() {
118        String fileName = JOptionPane.showInputDialog(this, "Nom du fichier:");
119        if (fileName != null && !fileName.trim().isEmpty()) {
120            if (fileOperations.createFile(fileName)) {
121                diskManager.addFile(new FileMetaData(fileName, 0, "rw-r--r--"));
122                refreshFileTree();
123            } else {
124                JOptionPane.showMessageDialog(this, "Échec de la création du fichier.");
125            }
126        } else {
127            JOptionPane.showMessageDialog(this, "Nom de fichier invalide.");
128        }
129    }
```

Les autres méthodes (writeFile, findFileByName, readFile, deleteFile, createDirectory, deleteDirectory, readDirectory, defragment, refreshFileTree) suivent des logiques similaires pour leurs actions respectives.

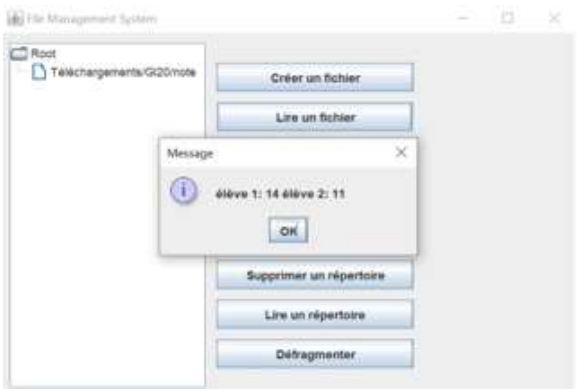
9. Captures d’écran :

- Interface utilisateur avant la création d’un fichier ou répertoire.



- Interface utilisateur pendant l’exécution (création d’un répertoire et d’un fichier) :

- Ecrire dans un fichier et lire le contenu d'un fichier et d'un répertoire :



- Supprimer un fichier et un répertoire :

Performances

Le projet de création d'un système de fichiers simplifié en mémoire présente de bonnes performances en termes de gestion de l'espace, rapidité des opérations et utilisation des ressources. La fiabilité est renforcée par la journalisation des opérations. L'interface graphique rend le système accessible et facile à utiliser. Cependant, il y a des possibilités d'améliorations, notamment en optimisant la structure de données pour la gestion des fichiers et en améliorant les algorithmes de défragmentation et de recherche.

1. Gestion de l'espace :

- **Allocation et libération de l'espace :** Le DiskManager gère efficacement l'allocation et la libération de l'espace. Lorsqu'un fichier est ajouté, l'espace est alloué et le journal enregistre cette opération. De même, la suppression d'un fichier libère l'espace et l'opération est également journalisée.

- **Défragmentation :** La méthode defragment trie les fichiers par date de création, ce qui aide à minimiser la fragmentation de l'espace mémoire. Cependant, pour une gestion plus avancée, une réorganisation physique des fichiers en mémoire pourrait être nécessaire.

2. Rapidité des opérations :

- **Création, suppression, lecture et écriture de fichiers :** Ces opérations sont généralement rapides car elles se font en mémoire. Le temps de réponse est donc très faible comparé à un système de fichiers sur disque.

- **Recherche de fichiers :** La recherche d'un fichier dans une liste peut devenir moins efficace à mesure que le nombre de fichiers augmente. Une structure de données plus avancée, comme un arbre ou une table de hachage, pourrait améliorer les performances de recherche.

4. **Fiabilité :**

- **Journalisation :** Le système inclut un journal qui enregistre les opérations importantes (création, suppression, allocation, etc.). Cette fonctionnalité améliore la fiabilité en permettant de récupérer l'état du système en cas de panne.

- **Gestion des erreurs :** Le projet inclut des mécanismes de gestion des erreurs, comme la vérification de l'existence des fichiers et des répertoires avant les opérations.

5. Interface utilisateur graphique (GUI) :

- **Facilité d'utilisation :** L'interface graphique permet aux utilisateurs de créer, supprimer, lire et écrire des fichiers de manière intuitive. Elle inclut également des fonctionnalités pour créer et supprimer des répertoires, ainsi que pour défragmenter le disque.

- **Réactivité :** L'interface graphique, construite avec Swing, est généralement réactive pour les opérations sur un petit nombre de fichiers. Cependant, pour une grande quantité de fichiers, il pourrait y avoir des ralentissements.

6. Extensions et fonctionnalités supplémentaires :

- **Système de fichiers journalisé :** La journalisation est implémentée, mais pourrait être améliorée avec des fonctionnalités supplémentaires, comme la journalisation des transactions pour les opérations complexes.

- **Fragmentation et défragmentation :** Bien que la défragmentation soit implémentée, une stratégie plus avancée pourrait être développée pour améliorer encore les performances.

Limitations et perspectives d’amélioration

1. Limitations :

- **Stockage en Mémoire Seulement** : Le système stocke tous les fichiers et répertoires en mémoire, ce qui limite sa capacité à grande échelle et sa persistance. Si l'application est fermée, toutes les données sont perdues.

- **Absence de Concurrence** : L'implémentation actuelle ne supporte pas l'accès concurrent. Plusieurs threads essayant de lire/écrire simultanément pourraient provoquer des corruptions de données.

- **Gestion Simpliste de l'Espace Disque** : La gestion de l'espace disque est basique et ne prend pas en compte les complexités réelles telles que la taille des blocs, la gestion des inodes ou les algorithmes d'allocation sophistiqués.

2. Perspectives d'Amélioration :

- **Stockage Persistant** : Intégrer des mécanismes de stockage persistant comme l'utilisation de bases de données sur disque ou la sérialisation pour maintenir l'état du système de fichiers entre les redémarrages de l'application.

- **Support de la Concurrence** : Implémenter des opérations sécurisées pour les threads et introduire des mécanismes de contrôle de la concurrence pour supporter l'accès multi-thread au système de fichiers.

- **Amélioration de la Gestion de l'Espace Disque** : Mettre en œuvre des techniques de gestion de l'espace disque plus sophistiquées, y compris l'allocation de blocs, les bitmaps d'espace libre ou l'allocation par liste chaînée pour gérer des besoins de stockage plus larges et plus complexes.