EPITA - Projet S4
Promo 2028

Projet

Linux File Manager

Cahier des charges

Groupe

WanderRust

Auteurs

Jean Philippe ZANGA
Clemence Dubouclard
Milan Muller-Lieberherr
Guillaume Gandon

Table des matières

Table des figures

1 Introduction

1.1 Nature du projet

Dans un monde numérique où les données se multiplient à une vitesse vertigineuse, la gestion efficace des fichiers est devenue une problématique centrale. Les utilisateurs, qu'ils soient professionnels ou particuliers, cherchent des solutions qui leur permettent de localiser, organiser et manipuler leurs fichiers avec aisance et rapidité. Cependant, les outils actuels présentent encore des limites, notamment en termes de personnalisation, de performances sur des volumes importants de données, et de sécurité.

Notre projet vise à développer un navigateur de fichiers moderne qui répond à ces enjeux.

1.2 Problématique

Comment concevoir un navigateur de fichiers capable de :

— Effectuer des recherches ultra-rapides parmi des millions de fichiers, même sur des systèmes aux ressources limitées ?
— Offrir des options avancées de tri et de filtrage qui s'adaptent aux besoins spécifiques de chaque utilisateur ?
— Prévisualiser une large variété de formats de fichiers tout en garantissant une légèreté d'exécution ?
— Assurer la sécurité des données et la stabilité du système pendant son exécution ?

1.3 Solutions Proposées

Pour relever ces défis, notre approche repose sur les points suivants :

— **Recherche optimisée** : Nous intégrerons un moteur de recherche basé sur des structures de données avancées telles que les arbres B-Tree et les tries, permettant des recherches rapides et précises.
— **Tri et filtrage personnalisés** : Des algorithmes efficaces comme Quicksort et Mergesort seront combinés à des interfaces intuitives pour offrir des outils de tri et de filtrage dynamiques.
— **Prévisualisation multi-format** : Des bibliothèques externes telles que Poppler (PDF) et ImageMagick (images) seront utilisées pour permettre une prévisualisation directe et fluide.
— **Usage de Rust** : Le choix du langage Rust nous permet de garantir des performances élevées et une sécurité maximale grâce à sa gestion stricte de la mémoire et de la concurrence.

1.4 Impact du Projet

Ce projet représente une opportunité unique de combiner des compétences techniques en algorithmique et en programmation avec des solutions pratiques aux problèmes rencontrés par les utilisateurs. Il valorise des notions apprises tout au long de la prépa, telles que les structures de données et les algorithmes, tout en explorant des technologies modernes comme Rust. Ce navigateur de fichiers a pour ambition de révolutionner la manière dont les utilisateurs interagissent avec leurs systèmes de fichiers.

2 État de l'art

Les références en terme de navigateurs de fichiers sont Finder sur MacOs, Thunar sur Linux et le plus vieux de tous, le navigateur Windows. Chacun présente plusieurs aspects intéressants qui complètent individuellement ceux des deux autres. Entrons dans une analyse comparative.

2.1 Finder (macOS)

Finder est le gestionnaire par défaut pour macOS, connu pour son intégration à l'écosystème Apple.

Points forts :

- **Interface utilisateur ergonomique :**
  - Design èpuré et intuitif facilitant la navigation.
  - Vue en colonnes efficace pour explorer les structures hiérarchiques complexes.
- **Recherche puissante :**
  - Basée sur Spotlight pour une recherche rapide et indexée.
  - Support des recherches avancées avec des critères comme le type de fichier ou la date.
- **Prévisualisation rapide avec Quick Look :**
  - Aperçu des fichiers sans ouverture de l'application native.
- **Intégration système :**
  - Synchronisation avec iCloud Drive.
  - Fonctionnalités de tags pour classer les fichiers.

Limites :

- **Performance :** Lent avec de grands volumes ou des disques externes non formatés en APFS.
- **Fonctionnalités limitées :** Pas de gestion avancée (ex. renommage massif).

— **Personnalisation :** Peu d'options pour configurer l'interface.

2.2 Thunar (Linux)

Thunar est le gestionnaire par défaut pour l'environnement XFCE, conçu pour être léger et rapide.

Points forts :

— **Légéreté et rapidité :** Consommation minimale de ressources.
— **Personnalisation :** Flexibilité via plugins.
— **Simplicité dútilisation :** Navigation intuitive et scripts personnalisés.

Limites :

— **Fonctionnalités limitées :** Pas de recherche indexée puissante.
— **Interface moins moderne :** Design basique.
— **Dépendance aux plugins :** Extensions souvent nécessaires pour combler les lacunes.

2.3 Explorateur de fichiers (Windows)

LÉxplorateur est le gestionnaire par défaut de Windows, présent depuis Windows 95.

Points forts :

— **Fonctionnalités étendues :**
  — Renommage massif, gestion d'archives ZIP.
  — Navigation rapide entre lecteurs locaux et réseaux.
— **Recherche intégrée :**
  — Basée sur Windows Search avec indexation rapide.
— **Support de multiples vues :** Icônes, liste détailée, etc.
— **Intégration système :**
  — Compatibilité avec OneDrive pour le cloud.
  — Gestion efficace des autorisations et partages.

Limites :

— **Performance :** Lent sur volumes non indexés ou systèmes anciens.
— **Complexité :** Interface parfois confuse pour les débutants.
— **Prévisualisation limitée :** Moins intuitive que Quick Look.

2.4 Conclusion

Finder se distingue par son intégration à l'écosystème Apple mais manque de personnalisation. Thunar excelle par sa légéreté et simplicité, mais reste

limité en termes de fonctionnalités. LÉxplorateur de Windows offre une solution équilibrée, mais souffre de lenteurs potentielles et d'une interface complexe. Ces analyses permettent d'identifier des améliorations potentielles pour un nouveau projet de gestionnaire de fichiers, notamment en combinant recherche rapide, personnalisation et interface ergonomique. Rust pourrait être un choix idéal pour relever ces défis avec performance et sécurité. Pour se projet, nous voulons donc nous inspirer des points forts de chacun des trois, dans la mesure du possible, afin d'apporter une nouvelle proposition.

3 Répartition des taches

3.1 Jean Philippe ZANGA

3.1.1 Développement du moteur de recherche et des fonctionnalités avancées

Dans le cadre du développement du navigateur de fichiers en Rust, l'implémentation d'un moteur de recherche efficace et de fonctionnalités avancées telles que le tri, le filtrage, et la prévisualisation des fichiers est cruciale pour une expérience utilisateur fluide. Cette section décrit les choix technologiques, les algorithmes de tri, et les structures de données utilisées pour implémenter ces fonctionnalités.

Le moteur de recherche doit permettre de localiser rapidement des fichiers en fonction de leur nom, type, taille ou contenu. Pour ce faire, il sera nécessaire d'implémenter un algorithme de recherche performant.

3.1.1.1 Bibliothèques utilisées

Pour faciliter l'implémentation du moteur de recherche, nous utiliserons les bibliothèques suivantes :

- `walkdir` : pour l'exploration récursive des répertoires et l'accès aux fichiers.
- `regex` : pour l'utilisation des expressions régulières permettant de rechercher des fichiers selon des motifs précis dans les noms de fichiers ou le contenu des fichiers.
- `fuzzy-search` : pour effectuer des recherches floues, permettant de trouver des fichiers même si la requête de recherche contient des erreurs de frappe.

3.1.1.2 Algorithmes de recherche

Le moteur de recherche peut utiliser les algorithmes suivants pour améliorer les performances :

— **Recherche binaire** : pour les recherches dans des listes de fichiers triées.

— **Recherche linéaire** : lorsque les fichiers ne sont pas triés, mais cette approche peut être améliorée en ajoutant un index inversé pour les fichiers les plus fréquemment consultés.

— **Recherche floue** : utilisant des algorithmes comme Levenshtein ou les arbres de préfixes (tries) pour effectuer des recherches robustes en cas de fautes de frappe.

3.1.2 Tri des fichiers

Le tri des fichiers permet de les organiser selon différents critères tels que le nom, la taille, la date de modification, ou encore l'extension. Le choix des algorithmes de tri est crucial pour garantir des performances optimales.

3.1.2.1 Algorithmes de tri

Les algorithmes de tri suivants seront utilisés en fonction de la taille et du type des données :

— **Tri rapide (QuickSort)** : utilisé pour son efficacité en moyenne (complexité en temps de $O(n \log n)$) et sa capacité à bien gérer des collections de fichiers volumineuses.
— **Tri fusion (MergeSort)** : utilisé lorsqu'une stabilité dans le tri est requise (par exemple, lors du tri des fichiers par date de modification).
— **Tri par tas (HeapSort)** : utilisé lorsque la mémoire est limitée et qu'un algorithme nécessitant une mémoire supplémentaire est à éviter.

3.1.2.2 Structures de données pour le tri

Les structures de données suivantes sont utilisées pour améliorer les performances de recherche et de tri :

— `Vec` : la structure la plus utilisée en Rust, elle permet de stocker et manipuler les fichiers dans une liste dynamique.

— `BTreeMap` : une structure de données basée sur un arbre binaire de recherche équilibré, permettant un tri automatique et une recherche rapide.
— `HashMap` : pour stocker des informations supplémentaires sur les fichiers (par exemple, index par taille ou date) et accéder rapidement à ces informations.

3.1.3 Filtrage des fichiers

Le filtrage des fichiers consiste à appliquer des critères comme le type de fichier (extension), la taille ou la date de modification. Cela permet à l'utilisateur de ne voir que les fichiers correspondant à ses critères spécifiques.

3.1.3.1 Algorithmes de filtrage

Les critères de filtrage les plus courants sont :
— Par extension de fichier : en utilisant des expressions régulières ou des chaînes de caractères pour filtrer les fichiers selon leur extension (`.txt`, `.jpg`, etc.).
— Par taille : en comparant la taille des fichiers avec les bornes spécifiées par l'utilisateur.
— Par date de modification : en comparant la date de modification des fichiers avec une plage temporelle donnée.

Le filtrage peut être optimisé par l'utilisation de structures comme `BTreeMap` ou `HashMap` pour associer rapidement les critères aux fichiers correspondants.

3.1.4 Prévisualisation des fichiers

La prévisualisation des fichiers permet à l'utilisateur de voir un aperçu de leurs contenus sans avoir à ouvrir le fichier dans un programme externe. Pour cela, différentes approches seront utilisées en fonction du type de fichier.

3.1.4.1 Bibliothèques utilisées pour la prévisualisation

Les bibliothèques suivantes seront utilisées pour générer les aperçus des fichiers :
— `image` : pour l'affichage des images.
— `pdf` : pour l'affichage des fichiers PDF.
— `tesseract` : pour la reconnaissance de texte dans les images.

3.1.4.2 Méthodologie de prévisualisation

Le moteur de prévisualisation choisira la méthode la plus adaptée en fonction du type de fichier. Par exemple, pour un fichier texte, le contenu sera extrait et affiché directement, tandis que pour une image, une réduction de la taille de l'image sera effectuée pour l'affichage rapide.

3.1.5 Conclusion

L'intégration de ces fonctionnalités dans le navigateur de fichiers assurera une expérience utilisateur riche et fluide. En utilisant des bibliothèques Rust de haute qualité et des algorithmes efficaces, nous garantirons des performances optimales et une gestion rapide des fichiers.

3.2 Clémence Dubouclard

3.2.1 Gestion des fichiers et des répertoires

Cette partie va aborder les différentes fonctionnalités à développer pour la gestion des fichiers et des répertoires dans un environnement Linux.

Objectifs :
Les principaux objectifs de cette partie sont :

- Une navigation fluide et intuitive dans l'arborescence des répertoires.
- La création et suppression de fichiers ou dossiers en toute sécurité.
- La gestion des modifications des permissions d'accès pour l'utilisateur et les groupes.
- La lecture et la manipulation des métadonnées des fichiers (tailles, date de modification, etc...)

3.2.1.1 La navigation dans les répertoires

L'utilisateur doit être en mesure de pouvoir parcourir l'arborescence de ses fichiers de manière efficace et intuitive.

Pour cela, il est nécessaire de pouvoir lister les fichiers et dossiers présents dans un répertoire, charger dynamiquement les sous-dossiers lorsque un ré-

pertoire est exploré.
Il faut aussi tenir compte des permissions des utilisateurs et ne pas donner accès à des fichiers où l'accès lui serait interdit.

Pour gérer les erreurs, il faut faire attention aux fichiers inexistants ou qui ont été déplacés, il faut aussi gérer les fichiers supprimés et veiller à les placer dans une "corbeille temporaire" pour une récupération éventuelle.

3.2.1.2 La gestion des permissions

L'utilisateur doit être capable de pouvoir voir les différentes permissions qu'il possède sur ses différents fichiers de façon lisible et claire.

Il doit pouvoir aussi les modifier, ajouter et supprimer que ce soit pour lui ou pour les groupes. Autrement dit, des actions qui sont assez similaires à la commande chmod.

3.2.1.3 La gestion des métatdonnées

L'utilisateur doit pouvoir les différentes métadonnées attribuées aux fichiers telles que leur taille, leur date de création, de dernière modification, de dernier accès ainsi que le type de fichiers (texte, image, dossier, etc).

Il peut aussi en ajouter lorsque cela est possible pour par exemple simplifier la recherche des documents. Son affichage doit être clair afin de ne pas s'y perdre.

3.2.1.4 La création et la suppression des fichiers

L'utilisateur doit pouvoir créer des fichiers ou des répertoires par le biais d'un chemin donné.

Pour cela, il faut vérifier le nom des fichiers entrés afin de s'assurer qu'ils ne sont pas en collision avec d'autres fichiers déjà existants, éviter les caractères interdits tout en tenant compte des permissions.

Il est nécessaire de sécuriser la suppression des fichiers ou dossiers en demandant confirmation à l'utilisateur.

3.2.1.5 Les différentes bibliothèques

Pour implémenter les différentes fonctionnalités liées à la gestion des fichiers et des répertoires, Rust propose un écosystème riche de bibliothèques, adaptées aux environnements Linux.

Pour effectuer des opérations courantes sur les systèmes de fichiers, on peut utiliser le module std : :fs qui offre les outils nécessaires pour effectuer des opérations courantes sur les systèmes de fichiers.

Des bibliothèques tierces comme **walkdir** ou **glob** sont particulièrement utiles. La première permet de simplifier le parcours récursif des répertoires en chargeant dynamiquement les sous-dossiers et en gérant automatiquement les erreurs liées à l'accès à des fichiers non valides. Quant à glob, elle permet d'effectuer des recherches de fichiers en se basant sur des motifs définis par des caractères génériques.

À l'aide de **std : : fs : : metadata()**, il est possible d'accéder à des informations basiques telles que la taille d'un fichier, son type ou ses dates de création et de modification. Et pour enrichir cette gestion, la bibliothèque **xattr** permet d'interagir avec les attributs étendus des fichiers. Ces attributs, spécifiques aux systèmes de fichiers Linux, permettent de stocker des informations supplémentaires comme des tags.

Certaines tâches spécifiques, comme la gestion des fichiers temporaires ou la manipulation asynchrone des données, peuvent être optimisées grâce à des bibliothèques dédiées. **tempfile**, par exemple, permet de créer des fichiers temporaires de manière tout à fait sécurisée, ce qui peut être utile pour l'implémentation d'une corbeille temporaire.

Tokio ou **async-std**, permettent quant à eux d'exécuter des tâches lourdes, comme la suppression en masse ou la recherche dans des répertoires volumineux, de manière assez fluide le tout sans interrompre le fonctionnement global du programme.

3.3 Milan Muller-Lieberherr

3.3.1 Interface Utilisateur

Postulat :

Cette partie a pour vocation de détailler l'ensemble des exigences, spécifications et orientations techniques relatives à la conception et à la mise en œuvre de l'interface utilisateur (UI) d'une application d'exploration et de gestion de fichiers développée en langage Rust. Ce projet vise à offrir aux utilisateurs une solution intuitive, performante et efficace pour parcou-

rir, organiser et manipuler leurs fichiers et répertoires de manière optimale. L'objectif est de fournir un cadre clair et structuré permettant d'assurer une réalisation conforme aux attentes en matière d'ergonomie, de performance et de maintenabilité.

Objectifs :
L'objectif principal est de concevoir et de développer une interface utilisateur qui soit à la fois réactive, intuitive et ergonomique, afin d'offrir une expérience utilisateur optimale lors de la gestion des fichiers et répertoires. Il s'agit notamment d'intégrer une bibliothèque graphique qui réponde aux besoins spécifiques du projet, tout en assurant une gestion efficace et robuste des différents événements utilisateur. Par ailleurs, l'UI vise à fournir des fonctionnalités interactives permettant une navigation fluide et agréable au sein de l'application, en tenant compte des bonnes pratiques en matière d'UX/UI design.

3.3.2 Spécifications fonctionnelles

3.3.2.1 Exigences générales

L'interface utilisateur devra répondre aux critères suivants :

— Elle devra être intuitive afin de garantir une prise en main rapide et efficace par les utilisateurs finaux, qu'ils soient novices ou expérimentés.

— Elle devra être responsive, c'est-à-dire s'adapter automatiquement à différentes résolutions d'écran et dispositifs (ordinateurs de bureau, tablettes, smartphones, etc.), offrant ainsi une flexibilité d'utilisation accrue.

— Elle devra fonctionner sur Linux.

— La gestion des événements devra couvrir l'ensemble des interactions utilisateur, telles que les entrées clavier, les clics souris pour la sélection et les interactions tactiles pour les dispositifs compatibles.

— La conception devra être modulaire, afin de permettre des évolutions et des ajouts de fonctionnalités dans le futur sans compromettre l'architecture initiale.

3.3.2.2 Composantes de l'UI

L'interface utilisateur devra inclure une gamme complète de composants permettant une gestion efficace des fichiers et répertoires. Parmi ces composants, on retrouvera notamment :

— **Une barre de navigation** permettant un accès rapide aux répertoires les plus fréquemment utilisés et aux emplacements favoris.

— **Une vue en liste et en grille** offrant aux utilisateurs la possibilité de visualiser leurs fichiers sous différentes formes en fonction de leurs préférences.

— **Un système de recherche avancée** permettant de localiser rapidement des fichiers en fonction de critères tels que le nom, l'extension, la taille et la date de modification.

— **Des boutons d'action** dédiés aux opérations courantes telles que la copie, le déplacement, la suppression et le renommage de fichiers.

— **Une gestion des thèmes et du style visuel** offrant la possibilité de personnaliser l'apparence de l'application selon les préférences de l'utilisateur.

3.3.3 Choix des Technologies

L'interface utilisateur sera développée en langage Rust, en s'appuyant sur une bibliothèque graphique qui devra répondre aux exigences du projet en matière de performance et de flexibilité. Plusieurs options technologiques sont envisagées, parmi lesquelles :

— **Druid** : une bibliothèque axée sur une architecture basée sur les données, offrant une approche réactive et modulaire, bien adaptée aux besoins d'un gestionnaire de fichiers.

— **Iced** : une solution inspirée du paradigme déclaratif de Elm, permettant de construire des interfaces utilisateur modernes et évolutives, avec un bon support pour la gestion des événements.

— **Egui** : une bibliothèque mettant l'accent sur la rapidité d'exécution

et la simplicité d'intégration grâce à son approche Immediate Mode GUI, permettant une mise en œuvre rapide de l'interface utilisateur.

3.3.4 Architecture et Technologies

Le langage de programmation Rust sera utilisé pour assurer robustesse et sécurité, tandis que des tests unitaires et d'intégration seront mis en place afin de valider le bon fonctionnement des différents composants de l'UI avant leur mise en production.

3.3.5 Conclusion

L'interface utilisateur du gestionnaire de fichiers devra être développée avec soin afin de garantir une expérience fluide et agréable aux utilisateurs. Le choix de la bibliothèque Rust appropriée sera déterminant pour assurer un bon compromis entre performance, flexibilité et facilité de maintenance. Un prototypage initial sera réalisé afin de valider les choix technologiques et d'ajuster l'approche en fonction des premiers retours utilisateurs. Des tests rigoureux seront également effectués pour s'assurer de la stabilité et de la fiabilité de l'application.

3.4 Guillaume Gandon

3.4.1 Gestion du dépôt Git - Responsable de la tâche

En tant que responsable de la gestion du dépôt Git pour ce projet, il est de ma responsabilité de m'assurer que toutes les modifications du code source sont bien suivies, documentées et intégrées de manière fluide dans notre flux de travail collaboratif. Ce processus inclut plusieurs tâches critiques telles que l'initialisation du dépôt, la gestion des branches, la création de commits, la fusion des modifications et la résolution des conflits. Voici un résumé des tâches effectuées, des difficultés rencontrées et des ressources mises en place pour y remédier.

3.4.1.1 Tâches réalisées

La gestion du dépôt Git pour ce projet comprend plusieurs aspects essentiels :

— **Initialisation du dépôt** : Le dépôt Git a été initialisé sur la machine locale de chaque membre, avec une première configuration du dépôt distant sur GitHub (ou autre plateforme).

— **Suivi des fichiers** : Tous les fichiers nécessaires ont été ajoutés au suivi de Git à l'aide de la commande `git add`. Des fichiers inutiles ont été exclus grâce au fichier `.gitignore`.
— **Création de commits** : Nous avons créé des commits réguliers, chacun étant associé à des messages clairs et descriptifs. Cette pratique garantit que chaque changement est bien documenté dans l'historique.
— **Gestion des branches** : Des branches ont été créées pour chaque fonctionnalité ou tâche spécifique, afin de travailler de manière indépendante et de faciliter l'intégration des modifications.
— **Fusion des branches** : Les branches développées ont été fusionnées dans la branche principale avec `git merge`. Des tests ont été effectués après chaque fusion pour s'assurer que le code fonctionne correctement.
— **Utilisation des dépôts distants** : Des interactions avec le dépôt distant ont été réalisées via `git push` et `git pull`, assurant que chaque membre puisse partager ses modifications avec le groupe et synchroniser les travaux.

3.4.1.2 Difficultés rencontrées

Malgré une gestion efficace du dépôt, plusieurs difficultés peuvent émerger, liées notamment aux défis de la collaboration simultanée sur des parties du code. Voici les principales difficultés que nous avons rencontrées :

— **Conflits de fusion** : Lors de la fusion des branches, des conflits de code sont survenus, surtout lorsque plusieurs membres ont modifié les mêmes lignes dans un même fichier. Ces conflits ont nécessité une résolution manuelle.
— **Mauvaise gestion des commits** : Dans certaines situations, des commits ont été réalisés sans messages adéquats ou ont inclus des changements inutiles, ce qui a compliqué la compréhension de l'historique du projet.
— **Problèmes de synchronisation avec le dépôt distant** : À certains moments, des divergences ont eu lieu entre les versions locales et distantes du code, causant des erreurs lors des `git pull` ou `git push`.

3.4.1.3 Ressources et solutions pour surmonter les difficultés

Pour remédier à ces difficultés, plusieurs actions ont été entreprises et des ressources ont été mises en place afin de garantir une gestion fluide du dépôt et de maintenir la productivité de l'équipe :

— **Formation et bonnes pratiques Git** : Des ressources sur l'utilisation optimale de Git, comme des tutoriels internes et des guides sur l'écriture de bons messages de commit, ont été partagées avec les membres de l'équipe.
— **Outils de résolution de conflits** : Des sessions de travail collaboratif ont été organisées pour résoudre les conflits de manière efficace. En outre, des outils graphiques comme GitKraken ou SourceTree ont été recommandés pour faciliter la gestion des conflits.
— **Revue de code régulière** : Pour éviter les erreurs lors des fusions, des revues de code ont été mises en place afin de s'assurer que chaque commit et chaque fusion est examiné par un autre membre de l'équipe.
— **Mise en place de branches de développement et de production** : Une structure de branches claire a été adoptée, avec des branches dédiées à chaque fonctionnalité et des branches de production pour stabiliser le code. Cela a réduit les risques de conflits.
— **Sauvegardes et sauvegarde de l'historique** : Des snapshots réguliers du dépôt ont été effectués avant chaque modification majeure, garantissant ainsi que l'équipe puisse revenir facilement à une version antérieure en cas de problème.

La gestion efficace du dépôt Git est essentielle pour le bon déroulement de notre projet. Grâce à des pratiques de gestion rigoureuses et à l'utilisation des ressources appropriées, nous avons pu surmonter les difficultés rencontrées et maintenir une collaboration fluide au sein de l'équipe. La mise en place d'outils et de bonnes pratiques continue d'être un élément clé de notre processus de développement.

3.4.2 Tests et Optimisations des différentes parties

Dans un projet de développement logiciel en Rust, les tests et les optimisations sont des étapes essentielles pour garantir la qualité, la fiabilité et la performance du logiciel. Rust, en tant que langage performant, permet une gestion fine de la mémoire et des ressources, ce qui facilite l'optimisation. Voici une description détaillée des tests et des optimisations spécifiquement appliquées à un projet en Rust.

3.4.2.1 Tests dans un projet en Rust

Les tests sont utilisés pour s'assurer que le logiciel fonctionne correctement et respecte les exigences. Rust propose un excellent cadre de test intégré, facilitant la création, l'exécution et l'organisation des tests.

3.4.2.2 Tests unitaires

Les tests unitaires en Rust sont écrits dans le même fichier que le code à tester, mais dans un module séparé. Ils permettent de tester des fonctions ou des méthodes de manière isolée. Rust utilise l'annotation `[cfg(test)]` pour inclure les tests uniquement lors de l'exécution des tests.

- **Exemple** : Tester une fonction qui effectue un calcul, comme une fonction qui additionne deux nombres.
- **Commande** : `cargo test` pour exécuter les tests.

3.4.2.3 Tests d'intégration

Les tests d'intégration sont écrits dans des fichiers séparés sous le dossier `tests`. Ils vérifient l'interaction entre plusieurs unités de code, comme l'accès aux bases de données ou les appels réseau.

- **Exemple** : Tester si une fonction qui insère des données dans une base de données fonctionne correctement lorsqu'elle est utilisée dans le contexte global de l'application.
- **Commande** : `cargo test` pour exécuter tous les tests, y compris ceux d'intégration.

3.4.2.4 Tests de performance

Les tests de performance en Rust peuvent être réalisés avec la bibliothèque `criterion.rs`, qui permet de mesurer précisément le temps d'exécution des fonctions et d'obtenir des benchmarks.

- **Exemple** : Mesurer les performances d'un algorithme de tri ou d'une fonction de traitement de données volumineuses.
- **Commande** : `cargo bench` pour exécuter les tests de performance.

3.4.3 Optimisations

Rust est un langage conçu pour offrir de hautes performances tout en garantissant la sécurité mémoire. Cependant, plusieurs techniques d'optimisation peuvent être appliquées pour améliorer la vitesse, l'efficacité, et l'utilisation des ressources.

3.4.3.1 Optimisation des performances

Rust permet un contrôle fin sur la performance du code grâce à des outils de profilage et des optimisations spécifiques au code.

— **Optimisation des algorithmes** : Utiliser des algorithmes plus efficaces pour résoudre un problème (par exemple, passer à un tri rapide au lieu d'un tri à bulles).
— **Utilisation de types de données appropriés** : Choisir des types de données adaptés aux besoins de l'application, comme utiliser `usize` ou `u32` pour les indices.
— **Minimisation des allocations** : Réduire les allocations de mémoire inutiles en utilisant des structures comme `Vec` et en réutilisant les objets via des références ou des pointeurs.

3.4.3.2 Optimisation de la mémoire

Rust utilise un modèle de gestion de la mémoire basé sur la possession et les emprunts, ce qui réduit les fuites de mémoire. Toutefois, il est possible d'optimiser davantage l'utilisation de la mémoire :
— **Éviter les allocations inutiles** : Utiliser `str` au lieu de `String` si les chaînes de caractères sont immutables.
— **Réduire l'utilisation des** `Box` : Les `Box` allouent de la mémoire sur le tas et leur utilisation excessive peut entraîner des surcharges.

3.4.3.3 Optimisation de la concurrence

Rust permet de gérer la concurrence de manière sûre grâce à son système de possession, ce qui réduit les risques d'erreurs liées à l'accès concurrent à la mémoire.
— **Utilisation de** `async/await` : Le modèle asynchrone de Rust permet d'écrire des programmes hautement concurrents sans blocage des threads.
— **Optimisation des threads** : Utiliser des threads avec `std::thread` ou des bibliothèques comme `rayon` pour le traitement parallèle peut améliorer les performances dans des applications multithread.

3.4.4 Optimisation de la compilation

Rust offre plusieurs niveaux d'optimisation pendant la compilation, ce qui permet de réduire la taille du binaire et d'améliorer les performances.
— **Optimisation de la compilation** : Utiliser `cargo build -release` pour compiler le projet avec des optimisations pour la production.
— **Profilage avec** `cargo flamegraph` : Outil permettant de générer des graphiques pour identifier les parties du code les plus coûteuses en termes de performances.

3.4.4.1 Optimisation de la gestion des erreurs

Rust gère les erreurs avec les types `Result` et `Option`, ce qui garantit une gestion rigoureuse des erreurs. Cependant, il existe des pratiques d'optimisation pour rendre cette gestion encore plus efficace :

- **Éviter les `unwrap` et `expect`** : Ces méthodes peuvent entraîner des plantages si une erreur survient. Il est préférable d'utiliser `match` ou `unwrap_or_else`.
- **Utiliser des erreurs spécifiques** : Créer des types d'erreurs personnalisés pour une gestion plus fine des erreurs.

3.4.5 Conclusion

Dans un projet en Rust, les tests et les optimisations sont des processus essentiels pour garantir la qualité et la performance du logiciel. Grâce aux outils de tests intégrés et aux pratiques d'optimisation de la mémoire, de la concurrence et des performances, Rust permet de développer des applications sûres et efficaces. Les tests unitaires, d'intégration, de performance et de sécurité assurent que le code fonctionne comme prévu, tandis que les optimisations permettent d'obtenir un logiciel performant, scalable et facile à maintenir.

4 Estimation pour la première soutenance

Tâches	Prévisions
La navigation dans les répertoires	55%
La gestion des permissions	50%
La gestion des métadonnées	60%
La création et la suppression des fichiers	55%
vue en liste et en grille	45%
système de recherche avancée	50%
boutons d'action	65 %
Tri et filtrage des fichiers	75%
Interface graphique	65%

5 Conclusion

En conclusion , nous espérons pouvoir réaliser un bon nombre d'étapes essentielles, notamment l'interface utilisateur, la gestion des fichiers et quelques

fonctionnalités avancées comme la recherche pour pouvoir faire foi d'une bonne organisation, de la cohésion de notre groupe et d'une volonté manifeste de réussir ce semestre. Nous nous laissons une bonne marge de progression entre les trois soutenances.

Cependant, ce projet constitue sérieux challenge, compte tenu de la proximité des trois soutenances, de la charge de travail associée aux autres cours et autres projets et enfin, les problèmes de coordinations et les imprévus, y compris le climat qui est propice aux maladies.

Nous sommes confiants dans notre capacité à atteindre ces objectifs d'ici la soutenance finale et attendons de pied ferme la première soutenance