**File Explorer Application**

Group Members Name:

1.Souvik Dawn

2.Praneeth Reddy

3.Dhanashri Punde

Contents

[1. Project Overview: 3**.**](#_Toc173513456)

[2. Objectives: **3.**](#_Toc173513457)

[3. Features: **3.**](#_Toc173513458)

4.[Command Execution: **3.**](#_Toc173513459)

[5. Functional Requirements: **4.**](#_Toc173513465)

[6. Non-Functional Requirements: **4.**](#_Toc173513472)

[Performance: **4.**](#_Toc173513473)

[Usability: **4.**](#_Toc173513474)

[Reliability: **4.**](#_Toc173513475)

[Compatibility: **4.**](#_Toc173513476)

[7. Technologies Used: **4.**](#_Toc173513477)

[Programming Language: **4.**](#_Toc173513478)

[System Calls: **4.**](#_Toc173513479)

[Data Structures: **4.**](#_Toc173513480)

[8. Project Phases: **5.**](#_Toc173513481)

[Phase 1: Requirements Gathering **5.**](#_Toc173513482)

[Phase 2: Design **5.**](#_Toc173513483)

[Phase 3: Implementation **5.**](#_Toc173513484)

[Phase 4: Testing **5.**](#_Toc173513485)

[Phase 5: Documentation **5.**](#_Toc173513486)

[9. System Design: **6.**](#_Toc173513487)

[a. Overview of the components **7.**](#_Toc173513488)

[b. Data Structures: **9.**](#_Toc173513489)

[c. Key Functions: **9.**](#_Toc173513490)

[10. Test Cases: 11**.**](#_Toc173513491)

[11. Failure Cases: **12.**](#_Toc173513492)

[12. Skeleton Code: 13**.**](#_Toc173513498)

**1. Project Overview**

The File Explorer application is a console-based tool that provides users with an intuitive and efficient interface for managing files and directories in a Linux environment. Built using C++, the application leverages the C++ Standard Library, particularly the <filesystem> module, to handle various file operations seamlessly. The application also incorporates logging functionality to track user activities, essential for debugging, auditing, and ensuring the application's reliability.

**2. Objectives**

The primary objective of this capstone project is to develop a robust and user-friendly console-based file explorer application. The application will interact with the Linux operating system to execute essential file management tasks such as listing files, navigating directories, manipulating files (creating, copying, moving, deleting), searching for files, and managing file permissions, all through a command-line interface.

**3. Features**

**File Operations:**

* **Create Files:** Users can create new files within the current directory or a specified path.
* **Delete Files:** Files can be deleted permanently from the file system.
* **Copy Files:** Users can copy files from one location to another.
* **Move Files:** Files can be moved from one directory to another.
* **Search Files:** Users can search for specific files within the directory structure.
* **Change File Permissions:** Modify the permissions of files to control access.

**Directory Navigation:**

* **List Files:** Display a list of files in the current or specified directory.
* **Navigate Directories:** Move through directories, either by specifying a path or using relative paths.
* **Go Up One Directory Level:** Easily navigate to the parent directory.

**4. Command Execution**

Each file operation or directory navigation task is treated as a command. The FileExplorer class is responsible for processing these commands, executing the appropriate system calls, and handling any errors that may arise during execution.

**5. Functional Requirements**

* **Command Parsing:** The application reads user inputs from the command line, interprets them as commands, and executes the corresponding file or directory operations.
* **Error Handling:** The application should gracefully handle invalid commands or inputs, providing helpful error messages and logging the issues for later review.
* **Logging:** All operations and errors should be logged to a file for auditing and debugging purposes.

**6. Non-Functional Requirements**

* **Performance:** The application must handle file and directory operations quickly and efficiently, even when dealing with large numbers of files or directories.
* **Usability:** The command-line interface should be intuitive, providing clear instructions and options for the user, with minimal learning curve required.
* **Reliability:** The application must ensure data integrity during file operations, preventing data loss or corruption. It should also handle unexpected situations, such as unavailable files or directories, without crashing.
* **Compatibility:** The application is designed to run on Linux systems, leveraging standard C++ libraries and system calls. It should work seamlessly across different Linux distributions.

**7. Technologies Used**

* **Programming Language:** The application is written in C++, utilizing the C++ Standard Library, particularly <filesystem>, <iostream>, and <fstream>.
* **System Calls:** The application uses basic file I/O operations (such as open, close, chmod) and directory navigation functions (such as chdir, getcwd).
* **Data Structures:** Standard C++ containers and data types, including std::string for handling text, std::ofstream for logging, and std::filesystem::path for managing file paths.

**8. Project Phases**

**Phase 1: Requirements Gathering**

* **Functional Requirements:** Determine the file and directory operations that the application should support.
* **System Requirements:** Identify the system calls and libraries required for the application to interact with the Linux file system.

**Phase 2: Design**

* **Class Design:** Design the Logger and FileExplorer classes, defining their responsibilities and interactions.
* **Error Handling:** Plan for comprehensive error handling and logging mechanisms.
* **User Interface:** Design the command-line interface, ensuring it is user-friendly and intuitive.

**Phase 3: Implementation**

* **Core Functionalities:** Implement the core file and directory operations, focusing on correctness and efficiency.
* **Command Interface:** Develop the user interface in main.cpp, ensuring seamless interaction between the user and the application.
* **Integration:** Integrate the Logger and FileExplorer classes, ensuring they work together to manage files and log operations.

**Phase 4: Testing**

* **Unit Testing:** Test individual functions to ensure they perform as expected.
* **Integration Testing:** Test the application as a whole, verifying that different components work together without issues.
* **Performance Testing:** Ensure the application performs well under various scenarios, including large directories and heavy file operations.

**Phase 5: Documentation**

* **User Documentation:** Provide detailed instructions on how to use the application, including command descriptions and examples.
* **Code Documentation:** Comment the code extensively to ensure maintainability and ease of understanding for future developers.

**9. System Design**

* **High-Level Design:**

The high-level design outlines the major components and their interactions.

**Main Program:**

Responsibilities: Handles user inputs, commands, and controls the overall flow of the application.

Components:

User Interface: Provides a command-line interface for user interactions.

Command Parser: Parses and interprets user commands.

**File Explorer Module**:

Responsibilities: Manages file and directory operations.

Components:

File Operations: Handles basic operations like listing, creating, copying, moving, and deleting files.

Directory Operations: Manages directory navigation and listing.

Search Functionality: Searches for files within directories.

Permission Management: Sets file permissions.

**Logger Module:**

Responsibilities: Logs important events and actions to a file.

Components:

Log Writer: Writes log messages to a file with timestamps.

* **Low-Level Design:**

The low-level design provides detailed descriptions of each module's classes, functions, and data flow.

**FileExplorer Class:**

Methods:

listFiles (const std::string& directory): Lists all files in the specified directory.

changeDirectory (const std::string& directory): Changes the current working directory.

searchFiles (const std::string& directory, const std::string& filename): Searches for a file within the directory.

setPermissions (const std::string& filename, int mode): Sets permissions for the specified file.

createFile(const std::string& filename): Creates a new file.

copyFile (const std::string& source, const std::string& destination): Copies a file from the source to the destination.

moveFile (const std::string& source, const std::string& destination): Moves a file from the source to the destination.

deleteFile (const std::string& filename): Deletes the specified file.

**Logger Class:**

Methods:

Logger(const std::string& filename): Constructor that opens a log file.

~Logger(): Destructor that closes the log file.

log(const std::string& message): Logs a message with a timestamp.

**Main Function:**

Logic:

Initialize Logger and FileExplorer objects.

Run a loop to receive and process user commands.

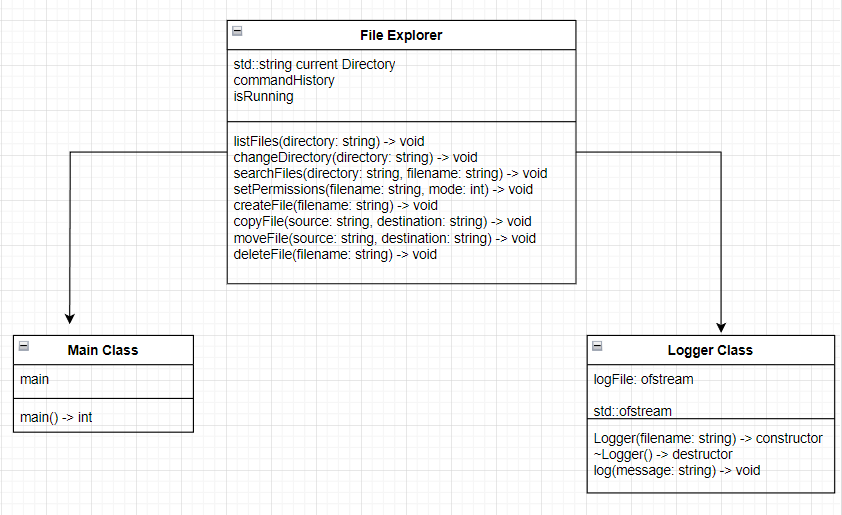
Call appropriate methods in the FileExplorer class based on user input

Log actions using the Logger class

**a. Overview of the Components:**

* **Logger Class:** Handles logging of operations and errors to a log file, providing an audit trail and aiding in debugging.
* **FileExplorer Class:** Manages file and directory operations, executing the necessary system calls and managing the interaction with the Linux file system.
* **main.cpp:** Serves as the entry point of the application, providing the command-line interface and handling user input.

**Class Diagram:**

****

**Activity Diagram:**

****

**b. Data Structures:**

* **std::string:** Used for storing file and directory names.
* **std::ofstream:** Used for writing log entries to a file.
* **std::filesystem::path:** Used for handling file paths, offering cross-platform path manipulation.

**c. Key Functions:**

* **listFiles(const std::string& directory):** Lists all files in the specified directory.
* **copyFile(const std::string& src, const std::string& dest):** Copies a file from the source path to the destination path.
* **moveFile(const std::string& src, const std::string& dest):** Moves a file from the source path to the destination path.
* **deleteFile(const std::string& path):** Permanently deletes a file from the specified path.
* **createFile(const std::string& path):** Creates a new file at the specified path.
* **searchFile(const std::string& filename):** Searches for a file within the directory structure.
* **setPermissions(const std::string& path, int permissions):** Changes the permissions of the specified file.
* **changeDirectory(const std::string& directory):** Changes the current working directory to the specified path.
* **goUpOneDirectory():** Moves up one directory level from the current directory.

**10. Test Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case ID | Description | Expected Result | Actual Result | Status |
| TC01 | Execute a command for listing Files | Command executes successfully | Pass |  |
| TC02 | Execute a Command Copy File | Command executes successfully | Pass |  |
| TC03 | Execute a Command Move File | Command executes successfully | Pass |  |
| TC04 | Execute a Command Delete File | Command executes successfully | Pass |  |
| TC05 | Execute a Command Create File | Command executes successfully | Pass |  |
| TC06 | Execute a Command Search File | Command executes successfully | Pass |  |
| TC07 | Execute a command Set File Permissions | Command executes successfully | Pass |  |
| TC08 | Execute a Command Change Directory | Command executes successfully | Pass |  |
| TC09 | Execute a Command GoupOneDirectory | Command executes successfully | Pass |  |

**11.Failure Cases:**

**Incorrect Directory Navigation in goUpOneDirectory Method**

* **Description:** The goUpOneDirectory method may cause the program to navigate incorrectly if the current directory is the root directory.
* **Steps to Reproduce:**
  1. Run the File Explorer.
  2. Choose the option to go up one directory level from the root directory (e.g., /).
* **Expected Behavior:** The method should prevent navigating up from the root directory and log an appropriate message.
* **Actual Behavior:** The method might attempt to change the directory to an invalid path, resulting in an error or crash.
* **Possible Fix:** Implement a check to determine if the current directory is the root directory before navigating up. If it is, log an appropriate message and prevent the directory change.
* **Status:** Open

### **Unhandled Exception when Copying a Non-existent File**

* **Description:** The copy File method does not handle cases where the source file does not exist, leading to an application crash.
* **Steps to Reproduce:**
  1. Run the File Explorer.
  2. Choose the option to copy a file.
  3. Enter a non-existent source file path.
* **Expected Behavior:** The method should handle the error gracefully and log an appropriate message.
* **Actual Behavior:** The application crashes.
* **Possible Fix:** Add a check using std::filesystem::exists(src) to verify if the source file exists before attempting to copy it. Log an error message if the file does not exist.
* **Status:** Open

**12. Skeleton Code:**

#include <iostream>

#include "FileExplorer.h"

#include "Logger.h"

int main()

{

// Create a Logger object to handle logging

Logger logger("file\_explorer.log");

// Create a FileExplorer object, passing the logger to it

FileExplorer explorer(logger);

// Variables to store user input

int choice;

std::string path, src, dest, filename;

int permissions;

do

{

// Display the current directory and options to the user

std::cout << "\nCurrent Directory: " << std::filesystem::current\_path() << std::endl;

std::cout << "File Explorer Options:\n";

std::cout << "1. List Files in Directory\n";

std::cout << "2. Copy File\n";

std::cout << "3. Move File\n";

std::cout << "4. Delete File\n";

std::cout << "5. Create File\n";

std::cout << "6. Search File\n";

std::cout << "7. Set File Permissions\n";

std::cout << "8. Change Directory\n";

std::cout << "9. Go Up One Directory\n";

std::cout << "10. Exit\n";

std::cout << "Enter your choice: ";

std::cin >> choice;

// Switch case to handle user input

switch (choice)

{

case 1:

std::cout << "Enter directory path: ";

std::cin >> path;

explorer.listFiles(path);

break;

case 2:

std::cout << "Enter source file name: ";

std::cin >> src;

std::cout << "Enter destination file name: ";

std::cin >> dest;

explorer.copyFile(src, dest);

break;

case 3:

std::cout << "Enter source file name: ";

std::cin >> src;

std::cout << "Enter destination file name: ";

std::cin >> dest;

explorer.moveFile(src, dest);

break;

case 4:

std::cout << "Enter file name to delete: ";

std::cin >> path;

explorer.deleteFile(path);

break;

case 5:

std::cout << "Enter file name to create: ";

std::cin >> path;

explorer.createFile(path);

break;

case 6:

std::cout << "Enter file name to search: ";

std::cin >> filename;

explorer.searchFile(filename);

break;

case 7:

std::cout << "Enter file name to set permissions: ";

std::cin >> path;

std::cout << "Enter permissions (e.g., 755): ";

std::cin >> permissions;

explorer.setPermissions(path, permissions);

break;

case 8:

std::cout << "Enter directory to navigate to: ";

std::cin >> path;

explorer.changeDirectory(path);

break;

case 9:

explorer.goUpOneDirectory();

break;

case 10:

std::cout << "Exiting...\n";

break;

default:

std::cout << "Invalid choice. Please try again.\n";

}

}

while (choice != 10); // Loop until the user chooses to exit

return 0;

}