**File Explorer Application**

**Contents**

1. **Objective**
2. **Introduction**
3. **High-Level Design (HLD)**
   * **Overview**
   * **Components**
   * **Flowchart**
4. **Low-Level Design (LLD)**
   * **Classes and Responsibilities**
   * **Class Diagram**
   * **Interactions**

**Objective**

Develop a console-based file explorer application in C++ that interfaces with the Linux operating system to manage files and directories.

**Introduction**

The Console-Based File Explorer Application is designed to provide users with a command-line interface for performing various file system operations on a Linux operating system. This application aims to simplify file and directory management through a set of intuitive commands, enabling users to navigate directories, manipulate files, search for content, and manage permissions seamlessly.

The primary objective of this project is to create a robust and efficient tool that leverages the power of the Linux file system, offering a comprehensive set of features typically found in graphical file explorers. By developing this application, users will be able to perform essential file operations directly from the terminal, enhancing productivity and streamlining workflows.

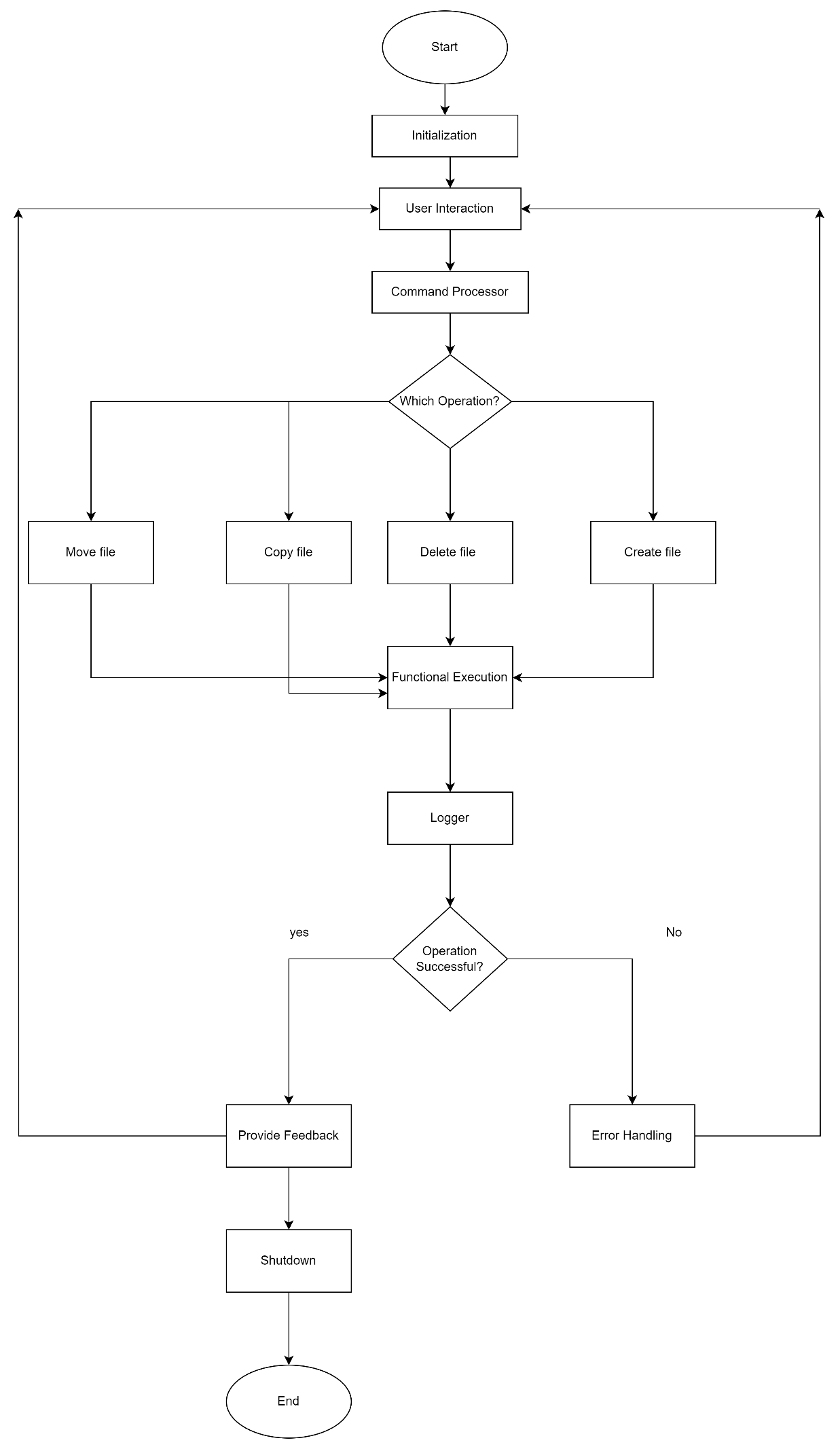
**High-Level Design (HLD)**

* **FileManager:** Responsible for file manipulation tasks.
* **CommandProcessor:** Handles user commands and directs them to the appropriate functions.
* **Logger:** Logs the operations and any errors encountered.

**Flow of Operations**

1. **Initialization:** Set up the environment and display the initial directory.
2. **User Interaction:** Accept user commands.
3. **Command Processor:** Validate and parse commands, then direct them to the appropriate module.
4. **File Operations:**
   * **Create:** Create a new file.
   * **Delete:** Delete an existing file.
   * **Copy:** Copy a file from one location to another.
   * **Move:** Move a file from one location to another.
5. **Logging:** Log significant events and errors.
6. **Feedback:** Provide feedback to the user based on the operation's result.
7. **Error Handling:** Handle any errors encountered during the operation.

**Flow Chart:**

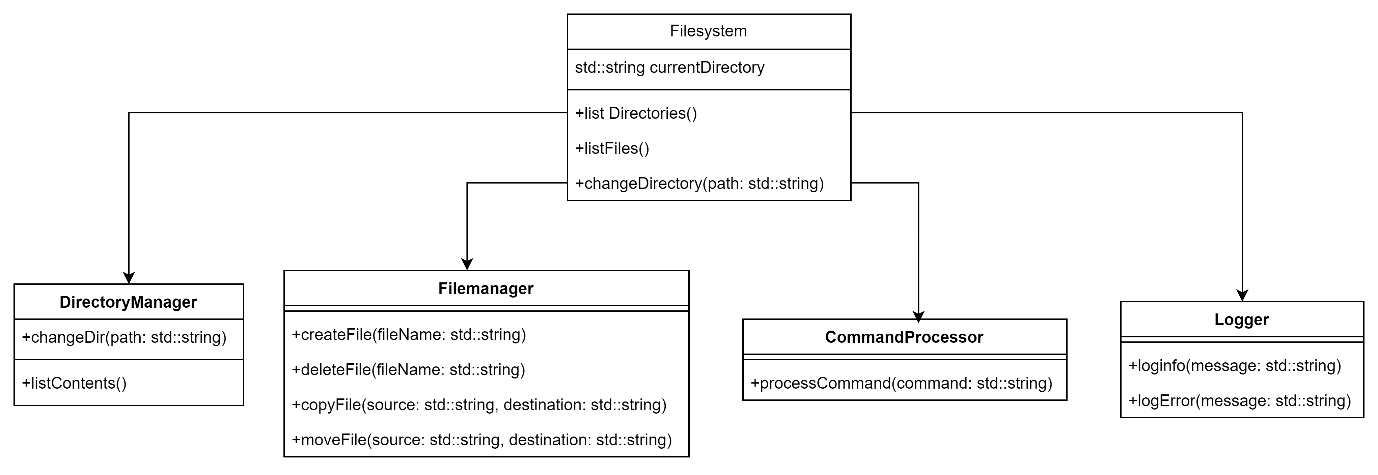
****

**Low-Level Design (LLD)**

**Classes and Responsibilities**

1. **FileSystem**: Manages the overall file system and interactions with other components.
   * **Methods:**
     + **listDirectories():** Lists all directories.
     + **listFiles():** Lists all files in a directory.
     + **changeDirectory(const std::string& path):** Changes the current working directory.
2. **DirectoryManager:** Handles directory-related operations.
   * **Methods:**
     + **changeDir(const std::string& path):** Changes to the specified directory.
     + **listContents():** Lists contents of the current directory.
3. **FileManager:** Manages file operations.
   * **Methods:**
     + **createFile(const std::string& fileName):** Creates a new file.
     + **deleteFile(const std::string& fileName):** Deletes an existing file.
     + **copyFile(const std::string& source, const std::string& destination):** Copies a file.
     + **moveFile(const std::string& source, const std::string& destination):** Moves a file.
4. **SearchManager:** Handles search operations.
   * **Methods:**
     + **searchFiles(const std::string& query):** Searches for files matching the query.
5. **PermissionManager:** Manages file and directory permissions.
   * **Methods:**
     + **setPermissions(const std::string& path, const std::string& permissions):** Sets permissions for a file or directory.
     + **getPermissions(const std::string& path):** Gets the permissions of a file or directory.
6. **CommandProcessor:** Processes user commands and directs them to the appropriate manager.
   * **Methods:**
     + **processCommand(const std::string& command):** Parses and processes the user command.
7. **Logger:** Logs events and errors.
   * **Methods:**
     + **logInfo(const std::string& message):** Logs informational messages.
     + **logError(const std::string& message):** Logs error messages.
8. **Error Handling**: Ensure that each method includes appropriate error handling. For example, methods in FileManager should handle cases where files do not exist or paths are invalid.
9. **Testing**: Develop test cases for each method to ensure they function correctly. This includes unit tests for individual methods and integration tests for interactions between classes.
10. **User Input Validation**: In CommandProcessor, validate user inputs to avoid processing invalid or harmful commands.
11. **Logging**: Use the Logger class effectively to capture important events and errors, which can be crucial for debugging and monitoring the application.

**Class Diagram:**

****

**Interactions**

* **User Command Handling:**
  + User inputs a command.
  + CommandProcessor parses the command and determines which manager to use.
  + The appropriate manager (e.g., DirectoryManager, FileManager) performs the operation.
  + Logger logs the result of the operation.
  + If successful, CommandProcessor provides feedback to the user. If an error occurs, it is handled by ErrorHandling.
* **Directory and File Operations:**
  + FileSystem interacts with DirectoryManager and FileManager to perform file and directory operations.
  + PermissionManager checks and sets permissions as needed.
* **Search Operations:**
  + SearchManager performs search operations and returns results.