**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.

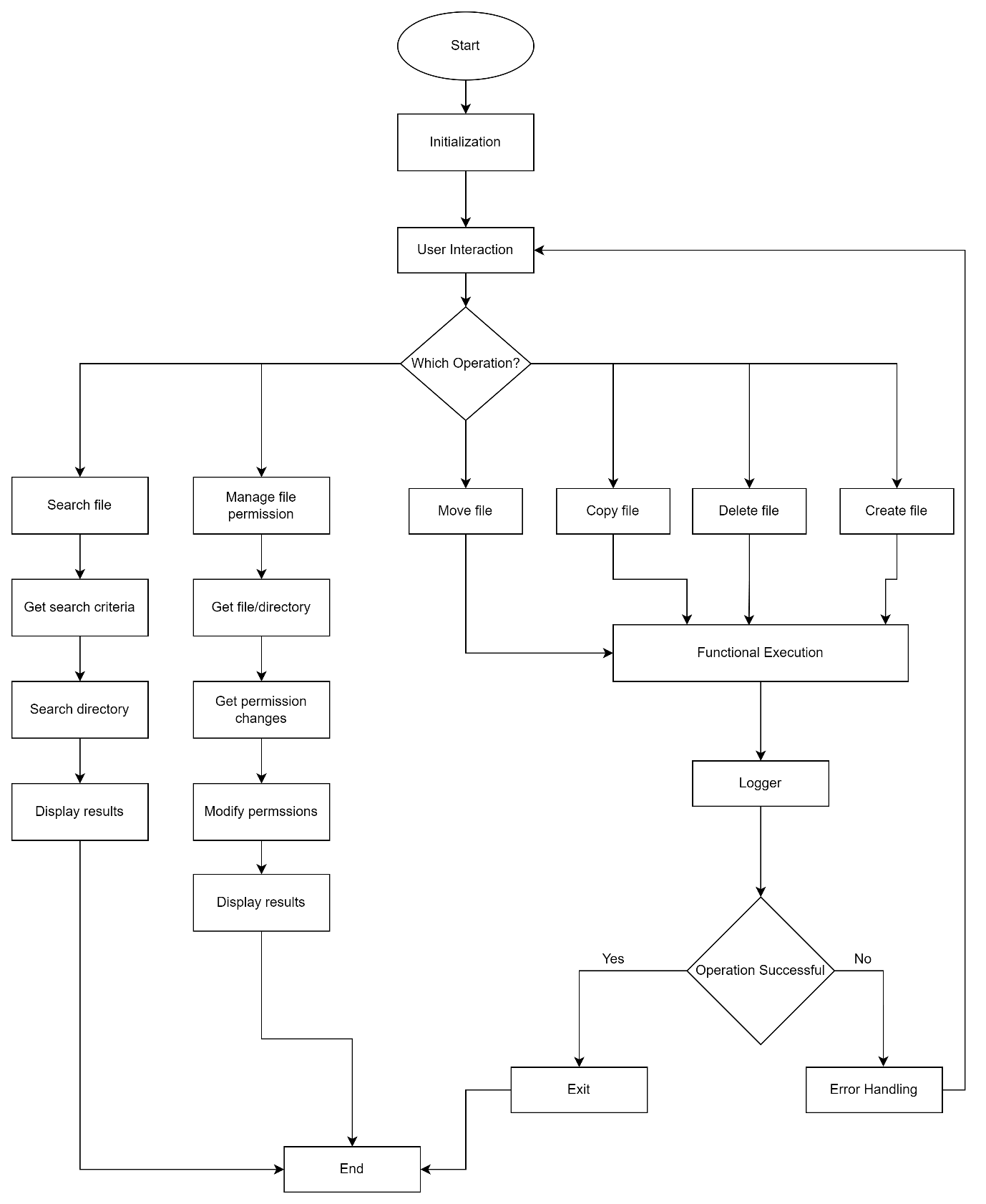
**High-Level Design (HLD)**

* **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:**

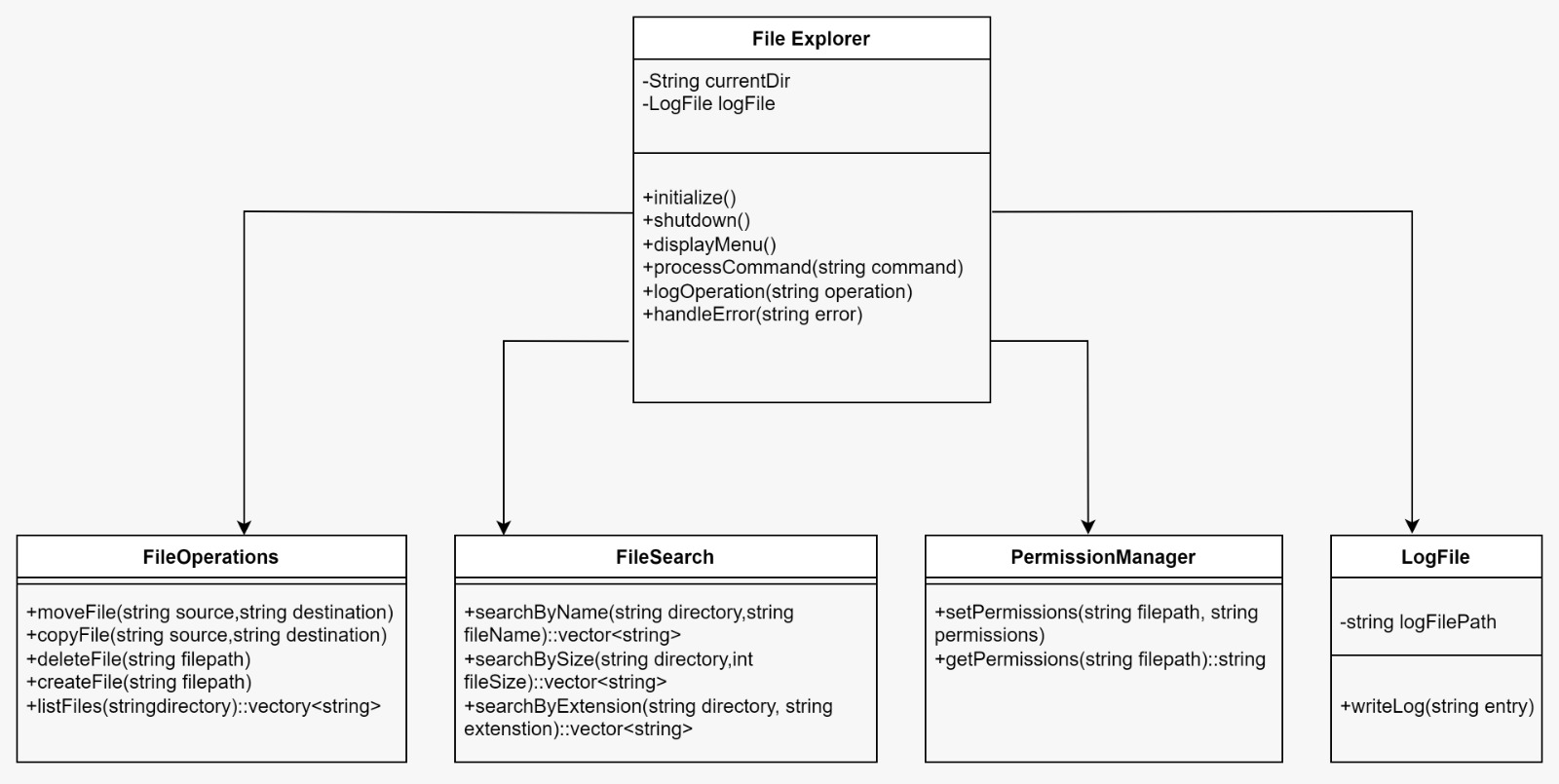
****

**Low-Level Design (LLD)**

**Classes and Responsibilities**

1. **FileSystem**
   * **Responsibility**: Manages the overall file system and interactions with other components.
   * **Methods**:
     + listDirectories(): Lists all directories in the current path.
     + listFiles(): Lists all files in the current directory.
     + changeDirectory(const std::string& path): Changes the current working directory to the specified path.
2. **DirectoryManager**
   * **Responsibility**: Handles directory-related operations.
   * **Methods**:
     + changeDir(const std::string& path): Changes the current directory to the specified path.
     + listContents(): Lists the contents (files and directories) of the current directory.
3. **FileManager**
   * **Responsibility**: Manages file operations such as creation, deletion, copying, and moving.
   * **Methods**:
     + createFile(const std::string& fileName): Creates a new file with the specified name.
     + deleteFile(const std::string& fileName): Deletes the specified file.
     + copyFile(const std::string& source, const std::string& destination): Copies a file from the source path to the destination path.
     + moveFile(const std::string& source, const std::string& destination): Moves a file from the source path to the destination path.
4. **SearchManager**
   * **Responsibility**: Handles search operations within the file system.
   * **Methods**:
     + searchFiles(const std::string& query): Searches for files matching the specified query in the current directory or recursively in subdirectories.
5. **PermissionManager**
   * **Responsibility**: Manages file and directory permissions.
   * **Methods**:
     + setPermissions(const std::string& path, const std::string& permissions): Sets permissions (e.g., read, write, execute) for the specified file or directory.
     + getPermissions(const std::string& path): Retrieves the current permissions of the specified file or directory.
6. **CommandProcessor**
   * **Responsibility**: Processes user commands and directs them to the appropriate manager for execution.
   * **Methods**:
     + processCommand(const std::string& command): Parses the user command and calls the relevant method in FileManager, DirectoryManager, SearchManager, or PermissionManager based on the command.
7. **Logger**
   * **Responsibility**: Logs events, operations, and errors that occur during the execution of the application.
   * **Methods**:
     + logInfo(const std::string& message): Logs informational messages such as successful operations or system states.
     + logError(const std::string& message): Logs error messages that include details of what went wrong.
8. **Error Handling**
   * **Responsibility**: Ensures robust error management within each class.
   * **Details**:
     + Each method in classes like FileManager, DirectoryManager, and SearchManager should include proper error handling mechanisms to deal with cases such as file not found, access denied, invalid paths, etc.
     + Errors should be logged using the Logger class, and appropriate error messages should be returned or displayed to the user.
9. **Testing**
   * **Responsibility**: Validate the correctness and reliability of each class and method.
   * **Details**:
     + **Unit Tests**: Develop unit tests for each method to ensure that it functions correctly in isolation.
     + **Integration Tests**: Develop integration tests to verify that the interactions between classes work as expected and that commands result in the correct operations being executed.
10. **User Input Validation**
    * **Responsibility**: Ensure that user inputs are safe and valid before processing.
    * **Details**:
      + The CommandProcessor should include validation checks for all user inputs to prevent invalid commands or harmful actions from being executed.
      + Examples include verifying file paths, ensuring command syntax is correct, and checking that required arguments are provided.
11. **Logging**
    * **Responsibility**: Capture important events and errors during application runtime.
    * **Details**:
      + The Logger class should be used consistently across all classes to log key operations (e.g., file creation, deletion, permission changes) and errors.
      + This logging is crucial for debugging and monitoring the application's behavior and helps in diagnosing issues during runtime.

**Class Diagram:**



**Interactions**

**User Command Handling:**

* User inputs a command.
  + The user inputs a command in the FileExplorer.
* FileExplorer processes the command.
  + FileExplorer uses the processCommand method to interpret the command.
* Appropriate operation is performed.
  + Depending on the command, FileExplorer calls one of the following:
    - FileOperations for file manipulation (e.g., move, copy, delete).
    - FileSearch for searching files based on name, size, or extension.
    - PermissionManager for setting or getting file permissions.
* Operation is logged.
  + After the operation, FileExplorer calls logOperation to log the action in LogFile.
* Error handling.
  + If an error occurs, FileExplorer calls handleError to manage and log the error using LogFile.

**Directory and File Operations:**

* FileExplorer interacts with FileOperations.
  + FileExplorer calls methods in FileOperations (e.g., moveFile, copyFile, deleteFile) to handle file operations.
* Permission checks.
  + Before certain operations, FileExplorer may interact with PermissionManager to ensure the correct permissions are in place using getPermissions and setPermissions.

**Search Operations:**

* FileExplorer interacts with FileSearch.
  + FileExplorer uses FileSearch to locate files based on different criteria (e.g., name, size, extension) using methods like searchByName, searchBySize, or searchByExtension.
* **Search results are returned.**
  + FileSearch returns the results back to FileExplorer, which then presents them to the user.