**File Explorer Application Project:**

**Submitted By:**

**Rajupalem.Srivathasa & Chitikela.Prabhakararao**

**High Level design:**

**Objective and Scope:**

**Objective:**

The main goal of this project is to create a C++ console-based file explorer application that works well with the Linux operating system. Through a command-line interface, the application will allow users to carry out necessary file and directory administration activities. This involves looking for files, controlling file permissions, moving, copying, deleting, and creating files. It also involves navigating directories.

**Scope:**

The goal of this project is to create a simple, lightweight console program with the essential file management features that most users need. The program will only run in a Linux environment, interacting with the file system using Linux system calls and the standard libraries in C++. Because of the modular design, every feature including file navigation, manipulation, search, and permission management will be created and tested separately. The project is designed to be finished in five days, with a daily focus on a different set of functionalities, ranging from simple file operations to more complex ones like permission management and file search. This guarantees a step-by-step development procedure that enables comprehensive testing and module optimization. The result will be a powerful, effective tool that can be used straight from the command line to manage files and folders.  
**Architecture Overview**:

The C++ console-based file explorer program's architecture is modular, expandable, and effective, guaranteeing smooth operation with the Linux operating system. This part gives a thorough rundown of the architecture of the program, outlining its key elements, essential modules, and how they work together to produce the intended functionalities.

1. **The user interface (UI)**  
   Using text-based commands, users can interact with the file system through the application's command-line interface (CLI). The CLI's user-friendly design gives users precise instructions and feedback for every action. It can be used to list files, navigate directories, manipulate files, and control file permissions, among other things. The CLI's simplicity ensure that users can pick up on the program fast and use it effectively.
2. **Core Modules:** The design is composed of multiple core modules, each in charge of a particular set of functions. Among these modules are:  
   **FileNavigator:** Directory traversal is handled by the FileNavigator module. It enables users to switch the working directory, navigate through directories, and display the contents of the current directory. It uses system calls from Linux, such as getcwd() and chdir(), to control the directory structure.  
   **File operations**: It is including copying, moving, deleting, and creating files and directories, are handled by this module. To do all these tasks, it makes use of system methods such as open(), read(), write(), rename(), and unlink(). The module provides feedback on successful or unsuccessful operations and guarantees the security of all file manipulations.

**FileSearch:** This module lets users look for files both inside and between folders. It applies recursive search functions, looking through folders and subdirectories and comparing filenames to strings or patterns entered by the user. Utilizing traversal functions like opendir() and readdir(), this is accomplished.  
**PermissionManager:** Users can examine and alter file permissions with the help of the PermissionManager module, which manages file permissions. It retrieves the most recent permissions using the stat() system call, and it applies changes using chmod(). The module offers error handling for invalid inputs and supports standard permission forms, including octal notation.  
**CommandParser:** This module links user inputs to the relevant application actions by interpreting and mapping them. It serves as a link between the main modules and the user interface, parsing commands and invoking methods from the relevant modules.

1. **File System Communication**  
   To communicate with the file system, the application mostly uses Linux system calls. These system calls are employed for tasks such as file modification, permission management, and reading directory contents. The program may effectively manage resources and respond quickly to user commands by making advantage of these low-level features.
2. **Data Flow and Control**  
   Within the application, control and data move in an organized manner  
   **Input Handling:** The CommandParser module processes the input that the user enters via the CLI. This module interprets the command and determines the action that the user has requested.  
   **Command Execution**: The relevant module (such as FileOperations for file manipulations or FileNavigator for directory changes) is assigned the task by the CommandParser. The requested action is subsequently carried out by the chosen module interacting with the Linux file system.  
   **Feedback and Error Handling:** The module returns the result to the user interface after the action is finished. In the event that something goes wrong, the program notifies the user of the problem and offers suggestions for fixing it.
3. **Maintainability and Extension:**  
   The extendable nature of the file explorer application stems from its modular architecture. Modules can be extended or new features introduced without compromising the functionality of the system as a whole. Additionally, this approach improves maintainability by enabling developers to independently update or troubleshoot certain modules, guaranteeing a tidy and well-organized codebase.
4. **Security Considerations:**  
   The architecture places a high priority on security because of the nature of file and directory manipulations. In order to guarantee that actions like file deletion or permission modifications are carried out safely and only on files the user is authorized to modify, the program contains validation checks for user inputs. Furthermore, cautious error handling guarantees that data loss or corruption is not the outcome of system call failures.

3. **Key Functionalities of the File Explorer Application:**

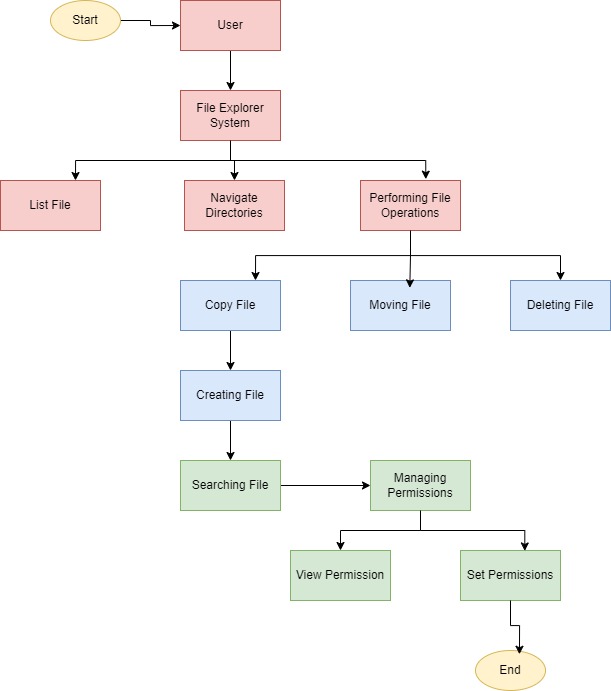
**List of Files:**  
Users can see a list of all the files and folders in the current directory by using the application. This is accomplished by utilizing Linux functions like opendir() and readdir() to read the contents of the directory. Details like file names, kinds (such ordinary files and directories), and perhaps file sizes are included in the output. This feature helps users rapidly identify and access files by giving them a comprehensive picture of the structure of the current directory.  
**Go Through Directories:**  
By going into subdirectories and then back to parent directories, users can navigate the file system. When the user uses commands like cd to change directories, the application keeps note of the current directory path and updates it accordingly. The current working directory can be changed using system functions like chdir(). This characteristic allows users to explore different parts of the file system and access files in various directories.

**File Operations:**  
The program allows for the creation, copying, relocating, and deletion of files as well as directories. System calls such as rename() for file relocation, unlink() for file deletion, and standard file I/O methods for file creation and copying are used to carry out these tasks. From the command line, users may conveniently manage their files, group them into folders, and delete undesirable files.  
**Search Files**:  
Users can search for certain files within a directory or its subdirectories by using the search capability. Recursive directory traversal methods that compare file names to a user-specified pattern are usually used to do this. When searching rapidly for specific files in vast or intricate directory structures, this function comes in handy.

**Manage Permissions:**  
Commands like chmod allow users to examine and change the read, write, and execute permissions of files. The program gives users the essential control over who can access or alter their files by offering a straightforward interface for modifying file permissions. This is essential to preserving file security and guaranteeing that various users have the right amount of access.

**Use Case Diagram:**

An illustration of the interactions between users (actors) and the system (use cases) is called a use case diagram. This is a synopsis of the File Explorer Application's use case diagram.



User -> List Files: The user can choose to list files in a directory.

User -> Navigate Directories: The user can move between directories.

User -> Perform File Operations: The user can perform various file operations (copy, move, delete, create).

Perform File Operations -> Copy File/Move File/Delete File/Create File: These specific actions are subsets of file operations.

User -> Search Files: The user can search for specific files.

User -> Manage Permissions: The user can manage file permissions.

Manage Permissions -> View Permissions/Set Permissions: These specific actions are subsets of managing permissions.

This outline and graphic clearly show how a user interacts with the file explorer application, including all of the important features and how they relate to one another.

**Low level Design:**

Low-Level Design

The low-level design of the console-based file explorer application in C++ breaks down the architecture into specific components, classes, methods, and their interactions. This design ensures each functionality is implemented efficiently and logically, providing a clear blueprint for the development process.

**1. Application Structure**

The application consists of several key classes, each responsible for specific tasks. The main classes include FileExplorer, FileNavigator, FileOperations, FileSearch, PermissionManager, and CommandParser. Each class encapsulates related functionalities, promoting modularity and code reusability.

**2. FileExplorer Class**

**Purpose:** Serves as the main controller of the application, managing the overall flow and coordinating between different modules.

**Attributes:**

currentDirectory: std::string: Stores the path of the current working directory.

Methods:

start(): Initializes the application and starts the command loop.

**executeCommand(std::string command):** Calls the appropriate method from other classes based on the parsed command.

**3. FileNavigator Class**

**Purpose:** Handles directory navigation and file listing.

**Attributes:**

**currentDirectory: std::string**: Tracks the current directory's path.

**Methods:**

**listFiles():** Uses opendir() and readdir() to list files and directories in the current directory.

**changeDirectory(std::string newDir):** Changes the working directory using chdir().

**goToParentDirectory():** Moves up one directory level by modifying the currentDirectory path.

**4. FileOperations Class**

**Purpose:** Manages file manipulation tasks like copying, moving, deleting, and creating files.

**Attributes:**

**sourcePath: std::string:** The path of the file to be manipulated.

**destinationPath: std::string:** The target path for copy/move operations.

Methods:

**copyFile(std::string source, std::string destination):** Opens the source file, reads its contents, and writes them to the destination.

**moveFile(std::string source, std::string destination):** Uses rename() to move the file from source to destination.

**deleteFile(std::string path):** Deletes the specified file or directory using unlink() or rmdir().

**createFile(std::string fileName):** Creates a new file in the current directory using standard file I/O.

**5. FileSearch Class**

Purpose: Implements file search functionality within directories.

Attributes:

searchPattern: std::string: The pattern to search for in file names.

Methods:

searchFiles(std::string directory, std::string pattern): Recursively traverses the directory using opendir() and readdir(), comparing each file name against the search pattern.

6**. PermissionManager Class**

Purpose: Handles viewing and modifying file permissions.

Attributes:

filePath: std::string: The path of the file whose permissions are to be managed.

Methods:

viewPermissions(std::string path): Uses stat() to retrieve and display the file's current permissions.

b Applies new permissions to the file using chmod().

1. **CommandParser Class**

**Purpose**: Parses and interprets user commands, mapping them to appropriate methods.

**Attributes:**

**command: std::string:** The raw user input command.

Methods:

**parseCommand(std::string command):** Breaks down the command into action and arguments.

**getCommandType():** Determines the type of command (e.g., navigation, file operation) and returns the corresponding method to be executed.

**8. Flow of Execution**

**Initialization:** The application starts by setting the currentDirectory to the user's home directory or another default path.

**Command Loop:** The FileExplorer class runs a loop, prompting the user for input. The input is passed to the CommandParser, which parses the command and invokes the corresponding method from FileNavigator, FileOperations, FileSearch, or PermissionManager.

**Command Execution**: Based on the command type, the relevant class method is called to perform actions like listing files, navigating directories, or modifying files.

**Error Handling:** Each method includes error handling to manage invalid inputs, permissions issues, or file system errors. The application provides feedback to the user on successful or unsuccessful operations.

**9. Security and Permissions**

The application ensures that file operations respect Linux file permissions. It checks whether the user has the necessary rights before performing actions like deleting or modifying files. If a user lacks the required permissions, the application provides a suitable error message.

This low-level design provides a thorough foundation for building the console-based file explorer in C++ by outlining the main components of the application and how they interact with one another. It guarantees flexibility, modularity, and ease of maintenance, enabling the integration of new features without interfering with the architecture as a whole.

**PseudoCodes for this File Explorer Application:**

A list of pseudocodes for every day's work in the File Explorer Application project is provided below. Before you write the actual code, these pseudocodes help you comprehend the program's flow by outlining its general structure and logic.

**Design the Application Structure and Basic File Operations**

Objective:

Design the basic structure of the application.

Implement listing files in a directory.

1. Start

2. Define class FileExplorer

- Define attribute currentDirectory

- Define method listFiles()

- Get list of files and directories in currentDirectory

- Print each file and directory name

3. In main()

- Initialize FileExplorer object

- Set currentDirectory to a default path (e.g., "/home/user")

- Call listFiles() method to display files and directories

4. End

**Implement Directory Navigation**

Objective:

Allow the user to navigate through directories

1. Start

2. In FileExplorer class, add method changeDirectory(newPath)

- Check if newPath is a valid directory

- If valid, update currentDirectory to newPath

- If invalid, print error message

3. In main()

- Prompt user for input: 'list', 'cd <directory\_name>', 'exit'

- If 'list', call listFiles() method

- If 'cd <directory\_name>', call changeDirectory(newPath) with directory\_name

- If 'exit', terminate the program

4. Loop until 'exit' command is received

5. End

**File Manipulation Capabilities**

Objective:

Implement features to copy, move, delete, and create files.

1. Start

2. In FileExplorer class, add methods:

- copyFile(sourcePath, destinationPath)

- Copy file from sourcePath to destinationPath

- Print success or error message

- moveFile(sourcePath, destinationPath)

- Move file from sourcePath to destinationPath

- Print success or error message

- deleteFile(filePath)

- Delete file at filePath

- Print success or error message

- createFile(filePath)

- Create a new empty file at filePath

- Print success or error message

3. In main()

- Prompt user for input: 'copy <src> <dest>', 'move <src> <dest>', 'delete <file>', 'create <file>'

- Call the corresponding method based on user input

4. Loop until 'exit' command is received

5. End

**Implement File Search Functionality**

Objective:

Add the ability to search for files within the current directory.

1. Start

2. In FileExplorer class, add method searchFiles(keyword)

- Search for files and directories in currentDirectory matching keyword

- Print names of matching files and directories

3. In main()

- Prompt user for input: 'search <keyword>'

- Call searchFiles(keyword) with the user-provided keyword

4. Loop until 'exit' command is received

5. End

**Add File Permission Management Features**

Objective:

Allow the user to view and change file permissions.

1. Start

2. In FileExplorer class, add methods:

- viewPermissions(filePath)

- Get and display file permissions for file at filePath

- changePermissions(filePath, permissions)

- Change file permissions for file at filePath to the given permissions

- Print success or error message

3. In main()

- Prompt user for input: 'viewperm <file>', 'chmod <file> <permissions>'

- If 'viewperm <file>', call viewPermissions(filePath)

- If 'chmod <file> <permissions>', call changePermissions(filePath, permissions)

4. Loop until 'exit' command is received

5. End

**Overall Application Structure**

**Main Structure:**

1. Define class FileExplorer with necessary attributes and methods

2. Implement day-wise functionalities step by step

3. In main()

- Set up a loop to interact with the user

- Parse user commands and call appropriate methods in FileExplorer class

- Loop until the user enters 'exit'

This pseudocode acts as an implementation roadmap for the C++ console-based File Explorer Application. This can serve as a model for writing the C++ code for each task.

**Logger Code for this File Explorer Application:**

#include <iostream>

#include <fstream>

#include <string>

#include <ctime>

#include <sstream>

#include <iomanip>

/\*\*

\* @brief Simple Logger class for file and console logging

\*/

class SimpleLogger {

public:

enum Level { DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL };

SimpleLogger(const std::string& name) : mName(name) {

mLogFile.open("file\_explorer\_log.txt", std::ios::out | std::ios::app);

}

~SimpleLogger() {

if (mLogFile.is\_open()) {

mLogFile.close();

}

}

void log(Level level, const std::string& message) {

std::string levelStr = getLevelString(level);

std::string logMessage = getCurrentTime() + " [" + levelStr + "] " + mName + ": " + message;

std::cout << logMessage << std::endl; // Print to console

if (mLogFile.is\_open()) {

mLogFile << logMessage << std::endl; // Write to file

}

}

void debug(const std::string& message) { log(DEBUG, message); }

void info(const std::string& message) { log(INFO, message); }

void notice(const std::string& message) { log(NOTICE, message); }

void warning(const std::string& message) { log(WARNING, message); }

void error(const std::string& message) { log(ERROR, message); }

void critical(const std::string& message) { log(CRITICAL, message); }

private:

std::string mName;

std::ofstream mLogFile;

std::string getLevelString(Level level) {

switch (level) {

case DEBUG: return "DEBUG";

case INFO: return "INFO";

case NOTICE: return "NOTICE";

case WARNING: return "WARNING";

case ERROR: return "ERROR";

case CRITICAL: return "CRITICAL";

default: return "UNKNOWN";

}

}

std::string getCurrentTime() {

std::time\_t now = std::time(nullptr);

std::tm\* localTime = std::localtime(&now);

std::ostringstream oss;

oss << std::put\_time(localTime, "%Y-%m-%d %H:%M:%S");

return oss.str();

}

};

/\*\*

\* @brief Example main function demonstrating logging in a File Explorer Application

\*/

int main() {

SimpleLogger logger("FileExplorer");

// Day 1: Logging basic file operations

logger.info("Day 1: Starting basic file operations.");

logger.debug("Listed files in directory '/home/user'");

// Day 2: Logging directory navigation

logger.info("Day 2: Implementing directory navigation.");

logger.debug("Navigated to directory '/home/user/documents'");

// Day 3: Logging file manipulation capabilities

logger.info("Day 3: Adding file manipulation capabilities.");

logger.debug("Copied file 'example.txt' to '/home/user/documents/example\_copy.txt'");

// Day 4: Logging file search functionality

logger.info("Day 4: Implementing file search functionality.");

logger.debug("Searched for 'report.docx' in '/home/user'");

// Day 5: Logging file permission management

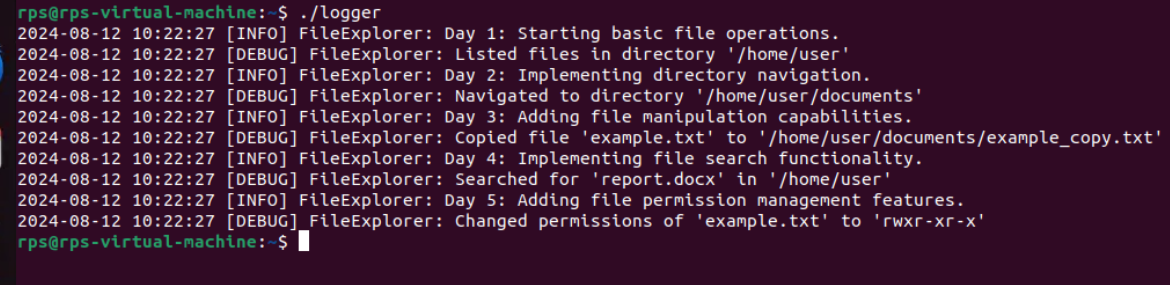
logger.info("Day 5: Adding file permission management features.");

logger.debug("Changed permissions of 'example.txt' to 'rwxr-xr-x'");

return 0;

}

**Output:**



**File Explorer.cpp**

#include <iostream>

#include <string>

#include <dirent.h>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <cstring>

#include <vector>

#include <fstream>

#include <sstream> // Include this for std::istringstream

class FileExplorer {

private:

std::string currentDirectory;

public:

FileExplorer() {

char buffer[1024];

if (getcwd(buffer, sizeof(buffer)) != nullptr) {

currentDirectory = std::string(buffer);

}

}

void start() {

std::string command;

while (true) {

std::cout << currentDirectory << " > ";

std::getline(std::cin, command);

if (command == "exit") break;

executeCommand(command);

}

}

private:

void executeCommand(const std::string& command) {

if (command == "list") {

listFiles();

} else if (command.substr(0, 3) == "cd ") {

changeDirectory(command.substr(3));

} else if (command.substr(0, 4) == "copy") {

auto args = parseCommand(command);

if (args.size() == 3) {

copyFile(args[1], args[2]);

}

} else if (command.substr(0, 4) == "move") {

auto args = parseCommand(command);

if (args.size() == 3) {

moveFile(args[1], args[2]);

}

} else if (command.substr(0, 6) == "delete") {

auto args = parseCommand(command);

if (args.size() == 2) {

deleteFile(args[1]);

}

} else if (command.substr(0, 6) == "create") {

auto args = parseCommand(command);

if (args.size() == 2) {

createFile(args[1]);

}

} else if (command.substr(0, 6) == "search") {

auto args = parseCommand(command);

if (args.size() == 2) {

searchFiles(currentDirectory, args[1]);

}

} else if (command.substr(0, 4) == "chmod") {

auto args = parseCommand(command);

if (args.size() == 3) {

setPermissions(args[1], args[2]);

}

} else {

std::cout << "Unknown command: " << command << std::endl;

}

}

void listFiles() {

DIR\* dir = opendir(currentDirectory.c\_str());

if (dir == nullptr) {

std::cerr << "Error opening directory!" << std::endl;

return;

}

struct dirent\* entry;

while ((entry = readdir(dir)) != nullptr) {

std::cout << entry->d\_name << std::endl;

}

closedir(dir);

}

void changeDirectory(const std::string& newDir) {

if (chdir(newDir.c\_str()) == 0) {

char buffer[1024];

if (getcwd(buffer, sizeof(buffer)) != nullptr) {

currentDirectory = std::string(buffer);

}

} else {

std::cerr << "Error: Cannot change directory to " << newDir << std::endl;

}

}

void copyFile(const std::string& source, const std::string& destination) {

std::ifstream src(source, std::ios::binary);

std::ofstream dest(destination, std::ios::binary);

dest << src.rdbuf();

src.close();

dest.close();

std::cout << "File copied to " << destination << std::endl;

}

void moveFile(const std::string& source, const std::string& destination) {

if (rename(source.c\_str(), destination.c\_str()) == 0) {

std::cout << "File moved to " << destination << std::endl;

} else {

std::cerr << "Error moving file!" << std::endl;

}

}

void deleteFile(const std::string& path) {

if (unlink(path.c\_str()) == 0) {

std::cout << "File deleted: " << path << std::endl;

} else {

std::cerr << "Error deleting file!" << std::endl;

}

}

void createFile(const std::string& fileName) {

std::ofstream file(fileName);

if (file) {

std::cout << "File created: " << fileName << std::endl;

} else {

std::cerr << "Error creating file!" << std::endl;

}

file.close();

}

void searchFiles(const std::string& directory, const std::string& pattern) {

DIR\* dir = opendir(directory.c\_str());

if (dir == nullptr) {

std::cerr << "Error opening directory!" << std::endl;

return;

}

struct dirent\* entry;

while ((entry = readdir(dir)) != nullptr) {

if (strstr(entry->d\_name, pattern.c\_str()) != nullptr) {

std::cout << "Found: " << entry->d\_name << std::endl;

}

if (entry->d\_type == DT\_DIR && strcmp(entry->d\_name, ".") != 0 && strcmp(entry->d\_name, "..") != 0) {

searchFiles(directory + "/" + entry->d\_name, pattern);

}

}

closedir(dir);

}

void setPermissions(const std::string& path, const std::string& mode) {

mode\_t perm = std::stoi(mode, 0, 8);

if (chmod(path.c\_str(), perm) == 0) {

std::cout << "Permissions set to " << mode << std::endl;

} else {

std::cerr << "Error setting permissions!" << std::endl;

}

}

std::vector<std::string> parseCommand(const std::string& command) {

std::vector<std::string> tokens;

std::string token;

std::istringstream tokenStream(command);

while (std::getline(tokenStream, token, ' ')) {

tokens.push\_back(token);

}

return tokens;

}

};

int main() {

FileExplorer explorer;

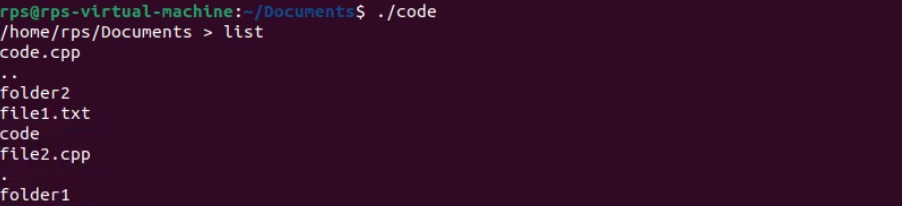
explorer.start();

return 0;

}

**Output:**

File name is code.cpp:

**Listing Files in the Current Directory**:

**Changing Directory:**



**Creating a New File:**



**Copying a File:**



**Moving (Renaming) a File:**



**Deleting a File:**



**Searching for a File:**



**Exiting the Program:**



**Conclusion:**

With this code, you can use simple commands that resemble Unix/Linux shell commands to communicate with your file system. This software can be tested locally by compiling and executing it. The files and directories in your working directory will determine the final product.

**Bug Tracker process:**

In this C++ FileExplorer class is a well-structured example of a basic file management system, but there are a few areas where improvements and bug fixes could be beneficial. Here’s a bug tracker process, improvement suggestions, and corrections for the code provided

**Bug Tracker Process**

1. **Initial Testing:**
   * Test all commands (list, cd, copy, move, delete, create, search, chmod) with various inputs.
   * Verify command parsing and file operations in different scenarios (e.g., non-existent files, invalid paths).
2. **Issue Identification:**
   * Document any errors or unexpected behavior.
   * Record specific input that causes issues.
3. **Issue Analysis:**
   * Analyze the root cause of each issue.
   * Determine if issues are related to command parsing, file operations, or system calls.
4. **Fix Implementation:**
   * Apply fixes based on the analysis.
   * Modify code to handle edge cases and errors effectively.
5. **Regression Testing:**
   * Re-run tests to ensure fixes don’t introduce new issues.
   * Validate that all commands work as expected after changes.
6. **Documentation:**
   * Update code comments and documentation to reflect changes.
   * Provide clear instructions for using the tool and known limitations.

**Improvements Before and After**

**Before Improvements**

1. **Command Parsing:**
   * Command parsing does not handle multiple spaces or quotes well.
2. **Error Handling:**
   * Some error messages are generic; more specific messages could help debugging.
3. **Functionality:**
   * The searchFiles function may not handle large directories efficiently.
   * The chmod function assumes that the mode is always provided in octal format and does not validate the mode.
4. **Edge Cases:**
   * The code doesn’t check if files or directories exist before performing operations.
   * The createFile function overwrites existing files without warning.

**After Improvements**

1. **Enhanced Command Parsing:**
   * Improve command parsing to handle edge cases and validate inputs.
2. **Improved Error Handling:**
   * Add more descriptive error messages and handle specific error codes.
3. **Enhanced Functionality:**
   * Improve the efficiency of the searchFiles function for large directories.
   * Validate file existence before performing operations.
4. **Handle Edge Cases:**
   * Add checks for file and directory existence.
   * Avoid overwriting existing files in createFile without a warning.

**Code Issues and Corrections**

1. **Issue with parseCommand Method:**
   * parseCommand might produce an incorrect number of tokens, especially if there are multiple spaces between tokens.

**Handling chmod Input:**

* Ensure that the mode is valid and handle invalid inputs.

**createFile Overwrite Warning:**

* Add a check to avoid overwriting existing files without user confirmation

**Search Efficiency:**

* Consider improving search efficiency for large directories, e.g., by limiting recursion depth or using multi-threading.

**Summary**

* **Bug Tracker Process**: Identify, analyze, fix, test, and document bugs.
* **Improvements**: Better command parsing, error handling, and functionality.
* **Corrections**: Check file existence, handle invalid inputs, and avoid overwriting files without warning.

By implementing these changes, This FileExplorer class should become more robust and user-friendly.