# Filesystem Management System

Design, Implementation, and Testing

Student Number: M00891519

Name: Ankit KC

# Introduction

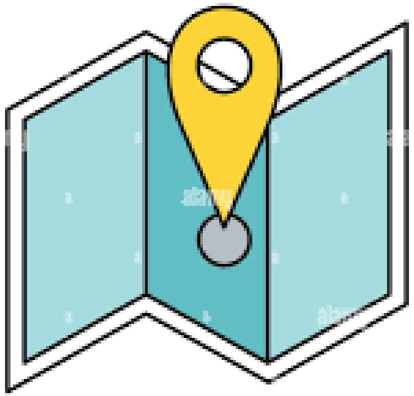
"Development of a simple filesystem management system with functionalities for file and directory operations.“

"This presentation covers the design, implementation, testing, and evaluation of the filesystem management system."

# Project Overview : Goals

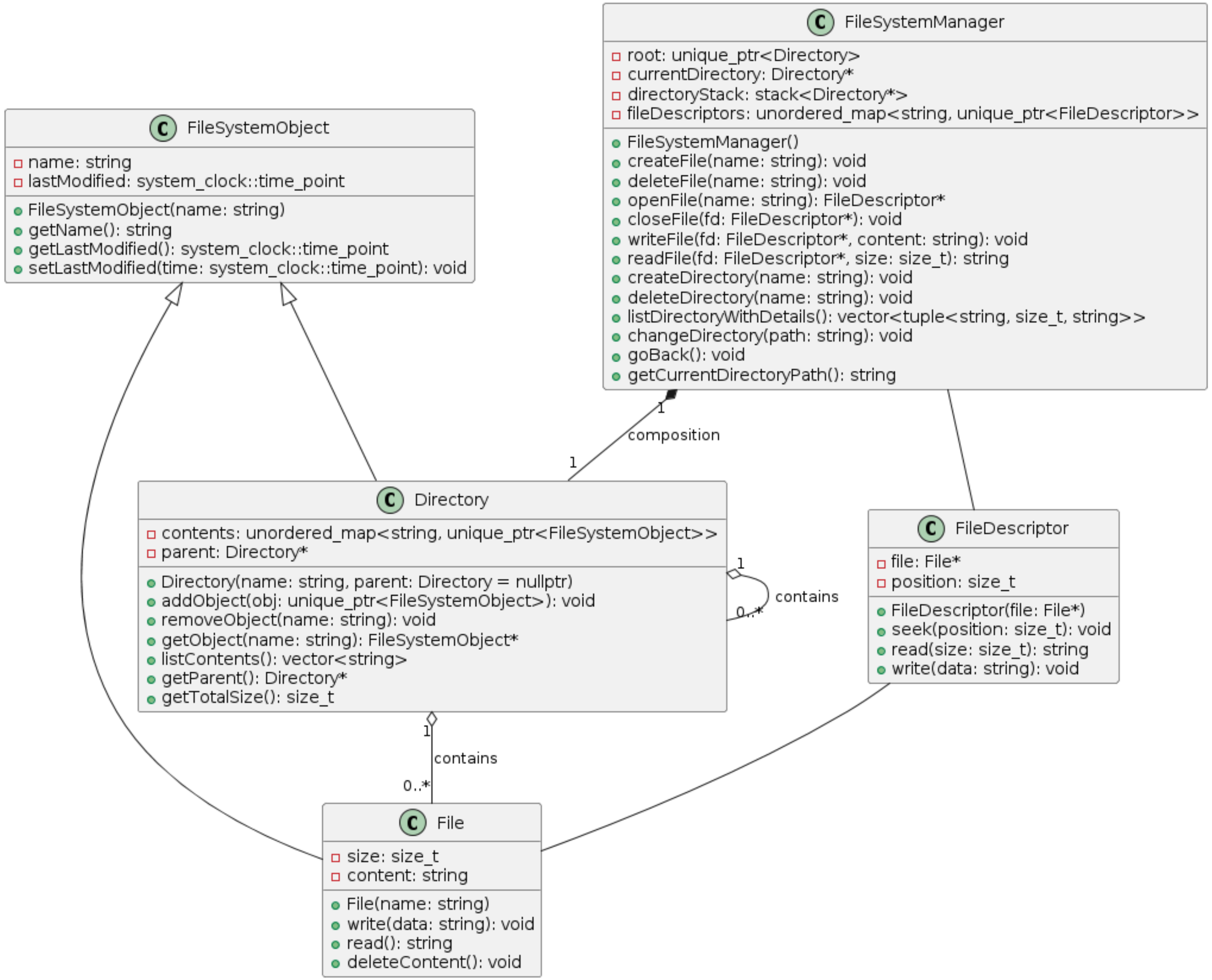
Implementing core filesystem functionalities including file and directory creation, deletion, reading, and writing.

Command-line interface, file handling with content writing/reading, directory navigation.



shutterstock.com · 2380564157

# Class Diagram



FileSystemObject
- name: string
- lastModified: system_clock::time_point
+ FileSystemObject(name: string)
+ getName(): string
+ getLastModified(): system_clock::time_point
+ setLastModified(time: system_clock::time_point): void

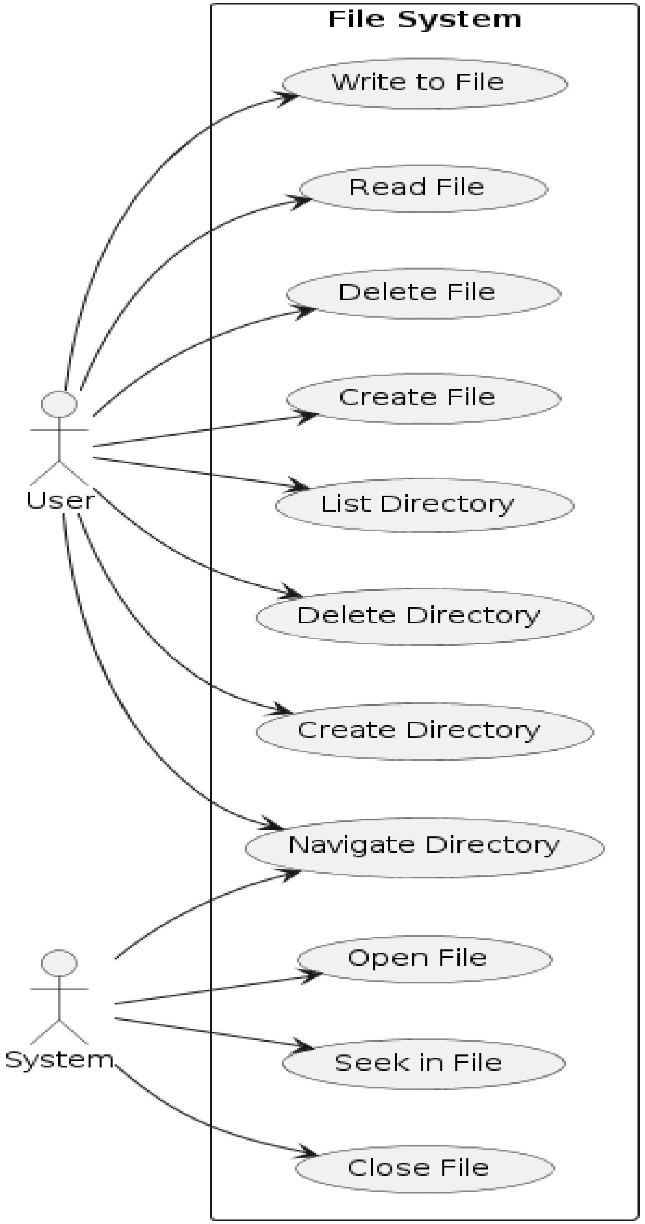
FileSystemManager
- root: unique_ptr<Directory>
- currentDirectory: Directory*
- directoryStack: stack<Directory*>
- fileDescriptors: unordered_map<string, unique_ptr<FileDescriptor>>
+ FileSystemManager()
+ createFile(name: string): void
+ deleteFile(name: string): void
+ openFile(name: string): FileDescriptor*
+ closeFile(fd: FileDescriptor*): void
+ writeFile(fd: FileDescriptor*, content: string): void
+ readFile(fd: FileDescriptor*, size: size_t): string
+ createDirectory(name: string): void
+ deleteDirectory(name: string): void
+ listDirectoryWithDetails(): vector<tuple<string, size_t, string>>
+ changeDirectory(path: string): void
+ goBack(): void
+ getCurrentDirectoryPath(): string

Directory
- contents: unordered_map<string, unique_ptr<FileSystemObject>>
- parent: Directory*
+ Directory(name: string, parent: Directory = nullptr)
+ addObject(obj: unique_ptr<FileSystemObject>): void
+ removeObject(name: string): void
+ getObject(name: string): FileSystemObject*
+ listContents(): vector<string>
+ getParent(): Directory*
+ getTotalSize(): size_t

FileDescriptor
- file: File*
- position: size_t
+ FileDescriptor(file: File*)
+ seek(position: size_t): void
+ read(size: size_t): string
+ write(data: string): void

File
- size: size_t
- content: string
+ File(name: string)
+ write(data: string): void
+ read(): string
+ deleteContent(): void

# Use Case Diagram



# Key Features

FILE **CREATION** AND **DELETION**

DIRECTORY **MANAGEMENT**

DIRECTORY **NAVIGATION**

FILE **READING** AND **WRITING**

# Implementation Approach

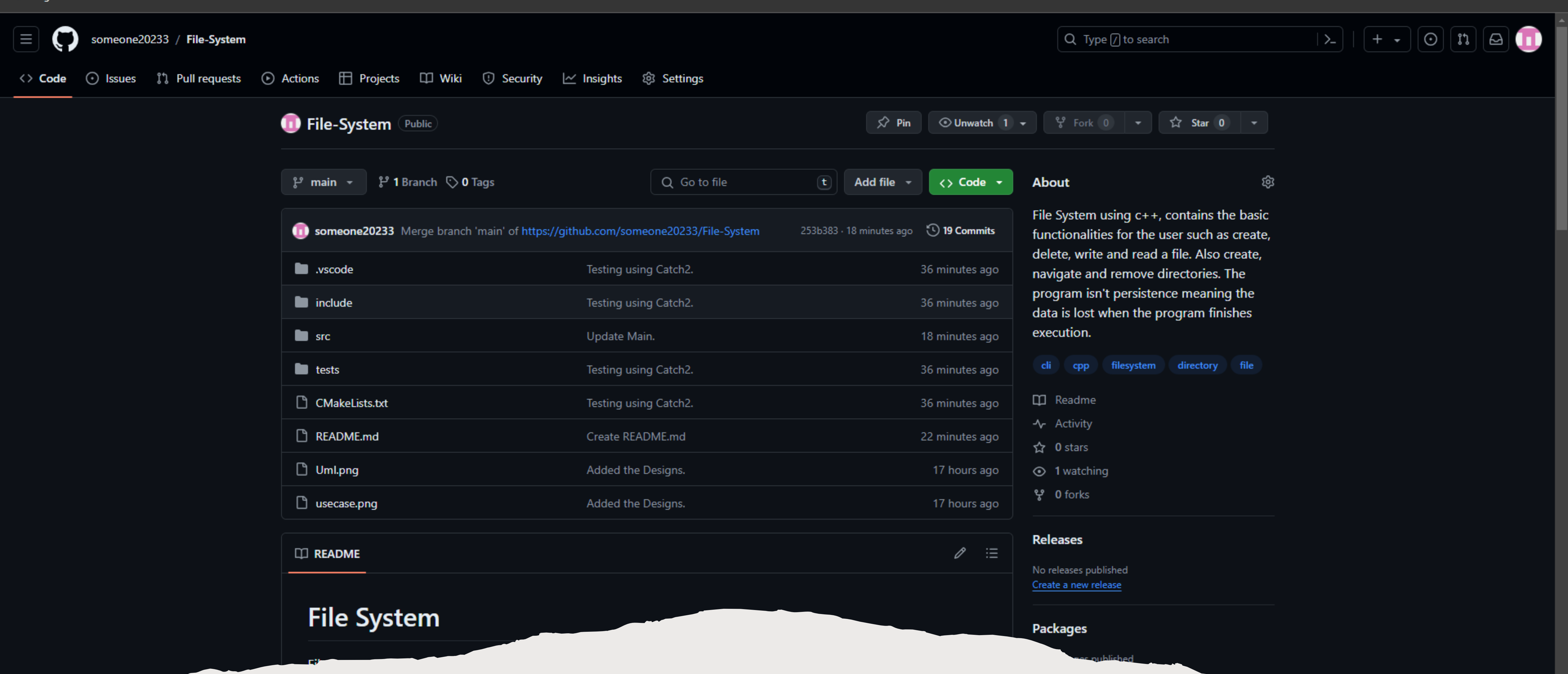
Overview of the implementation approach:

- Developed the filesystem management system in C++ using object-oriented programming principles.

- Object-oriented design (OOD) was chosen for its ability to encapsulate data and behavior within classes, promoting modularity and reusability.

- Leveraged C++ features such as classes, and inheritance to model filesystem objects like directories and files.

- Separation of concerns was maintained through modular design, with distinct classes for different filesystem entities (e.g., FileSystemManager, Directory, File, FileDescriptor).



someone20233 Merge branch 'main' of https://github.com/someone20233/File-System 253b383 · 18 minutes ago 19 Commits

| | | |
|---|---|---|
| .vscode | Testing using Catch2. | 36 minutes ago |
| include | Testing using Catch2. | 36 minutes ago |
| src | Update Main. | 18 minutes ago |
| tests | Testing using Catch2. | 36 minutes ago |
| CMakeLists.txt | Testing using Catch2. | 36 minutes ago |
| README.md | Create README.md | 22 minutes ago |
| Uml.png | Added the Designs. | 17 hours ago |
| usecase.png | Added the Designs. | 17 hours ago |

**About**

File System using c++, contains the basic functionalities for the user such as create, delete, write and read a file. Also create, navigate and remove directories. The program isn't persistence meaning the data is lost when the program finishes execution.

cli cpp filesystem directory file

Readme

Activity

0 stars

1 watching

0 forks

No releases published

Create a new release

File System

# Make File & Version Control

# Makefile & Version Control

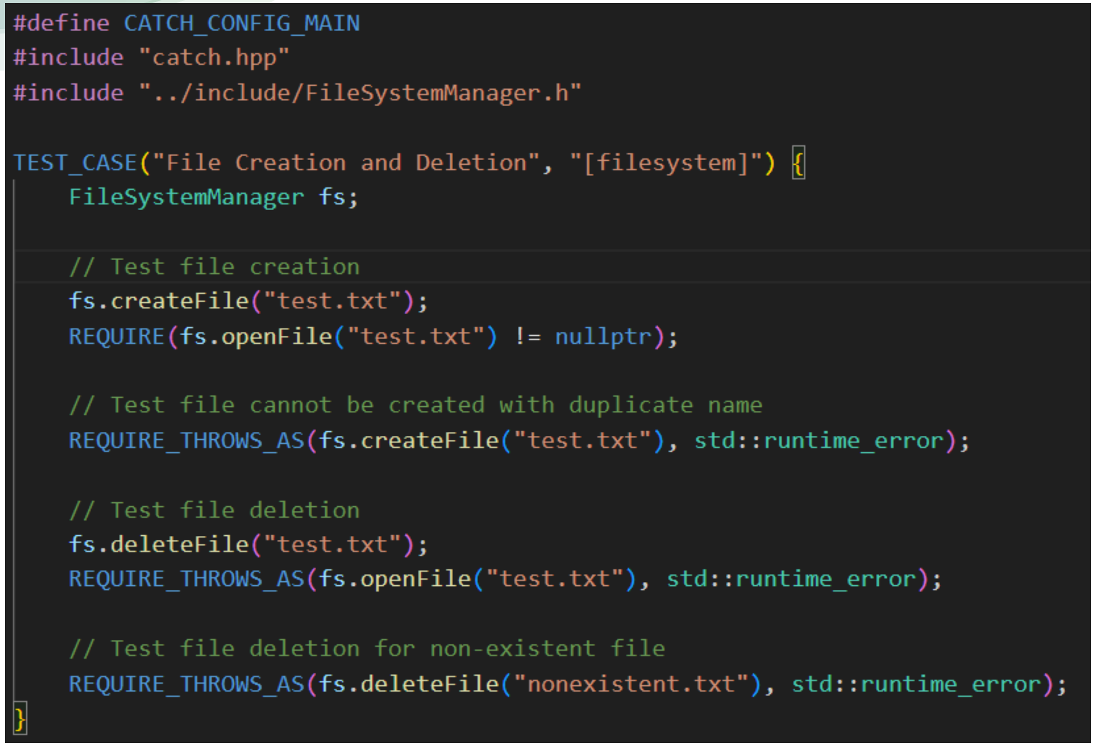
## Description of the Makefile:

- The Makefile served as a build automation tool for compiling and linking the filesystem management system.
- Defined compilation rules, dependencies, and build targets to streamline the build process across different platforms.
- Managed complex build scenarios, ensuring that only modified source files were recompiled to optimize build times
- Included targets for cleaning up object files (clean target) and rebuilding the entire project (all target) to maintain project cleanliness and integrity.

## Description of version control with Git:

- Git was used for version control to track changes made to the codebase over time.
- Maintained a centralized repository on platforms like GitHub or GitLab for remote access and backup of the codebase.

# Testing Approach



```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "../include/FileSystemManager.h"

TEST_CASE("File Creation and Deletion", "[filesystem]") {
    FileSystemManager fs;

    // Test file creation
    fs.createFile("test.txt");
    REQUIRE(fs.openFile("test.txt") != nullptr);

    // Test file cannot be created with duplicate name
    REQUIRE_THROWS_AS(fs.createFile("test.txt"), std::runtime_error);

    // Test file deletion
    fs.deleteFile("test.txt");
    REQUIRE_THROWS_AS(fs.openFile("test.txt"), std::runtime_error);

    // Test file deletion for non-existent file
    REQUIRE_THROWS_AS(fs.deleteFile("nonexistent.txt"), std::runtime_error);
}
```

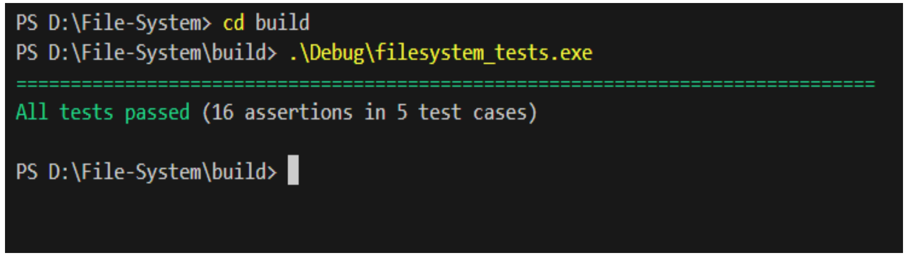
**Unit Testing:** Implemented unit tests using Catch2, a C++ testing framework, to validate individual components like FileSystemManager, Directory, File, and FileDescriptor.

**Coverage:** Ensured comprehensive test coverage by testing various functionalities, including file creation, deletion, reading, writing, directory management, and navigation.

**Edge Cases:** Designed tests to handle edge cases such as empty files, non-existent files or directories, maximum file size limits, and nested directory structures.

**Automation:** Integrated tests into the build process using CMake and Makefile, enabling automated execution of tests to verify code functionality across different environments.

# Testing Result



```
PS D:\File-System> cd build
PS D:\File-System\build> .\Debug\filesystem_tests.exe
===============================================================================
All tests passed (16 assertions in 5 test cases)

PS D:\File-System\build> 
```

**Test Suites:** Created multiple test cases organized into logical test suites to systematically verify different aspects of the filesystem management system.

**Assertions:** Used Catch2 assertions (REQUIRE, REQUIRE_FALSE, etc.) to validate expected behaviors and outcomes, ensuring robustness and reliability of the codebase.

**Test Execution:** Executed tests regularly during development to catch regressions and ensure new code did not break existing functionalities.

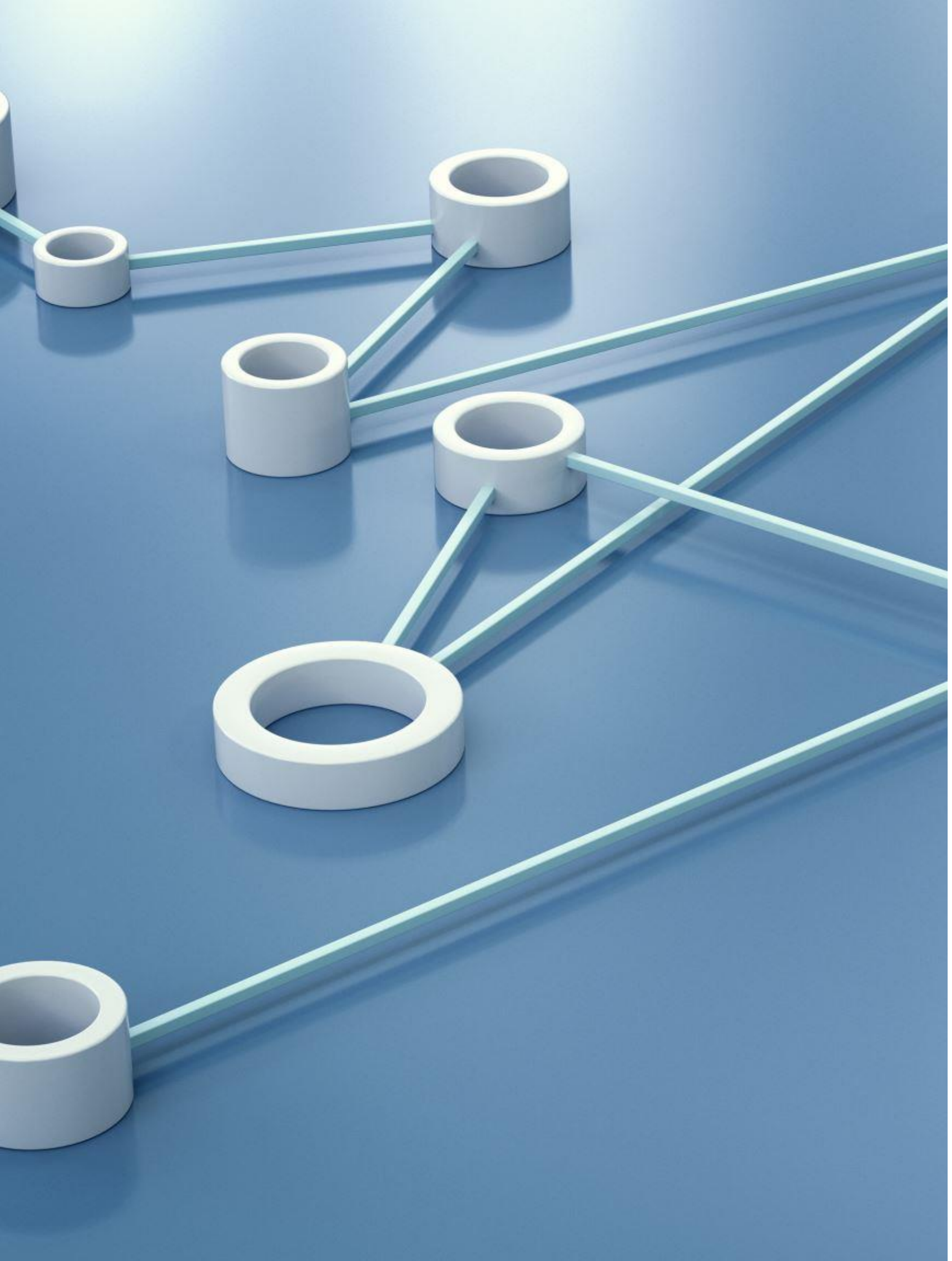
**Outcome**: All critical functionalities passed tests successfully, demonstrating the system's correctness and adherence to specifications.

# Challenges Faced

**MEMORY MANAGEMENT:** ENSURING PROPER MEMORY MANAGEMENT, ESPECIALLY WITH THE USE OF SMART POINTERS (UNIQUE_PTR) REQUIRED CAREFUL ATTENTION TO AVOID MEMORY LEAKS AND DANGLING POINTERS.

**TESTING ACROSS PLATFORMS**: ENSURING CONSISTENT BEHAVIOR AND PERFORMANCE ACROSS DIFFERENT PLATFORMS (WINDOWS AND UNIX-BASED SYSTEMS) DURING TESTING AND DEPLOYMENT.

**EXCEPTION HANDLING:** DESIGNING ROBUST EXCEPTION HANDLING MECHANISMS TO HANDLE UNEXPECTED ERRORS AND EDGE CASES GRACEFULLY WITHOUT COMPROMISING SYSTEM STABILITY.



# Conclusion

Summary of Work Done:

- Developed a robust filesystem management system in C++ using object-oriented design principles.
- Achievements: Successfully implemented core functionalities such as file creation, deletion, reading, writing, directory management, and navigation.
- Reflections: Recognized the importance of modular design, thorough testing, and effective use of version control in maintaining code quality and project organization.

Limitations:

- Acknowledged limitations in current implementation, such as lack of advanced features like file permissions (r, w), extensive error handling, and concurrency support.

Future Directions:

- Proposed improvements for future iterations, including adding support for advanced features, enhancing performance optimizations, and integrating concurrency for multi-threaded operations.

# References

https://github.com/catchorg/Catch2

Thank You!