CST-2550

07/07/2024
RESET COURSEWORK

NIKUNJ LIMBU

M00879919

INTRODUCTION

The purpose of this report is to detail the design and implementation of a filesystem created using C++. This filesystem includes functionalities such as creating, reading, writing, and deleting files and directories. The implementation revolves around several key classes: File, Filesystem, FileDescription, and Directory. Each class encapsulates specific responsibilities to ensure the integrity and functionality of the filesystem.

1 Class Overview

File: Represents a file within the filesystem. It contains attributes such as name, size, content, and permissions. The File class provides methods to read from and write to the file, as well as to modify its attributes.

Directory: Represents a directory within the filesystem. It contains a list of files and subdirectories. The Directory class provides methods to add, remove, and list contents (files and subdirectories).

Filesystem: Acts as the main interface to interact with the filesystem. It manages the overall structure of directories and files. The Filesystem class provides methods to create, delete, open, and close files, as well as navigate through directories.

FileDescription: Represents a descriptor for a file, providing metadata and handling file operations. It maintains information such as file status (open or closed) and current read/write position.

3. Functionality

File Operations:

Create File: Allows users to create a new file within a specified directory.
Read File: Enables reading the contents of a file.
Write to File: Enables writing data to a file, either appending or overwriting existing content.
Delete File: Deletes a file from the filesystem.
Directory Operations:

Create Directory: Creates a new directory within the filesystem.
List Contents: Lists all files and subdirectories within a specified directory.
Delete Directory: Deletes an empty directory from the filesystem.
Filesystem Operations:

Mount/Unmount: Initializes and shuts down the filesystem.
Navigate: Allows navigation through directories, ensuring paths are valid.

4. Implementation Details

The filesystem is implemented using object-oriented principles in C++. Each class encapsulates data members and methods relevant to its responsibilities. For instance, the File class manages file content and attributes, while the Directory class manages collections of files and subdirectories.

Data structures like maps or lists are used to organize files and directories within the Filesystem class. Methods ensure proper error handling and validation of user inputs to maintain data integrity and prevent unauthorized access.

5.Uml Class Diagram and Use case diagram

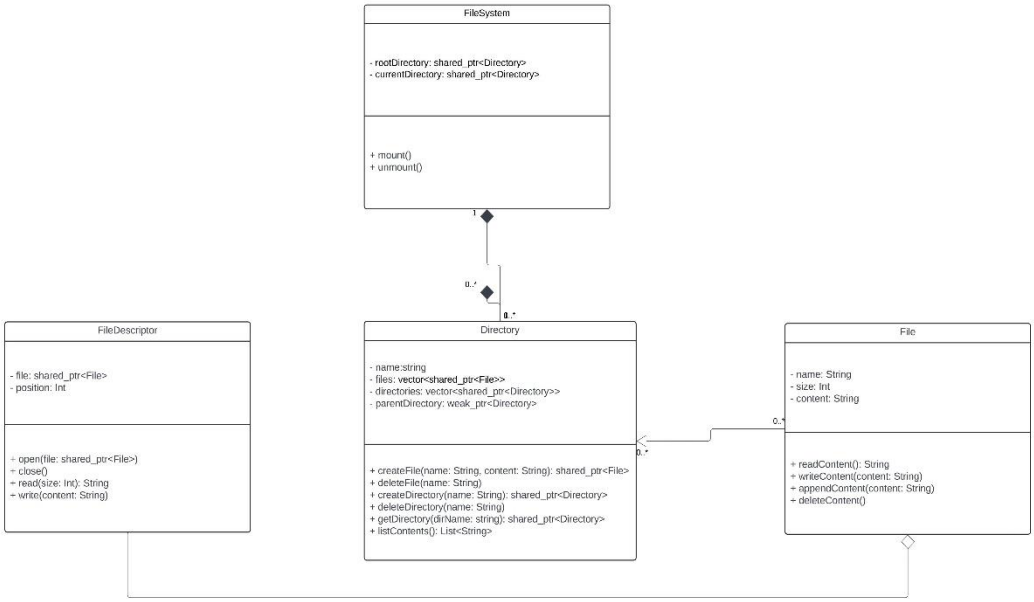


FIG 1: Use case diagram

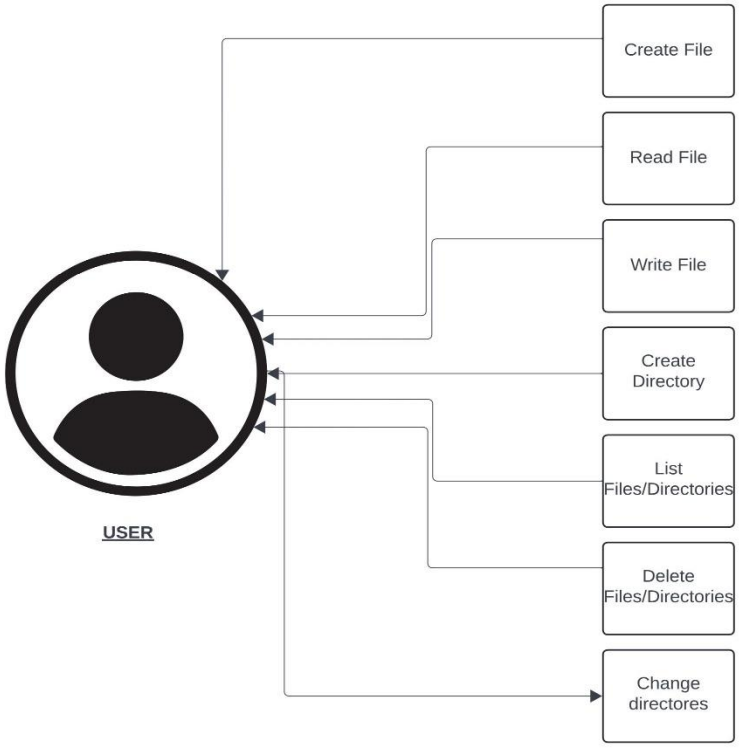


FIG 2: Use case

6.Challenges

Implementing Catch2 tests for the filesystem project presented several challenges that impacted the testing process and effectiveness of the test suite.

Firstly, managing the filesystem state across tests proved to be complex. The nature of filesystem operations, such as creating, modifying, and deleting files and directories, meant that each test could potentially alter the state of the filesystem in unexpected ways, leading to non-deterministic test results and difficult-to-reproduce failures.

Secondly, ensuring proper isolation and cleanup between tests was challenging. Filesystem operations often persisted beyond the scope of individual tests, requiring careful setup and teardown procedures to ensure that each test started with a consistent state and did not interfere with subsequent tests.

Additionally, mocking filesystem interactions for testing purposes was not straightforward. While Catch2 provides mechanisms for mocking, accurately simulating filesystem behaviors such as permissions, error conditions, and edge cases required meticulous setup and sometimes custom mock implementations.

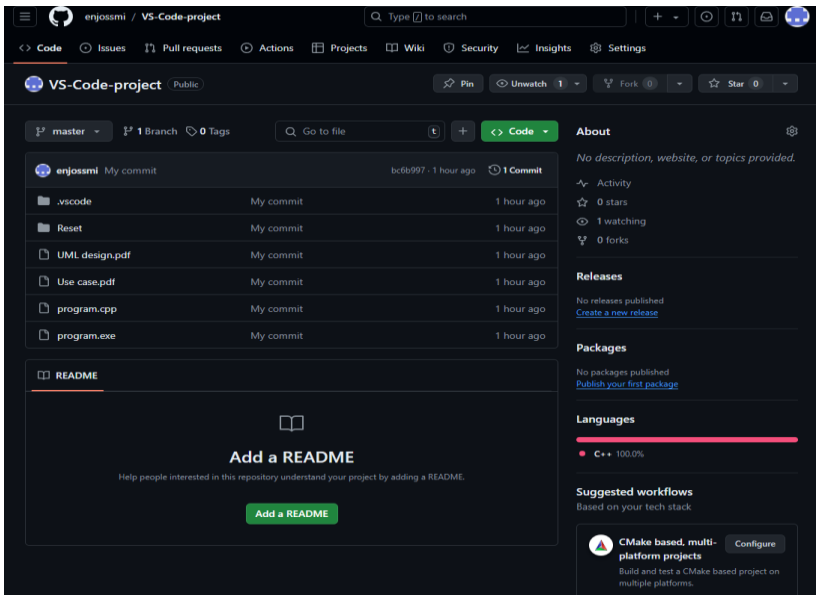
Moreover, the performance impact of filesystem operations in tests was notable. Tests involving large files or directories could be slow, affecting the overall test execution time and developer productivity, particularly in environments with frequent test runs or continuous integration setups.

Lastly, comprehensively testing error handling paths and edge cases related to filesystem operations required significant effort. Validating scenarios like permission denied errors, handling of non-existent files, and disk space limitations necessitated thorough testing setups and coverage to ensure robustness in real-world usage.

In conclusion, while Catch2 provided a robust framework for unit testing, the unique challenges posed by filesystem interactions required careful consideration and specialized approaches to achieve effective and reliable test coverage for the filesystem project.

7.Version Control

For me, using version control, like Git, was essential during the filesystem project in C++. It allowed me to manage my code changes effectively, keeping track of every edit I made over time. Working on my own branch meant I could develop new features or fix issues without worrying about affecting the main codebase. Git's backup feature also gave me peace of mind, ensuring that my work was safe and recoverable if anything went wrong. Overall, Git simplified how I managed my code, enabling a more organized and efficient development process.



CONCLUSION

In conclusion, the implementation of this filesystem in C++ provides essential functionalities for managing files and directories. The design ensures modularity, encapsulation, and ease of use through well-defined class structures. Future enhancements could include additional features such as file locking, symbolic links, and improved error handling mechanisms to further enhance reliability and usability.

This filesystem project demonstrates the application of object-oriented programming concepts in system-level software development, providing a robust foundation for further exploration and refinement in filesystem management.