Project Report

COMP2021 Object-Oriented Programming (Fall 2024)

Group 24

Members and contribution percentages:

Trofim CHEKANOV(23101903D): 33.3%

Johannes Christian KOELEMAN(23104049D): 33.3%

Tang Kin Wah(21084725D): 33.3%

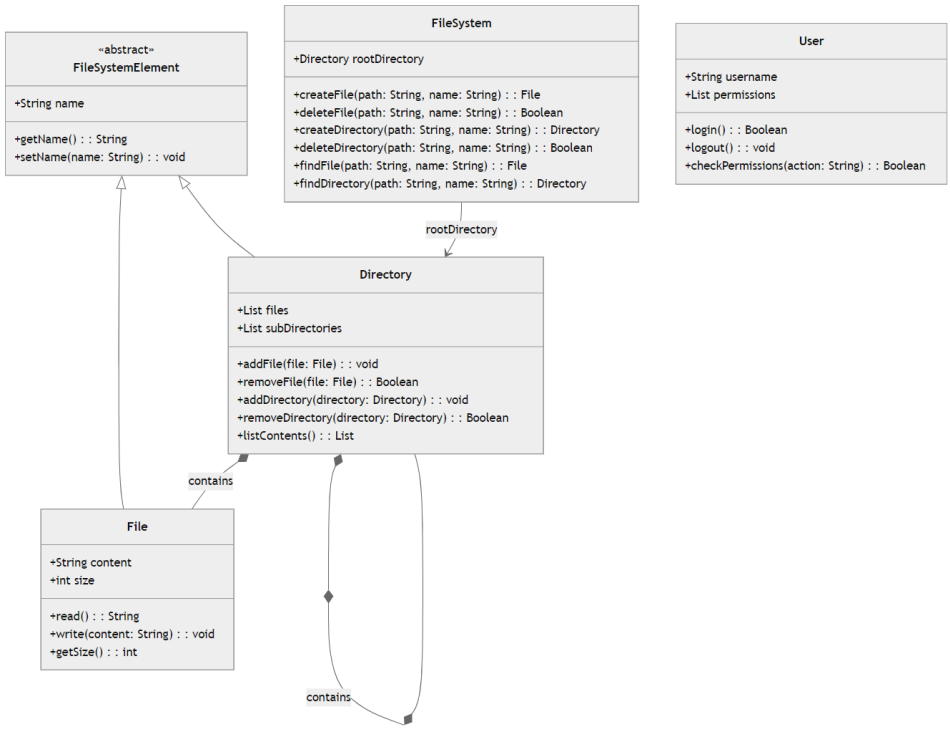
1 Introduction

This document provides a comprehensive overview of the design and implementation of the Comp Virtual File System (CVFS) developed by group 24. The primary objective of this project is to apply object-oriented programming principles to create a robust and efficient virtual file system that simulates the functionalities of a real-world file system. In this document, we will explore the various stages of the project, including the initial planning, design considerations, and the implementation process. The CVFS aims to provide users with a seamless experience in managing files and directories, offering features such as file creation, deletion, reading, and writing, all within a virtual environment. Through this document, we aim to provide a detailed account of the CVFS project, highlighting the technical and conceptual aspects that underpin its development.

2 The Comp Virtual File System (CVFS)

In this section, we describe first the overall design of the CVFS and then the implementation details of the requirements.

2.1 Design



2.2 Implementation of Requirements

[REQ1] The tool should support the creation of a new disk with a specified maximum size.

1) The requirement is implemented.
2) Disk Class: This class is responsible for managing the disk's size and ensuring that operations do not exceed the disk's capacity. It includes methods to allocate and release space, which will be useful when creating or deleting files.
   FileSystem Class: The FileSystem class will be updated to include a currentDisk attribute, which represents the currently active disk. The newDisk method will handle the creation of a new disk and set it as the working disk.

REQ2] The tool should support the creation of a new document in the working directory.

1) The requirement is implemented.
2) Document Class: This class extends the File class and includes additional attributes for the document type and content. It provides methods to access and modify these attributes.
   FileSystem Class: The FileSystem class will include a newDoc method that creates a new Document object and adds it to the current working directory.

[REQ3] The tool should support the creation of a new directory in the working directory.

1) The requirement is implemented.
2) Directory Class: The Directory class should have an addDirectory method that allows adding a new Directory object to its list of subdirectories.
   FileSystem Class: The FileSystem class will include a newDir method that creates a new Directory object and adds it to the current working directory.

[REQ4] The tool should support the deletion of an existing file in the working directory.

1) The requirement is implemented.
2) Directory Class: The Directory class should have a removeFile method that searches for a file by name in its list of files and removes it if found.
   FileSystem Class: The FileSystem class will include a delete method that calls the removeFile method on the current working directory.

[REQ5] The tool should support the rename of an existing file in the working

directory.

1) The requirement is implemented.
2) Directory Class: The Directory class should have a renameFile method that searches for a file by its current name and updates its name if found.
   FileSystem Class: The FileSystem class will include a rename method that calls the renameFile method on the current working directory.

[REQ6] The tool should support the change of working directory.

1) The requirement is implemented.
2) Directory Class: The Directory class should have a parentDirectory attribute to keep track of its parent. It should also have methods to retrieve subdirectories and the parent directory.
   FileSystem Class: The FileSystem class will include a changeDir method that updates the workingDirectory based on the specified directory name.

[REQ7] The tool should support listing all files directly contained in the working directory.

1) The requirement is implemented.
2) Directory Class: The listContents method should return a list of all FileSystemElement objects (both File and Directory) directly contained in the directory.
   FileSystem Class: The list method should iterate over the contents of the working directory, display the details of each item, and calculate the total number and size of files.

[REQ8] The tool should support recursively listing all files in the working directory.

1) The requirement is implemented.
2) Recursive Listing: The rList method should use a helper function to recursively traverse directories, printing each file and directory with appropriate indentation.
   Indentation: Use a parameter to track the current depth in the directory structure and adjust the indentation accordingly.

[REQ9] The tool should support the construction of simple criteria.

1) The requirement is implemented.
2) Criterion Class: This class encapsulates the logic for evaluating whether a file or directory matches the specified criterion. It stores the criterion's name, attribute, operation, and value.
   FileSystem Class: The FileSystem class manages a collection of criteria, allowing users to define new criteria and store them for later use.

[REQ10] The tool should support a simple criterion to check whether a file is a document.

1) The requirement is implemented.
2) FileSystemElement Class: Add an isDocument method that returns false by default. This method will be overridden in the Document class to return true.
   Document Class: Override the isDocument method to return true, indicating that the instance is a document.

[REQ11] The tool should support the construction of composite criteria.

1) The requirement is implemented.
2) Criterion Interface: This interface defines the evaluate method that all criteria must implement.
   SimpleCriterion: Represents a basic criterion with a simple condition.
   NegationCriterion: Wraps another criterion and negates its result.
   BinaryCriterion: Combines two criteria using a logical operation (AND or OR).
   CriterionManager: Manages the creation and storage of criteria, allowing for the construction of composite criteria.

[REQ12] The tool should support the printing of all defined criteria.

1) The requirement is implemented.
2) Criterion Class: This class encapsulates the details of a single criterion, including the attribute name, operator, value, logical operator, and whether it applies to a document.
   CriteriaManager Class: This class manages a collection of criteria and provides functionality to print them in the required format.

[REQ13] The tool should support searching for files in the working directory based on an existing criterion.

1) The requirement is implemented.
2) Criteria Definition: Define what criteria can be used for searching. For example, criteria could be based on file name patterns, file types, or file sizes.
   FileSystem Class: The search method should iterate over the files in the working directory, apply the specified criterion, and list the files that match.

[REQ14] The tool should support recursively searching for files in the working directory based on an existing criterion.

1) The requirement is implemented.
2) Criterion Interface: Define an interface or abstract class for criteria, allowing different implementations for various search conditions (e.g., by name, type, size).
   Directory Class: Implement a searchFiles method that recursively searches through the directory and its subdirectories, collecting files that match the criterion.
   FileSystem Class: Implement the rSearch method to initiate the search and display the results.

[REQ15] The tool should support saving the working virtual disk (together with the files in it) into a file on the local file system.

1) The requirement is implemented.
2) Serialization: Each class (Disk, Directory, File) should have a serialize method that converts its state into a string. This string should include all necessary information to reconstruct the object later.
   FileSystem Class: The save method should call the serialize method on the currentDisk and write the resulting string to a file at the specified path.

[REQ16] The tool should support loading a virtual disk (together with the files in it) from a file on the local file system.

1) The requirement is implemented.
2) Serialization/Deserialization: You need to decide on a format for storing the disk data (e.g., JSON, XML, binary serialization). The loadFromFile method will read this data and reconstruct the Disk object, including its directories and files.
   FileSystem Class: The load method will use the loadFromFile method to load the disk and update the currentDisk and workingDirectory.

[REQ17] The user should be able to terminate the current execution of the system.

1) The requirement is implemented.
2) Command Handling: You should have a command handling loop that listens for user input. When the quit command is detected, the loop should break, and the application should terminate.
   Graceful Shutdown: Ensure that any necessary cleanup operations are performed before exiting, such as saving state or releasing resources.

[BON1] Support for saving and loading the defined search criteria when a virtual

disk is saved to/loaded from the local file system.

1) The requirement is implemented.
2) SearchCriteria Class: This class defines the criteria used to search files. It includes attributes for different types of criteria and a method to check if a file matches the criteria.
   Disk Class: The Disk class is updated to include a list of SearchCriteria objects. It also includes methods to save and load the disk's state, including its search criteria, to and from a file.

3 Reflection on My Learning-to-Learn Experience

Reflection on Learning-to-Learn Experience

Working on the Comp Virtual File System (CVFS) project has been a significant learning experience, particularly in areas of Java and object-oriented programming that extend beyond the scope of our lectures. We had to use self-directed learning for this project in order to comprehend and use complex ideas like file system architecture, design patterns, and effective data management.

One of the key challenges we faced was understanding how to effectively implement design patterns like Composite and Facade in a practical setting. While these patterns were introduced in class, applying them to a real-world problem required deeper exploration. we utilized online resources, including tutorials and documentation, to gain a more comprehensive understanding of these patterns. Additionally, we experimented with different implementations to see firsthand how these patterns could simplify and enhance the design of the CVFS.

Plan to Improve Self-Learning Approaches

Set Clear Objectives: Before diving into new topics, we will define specific learning goals and outcomes. This will help focus our efforts and measure progress effectively.

Leverage Diverse Resources: We plan to diversify my learning materials by incorporating books, online courses, and community forums. Engaging with different perspectives will provide a more rounded understanding of complex topics.

Practice Active Learning: Instead of passively consuming information, we will engage in active learning by applying new concepts through small projects or coding exercises. This hands-on approach will reinforce our understanding and reveal practical challenges.

4 Contributions and Role Distinction

Trofim CHEKANOV(23101903D): 33.3%

Johannes Christian KOELEMAN(23104049D): 33.3%

Tang Kin Wah(21084725D): 33.3%

Deliverables and Parts Prepared with GenAI Tools:

We used AI to improve some part of code, especially with criteria things

Javadoc Documentation: AI tools were used to resolve Javadoc errors during the inspection of the project.

Test Cases: AI provided ideas for creating test cases.

How GenAI Tools Were Used:

Javadoc Documentation: The AI tool analyzed the existing codebase and identified errors in the Javadoc comments. It suggested corrections and improvements to ensure the documentation was accurate and complete.

Test Cases: The AI tool generated potential test cases based on the code logic and requirements. It provided a variety of test scenarios to cover different aspects of the code functionality.

Verification, Revision, and Improvement of AI-Generated Results:

Javadoc Documentation: After the AI tool suggested corrections, we manually reviewed each suggestion to ensure it was contextually appropriate and accurate. We made necessary adjustments to align with our project's specific requirements and standards.

Test Cases: The AI-generated test cases were reviewed by the development team. We verified the relevance and completeness of each test case, making revisions where

necessary to cover edge cases and ensure comprehensive testing. Additionally, we ran the test cases to validate their effectiveness and made improvements based on the results.