Project Report

COMP2021 Object-Oriented Programming (Fall 2024)
Groupmates: Kam Chun Yu, Lee Ming Hin, Wong Chi Chung, Tsang Hing Ho
Members and Contribution Percentages:
Kam Chun Yu: 33⅓%
Lee Ming Hin: 33⅓%
Wong Chi Chung: 33⅓%
Tsang Hing Ho: 0%

✪ Introduction

This document details the design and implementation of the Comp Virtual File System (CVFS). CVFS is a command-line interface tool that emulates a virtual file system, allowing users to create, manage, and manipulate virtual disks, directories, and documents. The primary objectives were to apply object-oriented principles, design patterns, and advanced Java programming techniques to build a robust and user-friendly virtual file system.

✪ The CVFS

Design

Overview

CVFS is architected using object-oriented principles to ensure modularity, scalability, and maintainability. The system comprises the following key components:

- Core Model Classes: Represent the fundamental entities such as VirtualDisk, Directory, Document, and abstract File.

- Command Pattern Implementation: Facilitates undo and redo functionalities through encapsulated command objects.

- Criteria System: Enables users to define and utilize various criteria for searching and filtering files.

- Application Interface: Provides a command-line interface for user interactions.

Class Diagram

- CVFS Class:

① Manages the virtual disk, working directory, criteria, and command stacks for undo/redo.
② Implements core functionalities corresponding to user commands.

- File (Abstract Class):

① Serves as the base class for *Directory* and *Document*.
② Contains common attributes like name and methods for name validation.

- Directory Class:

① Extends *File*.
② Contains a list of *File* objects, representing files and subdirectories.
③ Provides methods to add, remove, and retrieve files.

- Document Class:

① Extends *File*.
② Contains attributes like *type* and *content*.
③ Implements size calculation based on content length.

- VirtualDisk Class:

① Represents a virtual disk with a maximum size.
② Contains a root *Directory*.

- Criterion Interface and Implementations:

① Defines a method *evaluate* to determine if a file meets specific criteria.
② Implemented by classes like *SimpleCriterion*, *IsDocumentCriterion*, *NegationCriterion*, and *BinaryCriterion*.

- Command Interface and Implementations:

① Defines *undo* and *redo* methods.
② Implemented by classes like *NewDocCommand*, *NewDirCommand*, *DeleteCommand*, *RenameCommand*, *ChangeDirCommand*, and *NewCriterionCommand*.

- Application Class:

① Contains the *main* method.

② Handles user input and maps commands to CVFS functionalities.

Design Patterns

- Command Pattern:

➢ Employed to implement undo and redo functionalities.

➢ Each user action is encapsulated as a command object, maintaining a history stack for reversible operations.

- Strategy Pattern:

➢ Applied in the criteria system where different evaluation strategies (*SimpleCriterion*, *NegationCriterion*, *BinaryCriterion*) are interchangeable based on user-defined criteria.

Component Interaction

- User Interaction:

1. Users interact with CVFS via the command-line interface provided by the *Application* class.
2. Commands entered by the user are parsed and delegated to corresponding methods in the *CVFS* class.

- Command Execution:

1. Each command (e.g., *newDoc*, *delete*) triggers specific operations within the CVFS model.
2. Successful commands are pushed onto the *undoStack*, enabling reversible actions.

- Criteria Management:

1. Users can define custom criteria for searching files.
2. The criteria system allows complex queries through logical operations and negations.

- Persistence:

1. CVFS supports saving and loading the state of the virtual disk and criteria to

and from files, ensuring data persistence across sessions.

Implementation of Requirements

[REQ1] *newDisk* Command

1. **Implementation Details:**
    - Initializes a new *VirtualDisk* instance with the specified size.
    - Sets the *workingDirectory* to the root directory of the new disk.
    - Clears existing criteria and command stacks to reset the system state.
2. **Error Conditions and Handling:**
    - Throws an exception if the number of tokens is not equal to 2.
    - Handles *NumberFormatException* if the disk size is not a valid integer.

[REQ2] *newDoc* Command

1. **Implementation Details:**
    - Creates a new *Document* with the provided name, type, and content.
    - Adds the document to the current *workingDirectory*.
    - Updates the virtual disk size and logs the action for undo functionality.
2. **Error Conditions and Handling:**
    - Throws an exception if no virtual disk is loaded.
    - Validates the number of tokens.
    - Ensures the document type is among the allowed types (txt, java, html, css).

[REQ3] *newDir* Command

1. **Implementation Details:**
    - Creates a new *Directory* with the specified name.
    - Adds the directory to the current *workingDirectory*.
    - Updates disk size and logs the action for potential undo operations.
2. **Error Conditions and Handling:**
    - Throws an exception if no virtual disk is loaded.

- Validates the number of tokens.
- Ensures no duplicate directory names exist within the current directory.

[REQ4] ***delete*** **Command**

1. **Implementation Details:**
    - Deletes a specified *Document* or *Directory* from the *workingDirectory*.
    - Updates the virtual disk size accordingly.
    - Logs the deletion for undo functionality.
2. **Error Conditions and Handling:**
    - Throws an exception if no virtual disk is loaded.
    - Validates the number of tokens.
    - Checks if the specified file exists before attempting deletion.

[REQ5] ***rename*** **Command**

1. **Implementation Details:**
    - Renames an existing file within the *workingDirectory*.
    - Validates the new file name against predefined patterns.
    - Updates references and logs the renaming action for undo purposes.
2. **Error Conditions and Handling:**
    - Throws an exception if the virtual disk is not loaded.
    - Validates the number of tokens.
    - Ensures the old file exists and the new name does not duplicate existing files.

[REQ6] ***changeDir*** **Command**

1. **Implementation Details:**
    - Changes the current *workingDirectory* based on user input.
    - Supports navigating to parent directories using the .. notation.
    - Logs directory changes for potential undo actions.
2. **Error Conditions and Handling:**
    - Throws an exception if the virtual disk is not loaded.
    - Validates the number of tokens.
    - Ensures the target directory exists unless navigating to the root.

[REQ7] ***list*** **Command**

1. **Implementation Details:**
    - Lists all files and directories within the current *workingDirectory*.
    - Displays details such as name, type, and size.
    - Provides a summary of total files and cumulative size.
2. **Error Conditions and Handling:**
    - Throws an exception if no virtual disk is loaded.

[REQ8] *rList* Command

1. **Implementation Details:**
    - Recursively lists all files and directories within the current *workingDirectory* and its subdirectories.
    - Utilizes indentation to represent directory depth.
    - Aggregates total file count and size.
2. **Error Conditions and Handling:**
    - Throws an exception if no virtual disk is loaded.

[REQ9] *newSimpleCri* Command

1. **Implementation Details:**
    - Creates a new simple criterion based on attribute name, operator, and value.
    - Adds the criterion to the *criteriaMap* for future searches.
2. **Error Conditions and Handling:**
    - Validates the number of tokens.
    - Ensures the criterion name matches the required pattern and does not duplicate existing criteria.

[REQ10] *IsDocument* Criterion

1. **Implementation Details:**
    - A predefined criterion that evaluates whether a file is a *Document*.
2. **Error Conditions and Handling:**
    - (predefined)

[REQ11] *newNegation* and *newBinaryCri* Commands

1. **Implementation Details:**
    - *newNegation*: Creates a new criterion that negates an existing criterion.
    - *newBinaryCri*: Combines two existing criteria using logical operators (&&, ||).

2. **Error Conditions and Handling:**
    - Validates the number of tokens.
    - Ensures criterion names follow the naming pattern and that referenced criteria exist.

[REQ12] ***printAllCriteria*** **Command**

1. **Implementation Details:**
    - Prints all defined criteria along with their string representations.
2. **Error Conditions and Handling:**
    - N/A (always displays current criteria).

[REQ13] ***search*** **Command**

1. **Implementation Details:**
    - Searches for files in the current *workingDirectory* that match a specified criterion.
    - Displays matching files and aggregates their count and size.
2. **Error Conditions and Handling:**
    - Validates the number of tokens.
    - Ensures the specified criterion exists.

[REQ14] ***rSearch*** **Command**

1. **Implementation Details:**
    - Recursively searches for files matching a specified criterion across the *workingDirectory* and all subdirectories.
    - Displays matching files and provides a summary of results.
2. **Error Conditions and Handling:**
    - Validates the number of tokens.
    - Ensures the specified criterion exists.

[REQ15] ***save*** **Command**

1. **Implementation Details:**
    - Saves the current state of the *VirtualDisk* and *criteriaMap* to a specified file using serialization.
    - Enables users to persist their virtual file system state for later retrieval.
2. **Error Conditions and Handling:**
    - Validates the number of tokens.

- Handles I/O exceptions during the save process, providing relevant error messages.

[REQ16] ***load*** **Command**

1. **Implementation Details:**
   - Loads the state of the *VirtualDisk* and *criteriaMap* from a specified file using deserialization.
   - Restores the virtual file system to a previously saved state.
2. **Error Conditions and Handling:**
   - Validates the number of tokens.
   - Handles I/O and class-not-found exceptions during the load process, providing relevant error messages.

[Bonus] ***Undo*** **and** ***Redo*** **Commands**

1. **Implementation Details:**
   - Utilizes two stacks (*undoStack* and *redoStack*) to manage undo and redo operations.
   - Each user command that modifies the state is encapsulated as a *Command* object and pushed onto the *undoStack*.
   - The *undo* method reverses the last action by invoking the *undo* method of the last command.
   - The *redo* method reapplies the last undone action by invoking the *redo* method of the last command.
2. **Error Conditions and Handling:**
   - Throws exceptions when attempting to undo with an empty *undoStack* or redo with an empty *redoStack*.
   - Ensures the command stacks are appropriately managed after each operation.

✪ Reflection

During the development of the Comp Virtual File System (CVFS), We faced multiple challenges that required us to go beyond the lecture materials and enhance our self-learning skills. One major challenge was understanding and applying design patterns like Command and Strategy. Initially, we struggled to grasp their practical applications, but by studying examples, documentation, and trial and errors, we gained clearer understanding. Another challenge was implementing persistence through Java's serialization. Working with object

graphs and understanding how transient fields worked was initially confusing, but experimenting with code helped us overcome these difficulties. Additionally, designing a flexible and extensible criteria system for searching files was somewhat complicated. After exploring different approaches, the program worked effectively.

To address these challenges, we tried several strategies. We broke down topics into manageable sections and studied them systematically. Collaborating with together is invaluable, as discussing complex concepts and problem-solving together helped clarify our understanding. We frequently practiced by creating small prototypes to test and solidify our understanding of new concepts before implementing them into the project. These were also extremely helpful for learning best practices and troubleshooting later.

Looking ahead, we identified areas for improvement in our learning process. First, we plan to adopt better time management techniques to maintain focus and productivity during study sessions. We also aim to deepen our understanding of design patterns by exploring additional patterns. This approach will allow us to continuously evaluate and improve our learning strategies.

Overall, this project was a valuable learning experience that not only strengthened our understanding of object-oriented programming concepts and advanced Java techniques but also improved our ability to learn with groupmates. Through structured self-study, teamwork, and hands-on practice, we were able to overcome challenges and deliver a robust virtual file system. By continuously improving my learning methods and expanding my technical knowledge, I am confident I can handle future challenges more effectively.

✪ Conclusion

The development of the Comp Virtual File System (CVFS) was a comprehensive exercise in applying object-oriented programming principles, design patterns, and advanced Java concepts. The project provided a user input in the terminal to manipulate files using the functions implemented in the program, which not only reinforced theoretical knowledge but also significantly enhanced our practical programming skills and problem-solving abilities. Moving forward, the lessons learned and strategies developed during this project will serve as a solid foundation for tackling more complex software engineering challenges.