Project Report

COMP2021 Object-Oriented Programming (Fall 2024)

The COMP Virtual File System (CVFS) Project

Group 10

Please run the `Application` class to start the CVFS application

Members and Contributions:

YANG Xikun [23101291d]: 25%	Overall Design, Coding and Report
YANG Jinkun [23113164d]: 25%	Model Validating and Improving
REN Yixiao [23102145d]: 25%	Test Engineer
Arda EREN [23104019d]: 25%	User Manual

This project not only meets the requirements and bonuses,
but also explores the design principles and programming philosophies,
and brings them to reality.
I'm very happy that this project was designed and completed perfectly.

YANG Xikun, Nov 2024

CONTENTS (The most important parts are noted blue)

I. INTRODUCTION

The goal of this project is to develop an in-memory Virtual File System (VFS), named the COMP VFS (CVFS), in Java.

A VFS is usually built on top of a host file system to enable uniform access to files located in different host file systems, **while the CVFS simulates a file system in memory.**

CVFS allows you to create virtual disks and manage files within them. CVFS supports you to create criteria to filter specific files. You can also load/save the virtual disks and criteria file from/to your local file system.

This document describes the design and implementation of the CVFS by Group 10.

Project CVFS is part of the course COMP2021 Object-Oriented Programming at the Hong Kong Polytechnic University.

II. 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. We also used class diagrams to give an overview of the system design and explain how different components fit together.

If you need to quickly check out **the most important parts**, please see these sections first:

2.1A Design: Design Principles

- **Modularity:** The project application is divided into four parts: Model, Controller, Service, and View. Each part is highly cohesive and lowly coupled, interacting through well-defined interfaces. We strictly adhere to the MVC design pattern.

- **Separation of Responsibilities:** The four parts have distinct and clear responsibilities. Each part focuses on its own duties, without concern about issues not in its scope.

- **Isolation:** Each part strictly limits the interfaces provided to other parts, so to prevent excessive interference and coupling.

- **High-Fidelity Simulation:** Since the CVFS is a virtual file system, we aim to make it [as closely as possible] to real file systems or existing virtual file systems. We try our best in our simulations, including interface and responsibility simulations. Some details are too complicated to simulate, so we compromise by making them appear realistic to other parts, rather than focusing on every internal detail.

- **Safest structure:** We require all structures to be safe, especially ensuring that the safety of lower-level structures does not depend on upper-level structures. Reasonable but redundant safety checks are allowed. For undo and redo operations, we do not assume they can always be executed. A series of checks must be performed before any operation is executed.

2.1B Design: Project Structure Design

In this chapter, we will focus on our structural design.

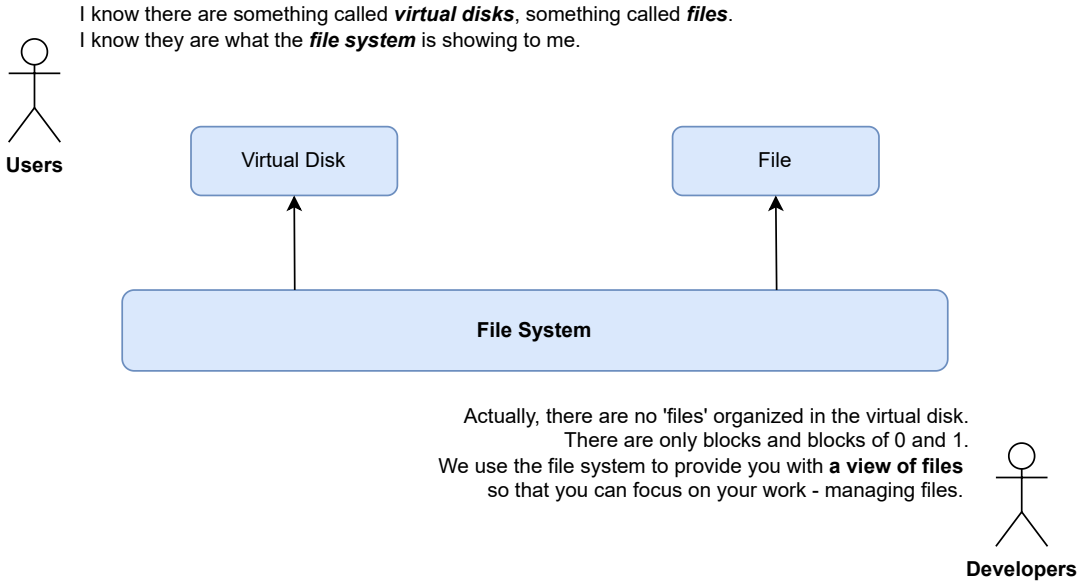
We try to explain our model in detail.

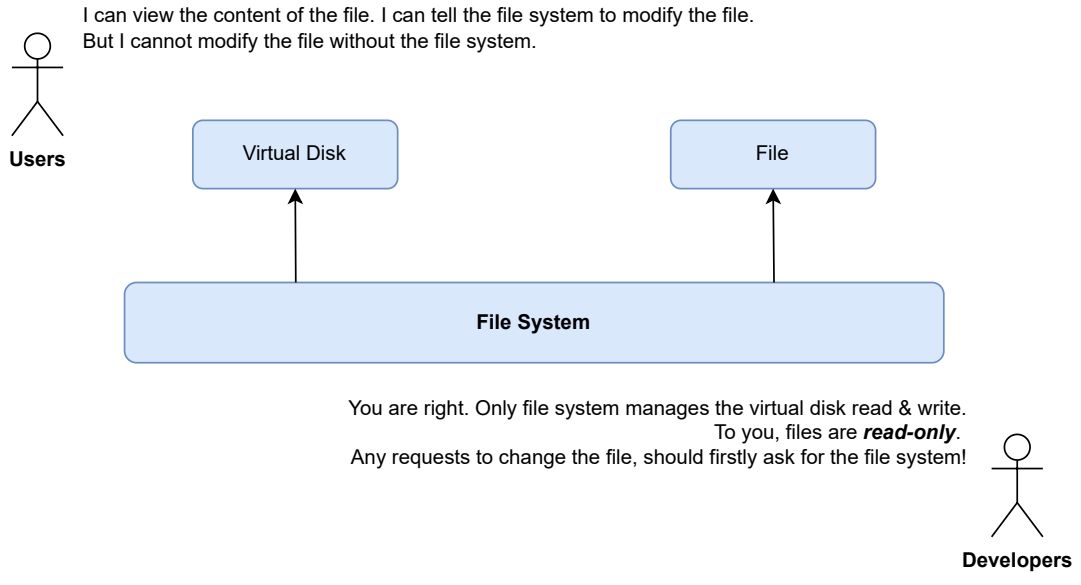
2.1B.1 The Overall File System Model

Since the core of CVFS is a file system, the very basic part is to simulate a file system.

A file system abstracts the concepts of files for the users. More specifically, in a disk, there is not a clear concept of files: a file might be split into multiple parts and stored separately.

The users, as well as other parts above the file system, such as the View, Controller, and Service in the MVC model mentioned later, all rely on the file system to interact with the virtual disk.





2.1B.2 *The Fetch-Decode-Execute Cycle*

Similar to *the fetch-decode-execute cycle* of a CPU, our system uses a cycle to keep handling user commands:

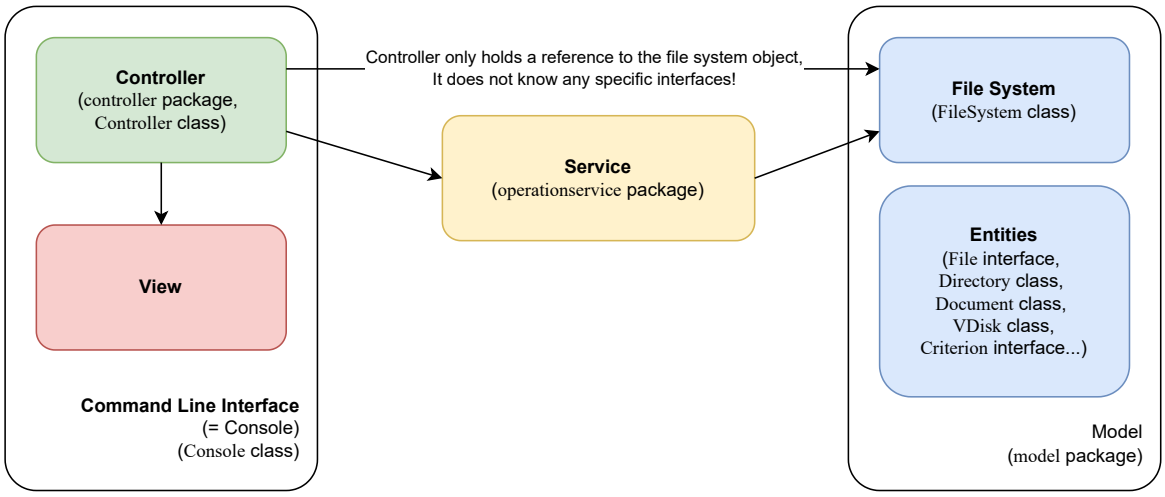
1. Receive user's input.
2. Parse the input.
3. Check if the input is valid.
4. Convert the input into an operation.
5. Execute the operation.
6. Present the result back to user.
7. Go back to 1.

This cycle led us to develop an **Enhanced MVC Model**. See next section.

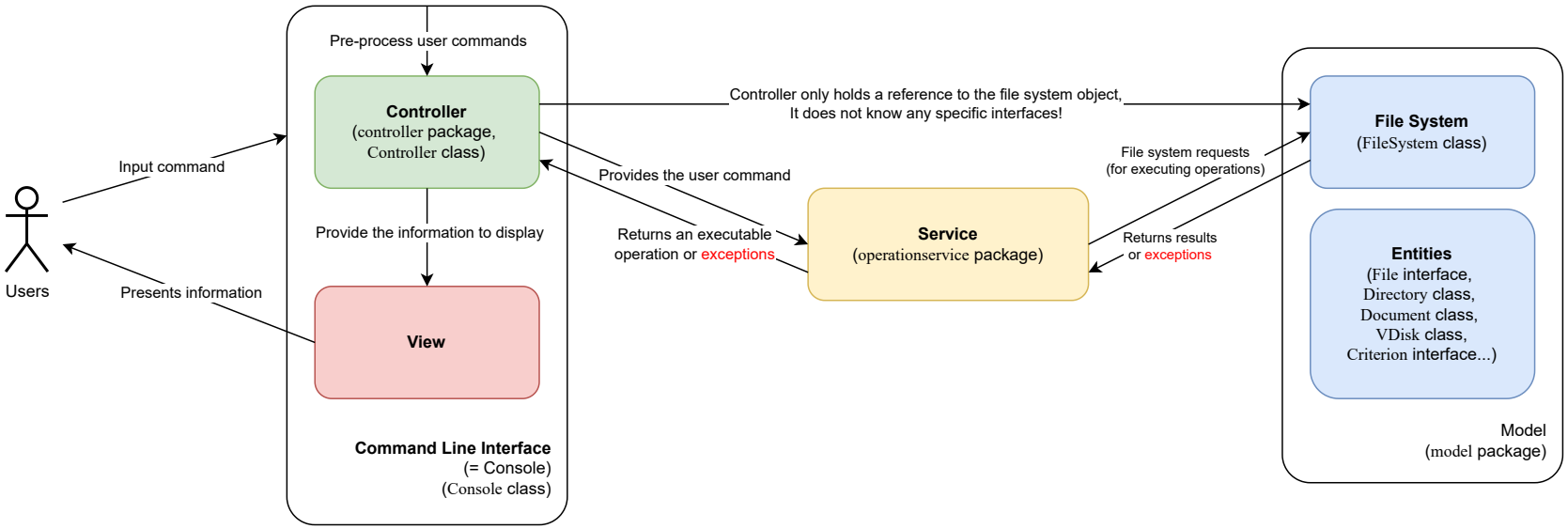
2.1B.3 The Enhanced MVC Model

There are four parts in our application, the **Model**, **Controller**, **Service** and **View**.

Below is a simple diagram showing their relationships, where arrows indicate dependencies (e.g., A → B means A depends on B's interfaces).



For detailed information on these parts interact with each other and with the user, see next three pages.


Pre-process user commands
Controller
(controller package,
Controller class)
Controller only holds a reference to the file system object,
It does not know any specific interfaces!
File System
(FileSystem class)
Input command
Provides the user command
File system requests
(for executing operations)
Provide the information to display
Returns an executable
operation or exceptions
Service
(operationservice package)
Returns results
or exceptions
Entities
(File interface,
Directory class,
Document class,
VDisk class,
Criterion interface...)
Users
Presents information
View
Command Line Interface
(= Console)
(Console class)
Model
(model package)

Further explanation

Application (as `Application` class) is the highest layer.

- It provides a main entry point.
- All other parts should be booted by it.

The Model (as `model` package) is the core part of the system.

- It implements several **Entities** (as `entities` package).
  - The entities include:
    - **Files** (as `File` interface)
    - **Directories** (as `Directory` class)
    - **Documents** (as `Document` class)
    - **Criteria** (as `Criterion` class)
    - **Virtual Disks** (as `VDisk` class)
  - In the view of other parts, entities are only piles of read-only information. For example, a `File` is just a file, and it cannot control is organization on the virtual disk.
  - Constructors like `new VDisk(...)` and `new Document(...)` are allowed, but nominally, they must call the File System to take effect.
- It has another part to maintain the application's state and process all file system requests, defined as the **File System** (as `FileSystem` class).
  - Any attempts to modify the Entities or find their relational context must request the File System first. The attempts include:
    - Create new files (requires allocating disk space)
    - Remove files (requires deallocating disk space)
    - Rename files (requires disk read & write)
    - Modify documents (also requires disk read & write)
    - Find the parent of files (requires disk read)
    - Find if a file exists in the directory (also requires disk read)

(Continue)

The Controller (as `Controller` class) is the application's controller.

- It handles user input, hence it is partially in the Console.
- It requests operations (as `Operation` class) from the Service, and then executes them.

Service (as `service` and `operationservice` package) converts the commands into operations and provides simple and unified interfaces to the Controller, so the Controller can perform the operation in few steps.

- It is designed to alleviate the Controller's load, because there are too many operations.
- It needs to use the interfaces provided by the Model. It does not know the implementation details of the Model.
- It serves as a bridge between the Controller and the Model.

The Console (as `Console` class) is the core of the command line interface.

- It includes the entire **View** part, because it handles all information displayed to the user.
- It is also a part of the Controller because users need to provide commands through the Console.

2.1B.4 The Cycle with The Enhanced MVC Model

After introducing the enhanced MVC model, we could show you a more detailed running cycle. This is the core of our program.

1. **[Controller, Console]** Receive user's input.
2. **[Controller, Console]** Parse the input.
   - Split the input by spacebars.
   - Process the quote marks and backticks.
3. **[Controller, Service]** Check if the input is valid.
   - Check if the command exists.
   - Check if the parameters of the command are correct.
   - If all correct, save the command's parameter information.
4. **[Service]** Convert the input into an operation.
   - Get the application's current state.
   - Generate an `Operation` object using the state information and the parameter information.
5. **[Controller]** Execute the operation.
   - Simply call the `exec()` method of the `Operation` object.
6. **[View, Console]** Present the result back to user.
   - If the operation executes successfully, present the success information.
   - Or, present the error information.
7. Go back to 1.

2.1B.5 High Cohesion and Low Coupling (also see: 2.1C.2)

We introduce the principle of **high cohesion and low coupling**. In our program, each part having clear responsibilities, and the interfaces to other parts are strictly limited.

The Model is the core part, it does not depend on any other parts. The Model's interfaces to other parts include:

- **[For Service and Controller]** The reference to the File System.

  ```
  // It is allowed to use such statements in the Service and Controller part.
  FileSystem fs = ...;
  ```

- **[Only for Service]** File system interfaces (such as retrieving files, creating files / allocating space, deleting files / deallocating space, and other file management operations).

  ```
  // These interfaces are provided, but only for the Service part:

  /* Methods involving virtual disk mounting and ejecting */
     • public void ejectVDisk(); // currently unavailable
     • public void mountVDisk(VDisk vDisk);
     • public void releaseResource();

  /* Public getters */
     • public Directory getRootDirectory();
     • public Directory getWorkingDirectory();
     • public String getWorkingDirectoryPathSafely();

  /* Public setter */
     • public void setNewWorkingDirectory(Directory newWorkingDirectory);

  /* File system responsibilities */
     • public TreeMap<String, File> getAllFiles(Directory directory);
     • public File findFile(Directory directory, String name);
     • public File getParent(File file);
     • public void storeFile(File file);
     • public void removeFile(File file);
     • public void renameFile(File file, String newName);
     • public void modifyDocument(File file, String newContent);

  /* Methods related to the criteria */
     • public void addCriterion(Criterion criterion);
     • public TreeMap<String, Criterion> getAllCriteria();
     • public Criterion findCriterion(String name);
     • public void removeCriterion(Criterion criterion);
  ```

(Continue)

- **[Only for Service and Read-only]** The Entities.

```
// Virtual disk interfaces: NONE

// File interfaces:
    • public String getName();
    • public String getFullname();
    • public long getSize();
    • public String getPath();

// Additional interfaces in Document:
    • public Document(String name, String type, String content, Directory
      parent);
    • public String getType();
    • public String getContent();

// Hence in the view of other parts,
// Files and Directories are just piles read-only information,
// looks like the abstract structures provided by the file system.
```

- **[Exception]** A general exception class (as `ModelException` class).

The Service encapsulates the second and third interfaces of the Model, greatly reducing the coupling between the Controller and the Model. The Service provides the following interfaces to the Controller:

- **[For Controller]** An **`exec()`** method to execute operations.

```
// You may see the following statements in the Controller:
String result = operation.exec();
console.printInformation(result);
```

- **[Exception]** Two exception classes (as `InvalidCommandException` exception and `OperationCannotExecuteException` exception), indicating that user commands from the Controller are invalid or operations cannot be executed.

It is clear that **the Controller** hardly depends on the Model; it only needs a reference to the file system. The Controller slightly depends on the Service, but it is minimal.

The View does not involve the Model or Service. It simply outputs the content as requested by the Controller.

2.1B.6 High-Fidelity Simulation

As we mentioned before, we try our best to simulate as much as possible. In our discussion above, it is clear that **the Controller, Service, and View fully aligns with the real file systems**. However, the entities part of the Model needs clarification.

- Our program does not maintain a space to simulate the actual storage of a virtual disk. Instead, our virtual disks store files by storing references. We have not simulated complex mechanisms, such as virtual page tables and space maps.

- And in order to keep code simple, some operations involving the disk read & write are not fully implemented in the `FileSystem` class. Instead, they might be in the `File` interface and `Directory` class.

- Luckily, we do not allow the files to operate tasks involving disk read & write in other parts, which greatly simulates the reality where real disk read & writes exist.

- We introduced a `@ModelInternalUse` annotation[1] to indicate that, to those things whose details are not aligned to the real world, and/or could cause serious issues if exposed to other parts, **they should remain internal in the Model.** The method with this annotation will also have a `__INTERNAL__` prefix. For example,

```
public void storeFile(File file) throws ... {
    checkAddressSpace(file);

    if (file.getSize() > currentVDisk.getFreeSpace()) {
        throw new ...;
    }

    // The following should be the actual disk storage logic.
    Directory parent = (Directory)file.__INTERNAL__getParent();
    if (parent.__INTERNAL__existsName(file.getName())) {
        throw new ...;
    }
    parent.__INTERNAL__add(file);
}
```

[1] Please also see 2.3.4.

(Continue)

Overall, in the view of other parts,

- The `FileSystem` class manages all responsibilities of the File System,
- The Entities just represents entities themselves, along with their information read from the virtual disk. More specifically,
    - The Entities themselves do not participate in any disk reading & writing activities.
    - **The Entities look like the structures abstracted by the file system.**

This is the best we can do, and we believe that not to over-simulate is compatible with the project's expectations.

2.1C Design: General Implementations

In this chapter, we provide a brief overview of the implementation logics behind some structures.

2.1C.1 Class List

Below, we have listed all the classes of our application.

The Application

- `Application` - The Application

The Model

- The Entities
  - `File` - Files
  - `Directories` - Directories
  - `Document` - Documents
  - Criteria
    - `Criterion` - Criteria
    - `CriterionFactory` - The Criterion Factory, used to generate `Criterion` objects
    - `IsDocument` - The IsDocument criterion
    - `NameCriterion` - The simple criterion with `attrType == name`
    - `TypeCriterion` - The simple criterion with `attrType == type`
    - `SizeCriterion` - The simple criterion with `attrType == size`
    - `NegationCriterion` - The negation criterion
    - `LogicAndCriterion` - The binary criterion with `logicOp == &&`
    - `LogicOrCriterion` - The binary criterion with `logicOp == ||`
  - `VDisk` - Virtual Disks
- The File System
  - `FileSystem` - The File System

The Controller

- `Controller` - The Controller

The Console

- `Console` - The Console

(Continue)

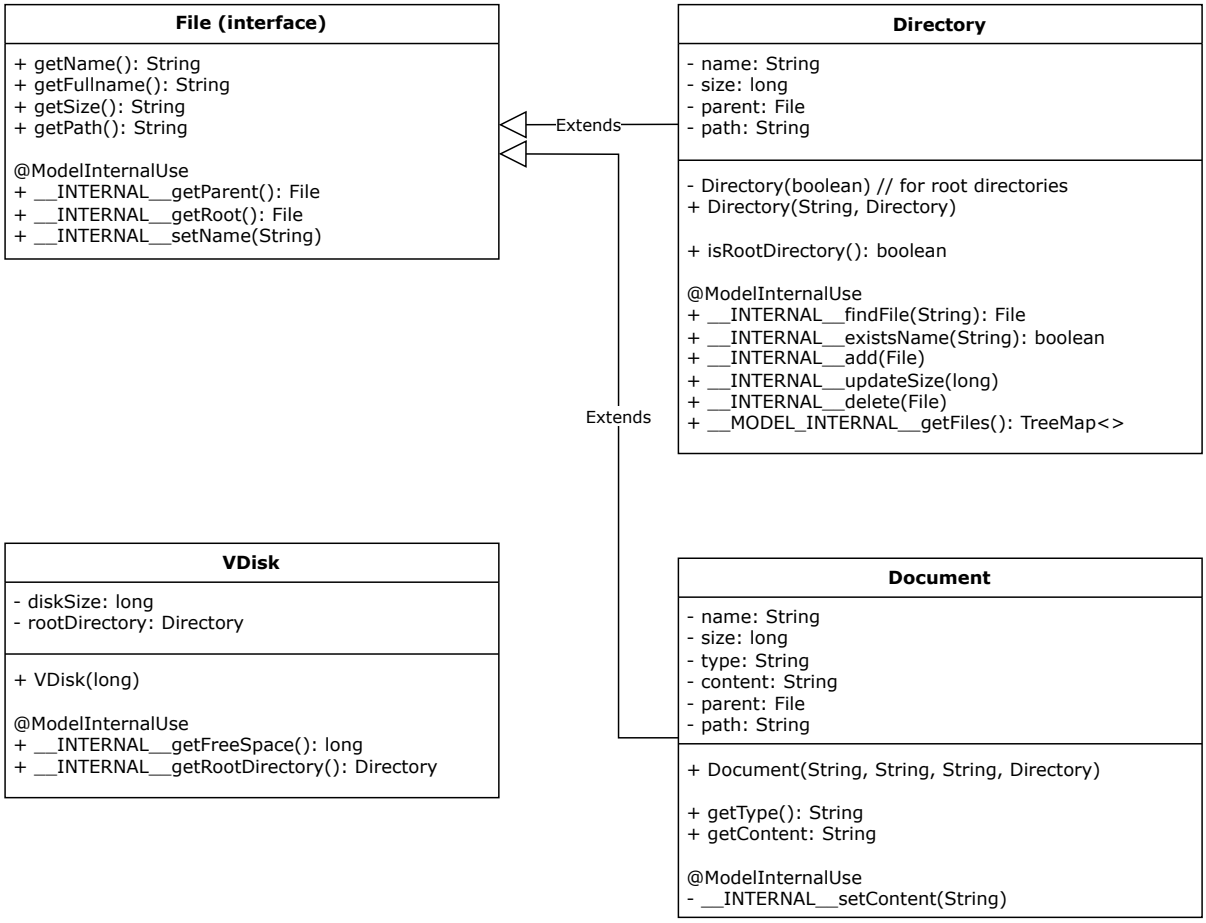
The Service (we also added some **exclusive features** to make the system more comprehensive)

- `OperationFactory` - The Operation Factory, used to generate `Operation` objects
- `OperationRecord` - The Operation Record System, used to record undoable operations
- Operations:
  - `Operation` - The general operations
  - `UndoableOperation` - The undoable operations
  - `FileUnrelatedUndoableOperation` - The undoable operations which are not related to files.
  - `NewDisk` - The operation of `newDisk` command
  - `NewDir` - The operation of `newDir` command
  - `NewDoc` - The operation of `newDoc` command
  - `ViewContent` - The operation of `view` command **(exclusive feature)**
  - `Remove` - The operation of `delete` command, the inverse operation of `PutBack`, `NewDir` and `NewDoc`
  - `PutBack` - The inverse operation of `Remove`
  - `ModifyContent` - The operation of `modify` command **(exclusive feature)**
  - `Rename` - The operation of `rename` command
  - `ChangeDir` - The operation of `changeDir` command
  - `List` - The operation of `list` command
  - `RList` - The operation of `rList` command
  - `NewSimpleCri` - The operation of `newSimpleCri` command
  - `NewNegation` - The operation of `newNegation` command
  - `NewBinaryCri` - The operation of `newBinaryCri` command
  - `RemoveCri` - The operation of `removeCri` command, the inverse operation of `PutBackCri`, `NewSimpleCri`, `NewNegation` and `NewBinaryCri` **(exclusive feature)**
  - `PutBackCri` - The inverse operation of `RemoveCri`
  - `PrintAllCriteria` - The operation of `printAllCriteria` command
  - `Search` - The operation of `search` command
  - `RSearch` - The operation of `rSearch` command
  - `Save` - The operation of `save` command
  - `Load` - The operation of `load` command
  - `Quit` - The operation of `quit` command

2.1C.2 UML Diagrams[2]

To better explain the relationships and interfaces between classes, we provide complete UML diagrams. Note that interface names are very descriptive, so we will not introduce their names specifically, unless they are specially designed.

The Model: Entities

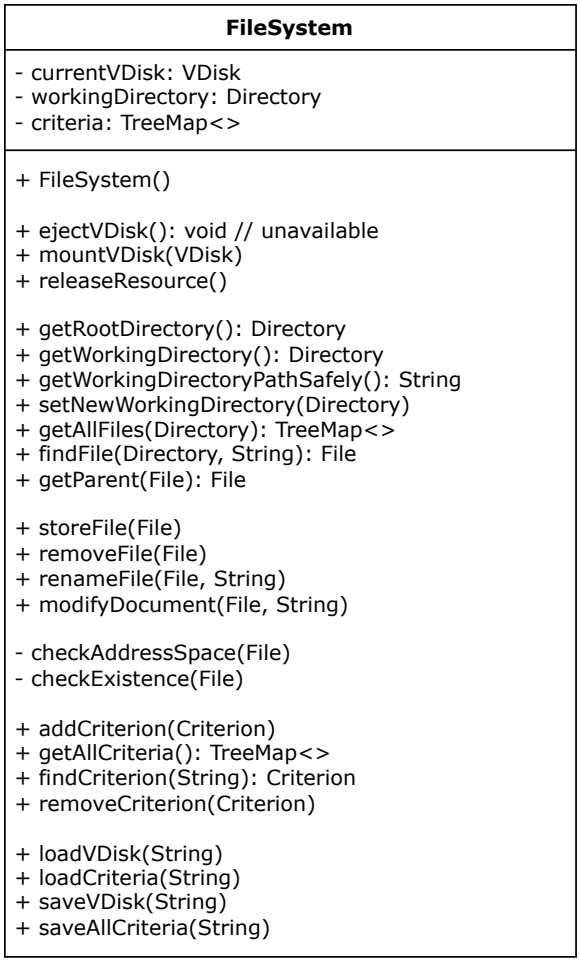


Note: For the meaning of `@ModelInternalUse` and the `__INTERNAL__` prefix, see 2.1B.6 and 2.1C.4.

[2] Due to space limitations, we have not included the parameter names for the functions. We ensure this will not affect your understanding.

(Continue)

The Model: The File System



Note:

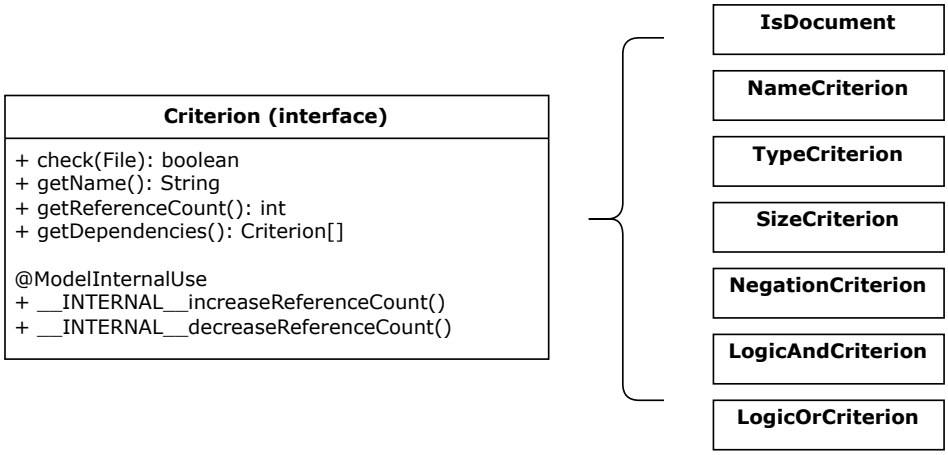
- For public methods (+), access is allowed for the Service part but not for the Controller and Console.
- For private methods (−) can only be accessed within the File System.
- In the following text, we will use *criteria list* to refer to the member `criteria`.

(Continue)

The Model: The Criterion Factory

CriterionFactory
+ Criterion createSimpleCriterion(String, String, String, String): Criterion + Criterion createNegationCriterion(String, Criterion, Criterion) + Criterion createBinaryCriterion(String, Criterion, String, Criterion)

The Model: Criterion



The Controller:

Controller
- fs: FileSystem - console: Console - operationFactory: OperationFactory - operationRecord: OperationRecord

+ Controller(FileSystem, Console) // called by the Application
+ boot()
+ work() // called by boot(), start the cycle

(Continue)

The Console

Console
- scanner: Scanner // input stream

+ Console() // called by the Application
+ boot()
+ getNextCommand(String) // called repeatedly by the Controller
+ printInformation(String) // called by the Controller
+ printErrorStream(String) // called by the Controller
 	
- parse(String): String[] // split the raw command into parts

The Service: The Operation Factory

OperationFactory
+ createOperation(FileSystem, OperationRecord, command: String[]): Operation

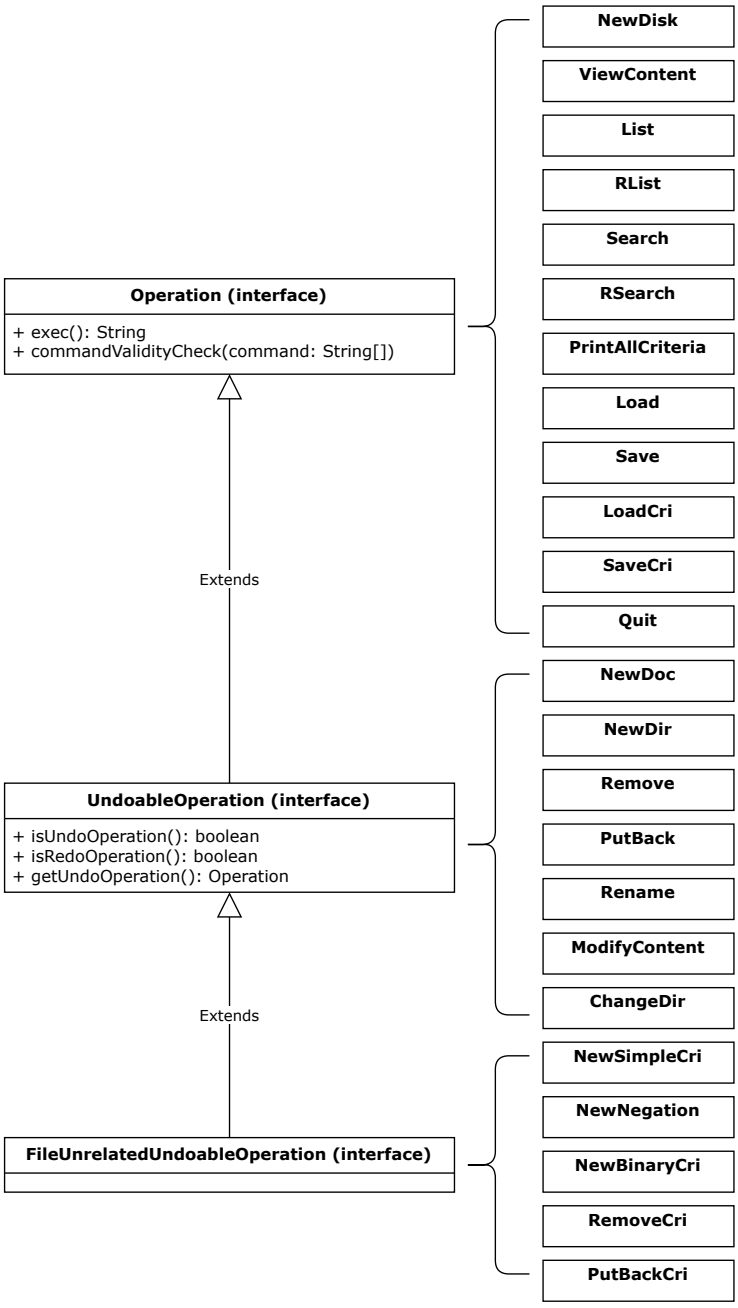
The Service: The Operation Record

OperationRecord
- undo: Stack<UndoableOperation> - redo: Stack<UndoableOperation>

+ OperationRecord() // called by the Controller
+ record(UndoableOperation)
+ popForUndo(): UndoableOperation
+ popForRedo(): UndoableOperation
 	
// Clear records
+ clearFileRelated()
+ clearFileUnrelated()
+ clearAll()

(Continue)

The Service: Operations



2.1C.3 Information Flow[3]

We have completed outlining our overall structure, but it still may not be easy to understand the information flow.

We will further refine the cycle introduced in section 2.1B.4. This time, we will specifically associate with the classes.

1. **`Console`** receives user's input.
2. **`Console`** parses the input, then returns it back to **`Controller`**.
3. **`Controller`** gives the parsed input to **`CriterionFactory`**.
4. **`CriterionFactory`** converts the input into an **`Operation`** object, and returns it back to **`Controller`**.
    - The Operation object corresponds to the command.
    - The Operation object includes the application's state information and the command parameter information.
5. **`Controller`** call the `exec()` method of the **`Operation`** object to execute the operation.
6. The **`Operation`** object would call the methods in the File System. Once its execution is finished, it returns the successful result back to **`Controller`**.
7. **`Controller`** provides the result to **`Console`**.
8. **`Console`** presents the result to the user.
9. Go back to 1.

[3] To make the whole process easier to understand, we have not emphasized the error handling.

2.1C.4 The `@ModelInternalUse` Annotation

This annotation indicates that a method should only be used within the Model part (i. e., the Entities and the File System), and should not be exposed to other parts.

The main reasons for using the `@ModelInternalUse` annotation include:

1. Some complex mechanisms, such as the actual disk storage logic, are not simulated. It is best not to expose such unsimulated things to other parts, and related methods should add this annotation and only be used internally. In addition, even with complete simulation, such methods should keep this annotation, refer to the second and third points.

2. Many methods are safely used within the Model part, but exposing it to other parts could lead to risks, such as unpredictable data changes. These methods should add this annotation.

3. Many methods are intended to be controlled and managed by the file system, and they are designed not to be exposed to other parts. These methods should add this annotation.

It is required to check the contexts and perform necessary error checking before using methods with this annotation.

2.1C.5 Implementation: Application Entry

We use a set of simple commands to start the application:

```
FileSystem fs = new FileSystem();
Console console = new Console();
Controller controller = new Controller(fs, console);
Application app = new Application(fs, console, controller);
```

2.1C.6 Implementation: Controller

We focus on how the Controller repeatedly executes the Cycle. From 2.1C.2, it is clear that this is the function of the method `work()`. Here we provide the general logic of `work()`, and important codes are highlighted:

```
private void work() {
    while (true) {
        try {
            String[] command = console.getNextCommand(...);

            Operation operation = operationFactory.createOperation(...);

            String result = operation.exec();
            console.printInformation(result);

            if (operation instanceof UndoableOperation) {
                operationRecord.record((UndoableOperation) operation);
            }
        } catch (...) {
            console.printErrorStream(e);
        }
    }
}
```

2.1C.7 Implementation: The Operation Factory

The Operation Factory is used to generate `Operation` objects. As mentioned, it needs to store the application's state information and the command's parameter information. The only method in this class, and the most important one, is `createOperation()`. Its general logic is:

```
public Operation createOperation(FileSystem fs,
                                 OperationRecord operationRecord,
                                 String[] command) {
      switch (command[0]) {
            case "newDisk":
                  return new NewDisk(fs, operationRecord, command);
            case "newDoc":
                  return new NewDoc(fs, command);
            ...
      }
}
```

2.1C.8 Implementation: The Undo and Redo Operations

To implement undo and redo features, we need some definitions. These definitions are all integrated into an enum type, called `OperationType`.

- Each undoable operation has three types:
  - **`REGULAR`**: Neither an undo nor a redo version of another operation.
  - **`UNDO`**: The undo version of another operation.
  - **`REDO`**: The redo version of another operation.
- There are state transitions between operations. If a type results from the state transition of another type, it is called the *inverse type* of that type.
  - The inverse type of `REGULAR` operation is of type `UNDO`.
  - The inverse type of `UNDO` operation is of type `REDO`.
  - The inverse type of `REDO` operation is of type `UNDO`.
- The enum type `OperationType` provides a `getInverseType()` method.

Then we need to find the inverse operation for each undoable operation. Since `UndoableOperation` interface provides a `getInverseOperation()` method, the following part is very simple.

Each time undo or redo command is called, the Operation Factory takes a similar approach, returns the *inverse operation* of the *latest* executed operation.

```
case "undo": return operationRecord.popForUndo()[4].getInverseOperation();
case "redo": return operationRecord.popForRedo().getInverseOperation();
```

Please see 2.1C.9 for more information.

[4] This method, and the following `popForRedo()` method, are used to get the latest executed operation for undo and redo. We believe the method names are descriptive.

2.1C.9 Implementation: Undoes and Redoes and the Operation Record

When an undoable operation is executed, the Controller will attempt to record it:

```
if (operation instanceof UndoableOperation) {
    operationRecord.record((UndoableOperation) operation);
}
```

So the operation is given to an Operation Record for further processing. The Operation Record is a structure used to record undoable operations. Basically, it maintains two stacks:

1. An **undo stack** for recording `REGULAR` and `REDO` operations.
2. A **redo stack** for recording `UNDO` operations.

Then it comes to the recording logic, which is handled by the `record()` method:

```
public void record(UndoableOperation o) {
    if (!o.isUndoOperation()) { // type is REGULAR or REDO
        undo.push(o);
        if (!o.isRedoOperation()) { // type is REGULAR
            redo.clear();
        }
    } else { // type is UNDO
        redo.push(o);
    }
}
```

The `record()` logic is reasonable. Let us illustrate it with an example, where the capital letters are the states of the application, and the arrows mean the execution of the operations:

$$A \xrightarrow{operation1} B \xrightarrow{operation2} C \xRightarrow{undo} B \xRightarrow{redo} C \xRightarrow{undo} B \xrightarrow{operation3} D \xrightarrow{cannot\ redo}$$

Additionally, let us briefly mention `UndoableOperation` and `FileUnrelatedUndoableOperation`. These interfaces are designed for certain special operations (like `Load` and `Save`, or `LoadCri` and `SaveCri`) where it is necessary to clear *some* operation records but not all. The separation between these two interfaces makes the application possible to clear part of the records.

2.1C.10 Implementation: Operation Execution and the Interfaces of the Model

In this section, we introduce the core of the Service part: the `exec()` method of the `Operation` object, which executes the operation.

Overall, the implementation details of each operation will be provided in section 2.1B. Now we only provide an example: the `NewDir` operation. Recall that `NewDir` corresponds to the `newDir` command, which creates a new directory.

```
@Override
public String exec() throws OperationCannotExecuteException {
    try {
        directory = new Directory(name, fs.getWorkingDirectory());
        fs.storeFile(directory);
        return ...
    } catch (ModelException e) {
        throw new OperationCannotExecuteException(e.getMessage());
    }
}
```

Note that `storeFile(File)` is a method of the File System. As part of the Service part, `NewDir` should not focus on its implementation.

2.1C.11 Implementation: File System Methods

In 2.1B.5, we listed all the File System interfaces, and,

In 2.1B.6, we introduced the implementation details of those interfaces and explained how we simulate a real file system as closely as possible.

We will not elaborate more here.

2.1C.12 Implementation: Read-Only Entities

In 2.1B.1 and 2.1B.3, we emphasized that for the Service part and other parts, entities including virtual disks, files, directories and documents are *read-only* and **appear abstracted by the File System**. These rules are set to better simulate a real file system.

These rules also indicates that, Entities cannot modify themselves or attempt to do tasks that are the responsibility of the File System. In 2.1B.3, we listed the tasks that only the File System can do.

We will not elaborate more here.

2.1C.13 Implementation: Other Implementations

In previous sections, we only introduced some of our core components. These components are innovative, but understanding them alone is not enough.

We list some other key components and provide brief introductions.

- Implementations of the Entities. In 2.1C.2, we showed the interfaces they provide to the File Systm as well as the other parts. In 2.1B.5, we showed the restrictions and requirements for their interaction with other parts, ensuring they seem abstracted by the File System, as explained in 2.1B.1.
- Implementations of the Criteria. The File System maintains a *criteria list* (see the notes of 2.1C.2) to record the saved criteria. Each criterion has a `check(File)` method to check if a file meets the criterion.
- Implementation of the Criterion Factory. In 2.1C.1, we explained the use of the Criterion Factory. In fact, it is the same as the Operation Factory. Given some parameters, it generates the corresponding `Criterion` object. In 2.1C.2, we showed the interfaces it provides.

2.1C.14 Exceptions

In this section, we list all the exceptions in our system. Most of these exceptions is in the Model part.

The Model

- Model external exception
  - `ModelException` - The general exception class of the Model part, and other parts shall catch this exception. All exceptions to the Model part should extend this exception.
- Model internal exceptions (all invisible to other parts, and all extends `ModelException`)
  - `CannotDeleteCriterionException` - Indicate that the criterion cannot be deleted. Generally, this is because other criteria depend on this criterion.
  - `CannotEditRootDirectoryException` - Indicate that the root directory cannot be edited.
  - `CannotInitializeFileException` - Indicate that the file cannot be initialized. This is most likely because the provided parameters do not meet the system requirements.
  - `CannotInitializeVDiskException` - Indicate that the virtual disk cannot be initialized. This is most likely because the provided parameters do not meet the system requirements.
  - `CriterionNotExistsException` - Indicate that the criterion with the specific name does not exist in the criterion list.
  - `DuplicateCriterionNameException` - Indicate that the criterion with the specific name does not exist in the criterion list.
  - `FileNotExistsException` - Indicate that the file does not exist in the directory.
  - `InvalidCriterionParametersException` - Indicate that the parameter is invalid for creating a criterion.
  - `LocalFileSystemException` - Indicate that the there are error(s) from the local file system, which prevents the virtual disk and criteria from being loaded or saved.
  - `NoMountedDiskOrWorkingDirectoryException` - Indicate that the command requires a mounted virtual disk and an available working directory, while there is not.
  - `VDiskOutOfSpaceException` - Indicate that the virtual disk is out of space to store the new file.
  - `WrongAddressSpaceException` - Indicate that the directory does not belong to the virtual disk. From a simulation perspective, this is an error related to address mapping.

(Continue)

The Service

- `InvalidCommandException` - Indicate that the user provided command is invalid. This is also the general exception in the Service part.
- `OperationCannotExecuteException` - Indicate that the operation cannot be executed due to various reasons.

The global exception

- `CVFS_Exception` - The most general exception class of this CVFS application. All other exceptions in the application should extend this exception.

Notes: It is too difficult to list which methods throw which exceptions here. Many methods can potentially throw various exceptions, especially those in the Model part. Luckily, because of the Enhanced MVC Model, each part does not need to know the specific exception types of others. In 2.1B.5, we provided a detailed explanation.

2.2 Implementation of Requirements

Unlike the previous chapter, this chapter focuses on the implementation of the requirements listed in the instruction document. Generally, we will first show the implementation of the `exec()` method for the `Operation` object corresponding to each command. If it calls methods from other classes, we will include those as well.

For each (compulsory and bonus) requirement, we described 1) whether it is implemented and, when yes, 2) how we implemented the requirement as well as 3) how we handled various error conditions.

[REQ1] `newDisk <diskSize>`

The requirement is implemented.

Implementation details

```
try {
    operationRecord.clearFileRelated();
    fs.mountVDisk(new VDisk(diskSize));
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The mountVDisk() method in the FileSystem class
public void mountVDisk(VDisk vDisk) {
    // ejectVDisk();
    currentVDisk = vDisk;
    workingDirectory = vDisk.__INTERNAL__getRootDirectory();
}
```

```
// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
try {
    if (Long.parseLong(command[1]) <= 0) {
        throw ...
    }
} catch (NumberFormatException e) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `CannotInitializeVDiskException`
- If the command is invalid (e.g., `diskSize` is not a positive and appropriate integer), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (which is rare here), an `OperationCannotExecuteException` will be thrown.

[REQ2] `newDoc <docName> <docType> <docContent>`

The requirement is implemented.

Implementation details

```
try {
    document = new Document(name, type, content, fs.getWorkingDirectory());
    fs.storeFile(document);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The storeFile(File) method in the FileSystem class
public void storeFile(File file) throws ... {
    checkAddressSpace(file);
    if (file.getSize() > currentVDisk.__INTERNAL__getFreeSpace()) {
        throw ...
    }
    Directory parent = (Directory)file.__INTERNAL__getParent();
    if (parent.__INTERNAL__existsName(file.getName())) {
        throw ...
    }
    parent.__INTERNAL__add(file);
}
```

```
// The commandValidityCheck() method
if (command.length != 4) {
    throw ...
}
if (command[1].length() > 10) {
    throw ...
}
```

(Continue REQ2)

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `WrongAddressSpaceException`
  - `VdiskOutOfSpaceException`
  - `DuplicatedFilenameException`
- If the command is invalid (e.g., `docName` is longer than 10 characters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., virtual disk out of space), an `OperationCannotExecuteException` will be thrown.

[REQ3] `newDir <dirName>`

The requirement is implemented.

Implementation details

```
try {
    directory = new Directory(name, fs.getWorkingDirectory());
    fs.storeFile(directory);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The storeFile(File) method in the FileSystem class
public void storeFile(File file) throws ... {
    checkAddressSpace(file);
    if (file.getSize() > currentVDisk.__INTERNAL__getFreeSpace()) {
        throw ...
    }
    Directory parent = (Directory)file.__INTERNAL__getParent();
    if (parent.__INTERNAL__existsName(file.getName())) {
        throw ...
    }
    parent.__INTERNAL__add(file);
}
```

```
// The commandValidityCheck() method
if (command.length != 4) {
    throw ...
}
if (command[1].length() > 10) {
    throw ...
}
```

(Continue REQ3)

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `WrongAddressSpaceException`
  - `VdiskOutOfSpaceException`
  - `DuplicatedFilenameException`
- If the command is invalid (e.g., `dirName` is longer than 10 characters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., virtual disk out of space), an `OperationCannotExecuteException` will be thrown.

[REQ4] `delete <fileName>`

The requirement is implemented.

Implementation details

```
try {
    if (file == null) {
        file = fs.findFile(fs.getWorkingDirectory(), name);
    }
    fs.removeFile(file);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The findFile(File, String) method in the FileSystem class
public File findFile(Directory directory, String name) throws ... {
    checkExistence(directory);
    if (!directory.__INTERNAL__existsName(name)) {
        throw ...
    }
    return directory.__INTERNAL__findFile(name);
}
```

```
// The removeFile(File) method in the FileSystem class
public void removeFile(File file) throws ... {
    checkAddressSpace(file);
    checkExistence(file);
    if ((file instanceof Directory) && ((Directory)file).isRootDirectory()) {
        throw ...
    }
    ((Directory)file.__INTERNAL__getParent()).__INTERNAL__delete(file);
}
```

```
// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
```

(Continue REQ4)

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `FileNotExistsException`
  - `WrongAddressSpaceException`
  - `CannotEditRootDirectoryException`
- If the command is invalid (e.g., the command includes more than 1 parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the file cannot be found in the working directory), an `OperationCannotExecuteException` will be thrown.

[REQ5] `rename <oldFileName> <newFileName>`

The requirement is implemented.

Implementation details

```
try {
    if (file == null) {
        file = fs.findFile(fs.getWorkingDirectory(), originalName);
    }
    fs.renameFile(file, newName);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The findFile(File, String) method in the FileSystem class
public File findFile(Directory directory, String name) throws ... {
    checkExistence(directory);
    if (!directory.__INTERNAL__existsName(name)) {
        throw ...
    }
    return directory.__INTERNAL__findFile(name);
}
```

```
// The renameFile(File, String) method in the FileSystem class
public void renameFile(File file, String newName) throws ... {
    checkAddressSpace(file);
    checkExistence(file);
    if ((file instanceof Directory) && ((Directory)file).isRootDirectory()) {
        throw ...
    }
    Directory parent = (Directory)file.__INTERNAL__getParent();
    if (parent.__INTERNAL__existsName(newName)) {
        throw ...
    }
    parent.__INTERNAL__delete(file);
    file.__INTERNAL__setName(newName);
    parent.__INTERNAL__add(file);
}
```

(Continue REQ5)

```
// The commandValidityCheck() method
if (command.length != 3) {
    throw ...
}
if (command[2].length() > 10) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `FileNotExistsException`
  - `WrongAddressSpaceException`
  - `FileNotExistsException`
  - `DuplicatedFilenameException`
  - `CannotEditRootDirectoryException`
- If the command is invalid (e.g., `newFileName` is longer than 10 characters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the file cannot be found in the working directory), an `OperationCannotExecuteException` will be thrown.

[REQ6] `changeDir <dirName>`

The requirement is implemented. We followed the first version of the instruction document, which would make our system more powerful.

Implementation details

```
try {
    if (originalDirectory == null) {
        originalDirectory = fs.getWorkingDirectory();
    }

    if (newDirectory == null) {
        Directory directoryNavigator = originalDirectory;
        String[] pathComponents = newDirectoryPath.split(":");
        int cur = 0;
        if (pathComponents[cur].equals("$")) {
            directoryNavigator = fs.getRootDirectory();
            cur++;
        }

        for (; cur < pathComponents.length; cur++) {
            if (pathComponents[cur].equals("..")) {
                directoryNavigator = (Directory)fs.getParent(directoryNavigator);
                continue;
            } else if (pathComponents[cur].equals(".")) {
                continue;
            }

          File nextDirectory =
                  fs.findFile(directoryNavigator, pathComponents[cur]);
            if (nextDirectory instanceof Directory) {
                directoryNavigator = (Directory)nextDirectory;
            } else {
                throw ...
            }
        }
        newDirectory = directoryNavigator;
    }

    fs.setNewWorkingDirectory(newDirectory);
    return ...
} catch (ModelException | InvalidCommandException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

(Continue REQ6)

```
// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}

if (command[1].length() > 10) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `FileNotExistsException`
  - `WrongAddressSpaceException`
- If the command is invalid (e.g., the command includes more than 1 parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the file cannot be found in the working directory), an `OperationCannotExecuteException` will be thrown.

[REQ7] `list`

The requirement is implemented.

Implementation details

```
try {
    StringBuilder result = new StringBuilder();
    Directory workingDirectory = fs.getWorkingDirectory();

    long size = workingDirectory.isRootDirectory() ?
      workingDirectory.getSize() :
       (workingDirectory.getSize() - Directory.EMPTY_NON_ROOT_DIRECTORY_SIZE);

    Collection<File> files = fs.getAllFiles(workingDirectory).values();

    if (!files.isEmpty()) {
        result.append(workingDirectory.getPath()).append(":\n");

        for (File file : files) {
            result.append(file.getFullname()).
                   append(" (").append(file.getSize()).append(")\n");
        }

        result.deleteCharAt(result.length() - 1);
        result.append("\n").
               append("Report: ").
               append(files.size()).
               append(" files, with total size ").append(size).append(".");
    } else {
        result.append("There are no files in the working directory.");
    }

    return result.toString();
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

(Continue REQ7)

```
// The commandValidityCheck() method
if (command.length != 1) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `NoMountedDiskOrWorkingDirectoryException`
  - `FileNotExistsException`
- If the command is invalid (e.g., the command includes a parameter), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the file cannot be found in the working directory), an `OperationCannotExecuteException` will be thrown.

[REQ8] r`List`

The requirement is implemented.

Implementation details

```
try {
    Directory workingDirectory = fs.getWorkingDirectory();

    long size = workingDirectory.isRootDirectory() ?
      workingDirectory.getSize() :
      (workingDirectory.getSize() - Directory.EMPTY_NON_ROOT_DIRECTORY_SIZE);

    if (fs.getAllFiles(workingDirectory).isEmpty()) {
        return "There are no files in the working directory.";
    }

    StringBuilder result = new StringBuilder();
    result.append(workingDirectory.getPath()).
            append("\n").
            append(getDirectoryTreeStructure(workingDirectory, ""));
    result.deleteCharAt(result.length() - 1);

    result.append("\n").
            append("Report: ").
            append(fileCount).
            append(" files, with total size ").append(size).append(".");
    return result.toString();
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

(Continue REQ8)

```
// The getDirectoryTreeStructure() method, notice the recursive tree structure
try {
    StringBuilder result = new StringBuilder();
    List<String> documentList = new ArrayList<>();
    List<Directory> subdirectoryList = new ArrayList<>();

    // Get all documents and subdirectories
    // Skipped here.

    // Print out all documents.
    for (int i = 0; i < documentList.size(); i++) {
        result.append(prefix).
                append(((i < documentList.size() - 1) ||
                        (!subdirectoryList.isEmpty()))
                        ? "├──"  // is not the last of this level
                        : "└──"). // is the last of this level
                append(documentList.get(i)).append("\n");
    }

    // Print out all directories (recursion).
    for (int i = 0; i < subdirectoryList.size(); i++) {
        Directory subdirectory = subdirectoryList.get(i);
        result.append(prefix).
                append((i < subdirectoryList.size() - 1)
                        ? "├──"  // is not the last of this level
                        : "└──"). // is the last of this level
                append(subdirectory.getFullname()).
                append(" (").append(subdirectory.getSize()).append(")\n");

        result.append(getDirectoryTreeStructure(subdirectory,
                (i < subdirectoryList.size() - 1)
                        ? (prefix + "│\t") // is not the last of this level
                        : (prefix + "\t"))); // is the last of this level
    }

    return result.toString();
} catch (ModelException e) {
    throw ...
}
```

(Continue REQ8)

```
// The commandValidityCheck() method
if (command.length != 1) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `NoMountedDiskOrWorkingDirectoryException`
  - `FileNotExistsException`
- If the command is invalid (e.g., the command includes a parameter), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the file cannot be found in the working directory), an `OperationCannotExecuteException` will be thrown.

[REQ9] `newSimpleCri <criName> <attrName> <op> <val>`

The requirement is implemented.

Implementation details

```
try {
    CriterionFactory criterionFactory = new CriterionFactory();
    criterion =
         criterionFactory.createSimpleCriterion(criName, attrName, op, val);
    fs.addCriterion(criterion);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}

// The createSimpleCriterion() method in the CiterionFactory class
try {
    checkCriName(criName);
    switch (attrType) {
        case "name": {
            if (!op.equals("contains")) { throw ... }
            if (val.length() <= 2 || !val.startsWith("\"") || !val.endsWith("\"")) {
                throw ...
            }
            return new NameCriterion(criName, val.substring(1, val.length() - 1));
        }
        case "type": {
            if (!op.equals("equals")) { throw ... }
            if (val.length() <= 2 || !val.startsWith("\"") || !val.endsWith("\"")) {
                throw ...
            }
            return new TypeCriterion(criName, val.substring(1, val.length() - 1));
        }
        case "size":
            return new SizeCriterion(criName, op, Long.parseLong(val));
        default:
            throw ...
    }
} catch (NumberFormatException e) {
    throw new InvalidCriterionParameterException("An integer expected.");
}
```

(Continue REQ9)

```
// The commandValidityCheck() method
if (command.length != 5) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `InvalidCriterionParameterException`
  - `DuplicatedCriterionNameException`
- If the command is invalid (e.g., the command includes less than 4 parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the given parameter(s) are invalid to initialize a criterion), an `OperationCannotExecuteException` will be thrown.

[REQ10] `IsDocument`

The requirement is implemented.

Implementation details

```
// The IsDocument criterion will be loaded to the criteria list when the
   File System is booted.
public FileSystem() {
    criteria = new TreeMap<>();
    criteria.put("IsDocument", new IsDocument());
}
```

Error conditions and error handling

None.

[REQ11] `newNegation <criName1> <criName2>`

The requirement is implemented.

Implementation details

```
try {
    CriterionFactory criterionFactory = new CriterionFactory();
    criterion =
      criterionFactory.createNegationCriterion(criName1, fs.findCriterion(criName2));
    fs.addCriterion(criterion);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}

// The createNegationCriterion() method in the CiterionFactory class
public Criterion createNegationCriterion(String criName,
                                         Criterion anotherCriterion) throws ...{
    checkCriName(criName);
    return new NegationCriterion(criName, anotherCriterion);
}

// The commandValidityCheck() method
if (command.length != 3) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `InvalidCriterionParameterException`
  - `CriterionNotExistsException`
  - `DuplicatedCriterionNameException`
- If the command is invalid (e.g., the command includes more than 2 parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the another criterion cannot be found in the criteria list), an `OperationCannotExecuteException` will be thrown.

[REQ11] `newBinaryCri <criName1> <criName3> <logicOp> <criName4>`

The requirement is implemented.

Implementation details

```
try {
    CriterionFactory criterionFactory = new CriterionFactory();
    criterion =
       criterionFactory.createBinaryCriterion(
                 criName1,
                 fs.findCriterion(criName3),
                 logicOp,
                 fs.findCriterion(criName4));
    fs.addCriterion(criterion);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}

// The createBinaryCriterion() method in the CiterionFactory class
public Criterion createBinaryCriterion(String criName,
                                       Criterion cri3,
                                       String logicOp,
                                       Criterion cri4) throws ... {
    checkCriName(criName);
    switch (logicOp) {
        case "&&":
            return new LogicAndCriterion(criName, cri3, cri4);
        case "||":
            return new LogicOrCriterion(criName, cri3, cri4);
        default:
            throw new InvalidCriterionParameterException(logicOp);
    }
}


// The commandValidityCheck() method
if (command.length != 5) {
    throw ...
}
```

(Continue REQ11)

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `InvalidCriterionParameterException`
  - `CriterionNotExistsException`
  - `DuplicatedCriterionNameException`
- If the command is invalid (e.g., the command includes less than 4 parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the another criterion cannot be found in the criteria list), an `OperationCannotExecuteException` will be thrown.

[REQ12] `printAllCriteria`

The requirement is implemented.

Implementation details

```
StringBuilder result = new StringBuilder();
Collection<Criterion> criteria = fs.getAllCriteria().values();

if (!criteria.isEmpty()) {
    result.append("There are ").
           append(criteria.size()).
           append(" criteria in the working directory: \n");
    for (Criterion criterion : criteria) {
        result.append(criterion.toString()).append("\n");
    }
    result.deleteCharAt(result.length() - 1);
} else {
    result.append("There are no criteria in the working directory.");
}
return result.toString();

// The commandValidityCheck() method
if (command.length != 1) {
    throw ...
}
```

Error conditions and error handling

- If the command is invalid (e.g., the command includes a parameter), an `InvalidCommandException` will be thrown.

[REQ13] `search <criName>`

The requirement is implemented.

Implementation details

```
try {
    StringBuilder result = new StringBuilder();
    Collection<File> files = fs.getAllFiles(fs.getWorkingDirectory()).values();
    Criterion criterion = fs.findCriterion(criName);

    if (!files.isEmpty()) {
        result.append("These file(s) satisfy the criterion:").
               append(criterion).append(":\n");
        for (File file : files) {
            if (criterion.check(file)) {
                result.append(file.getFullname()).
                       append(" (").append(file.getSize()).append(")\n");
                fileCount++;
                totalSize += file.getSize();
            }
        }
        result.deleteCharAt(result.length() - 1);
        result.append("\n").
                append("Report: ").
                append(fileCount).append(" files, with total size ").
                append(totalSize).append(".");
    } else {
        result.append("There are no files in the working directory.");
    }

    return result.toString();
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}

// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
```

(Continue REQ13)

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `FileNotExistsException`
  - `CriterionNotExistsException`
- If the command is invalid (e.g., the command includes a parameter), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the criterion cannot be found in the criteria list), an `OperationCannotExecuteException` will be thrown.

[REQ14] `rSearch <criName>`

The requirement is implemented.

Implementation details

```
try {
    Directory workingDirectory = fs.getWorkingDirectory();
    Criterion criterion = fs.findCriterion(criName);

    if (fs.getAllFiles(workingDirectory).isEmpty()) {
        return "There are no files in the working directory.";
    }

    StringBuilder result = new StringBuilder();
    result.append("These file(s) satisfy the criterion: ").
           append(criterion).append(":\n").
           append(recursiveSearch(workingDirectory, criterion));
    result.deleteCharAt(result.length() - 1);
    result.append("\n").
            append("Report: ").
            append(fileCount).
            append(" files, with total size ").append(totalSize).append(".");

    return result.toString();
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

(Continue REQ14)

```
// The recursiveSearch(Directory, Criterion) method
try {
    StringBuilder result = new StringBuilder();
    Collection<File> files = fs.getAllFiles(directory).values();

    for (File file : files) {
        if (criterion.check(file)) {
            result.append(file.getPath()).
                    append(" (").append(file.getSize()).append(")\n");
            fileCount++;
            totalSize += file.getSize();
        }

        if (file instanceof Directory) {
            result.append(recursiveSearch((Directory) file, criterion));
        }
    }
    return result.toString();
} catch (ModelException e) {
    throw new OperationCannotExecuteException("Some directories are invalid with
unknown reasons.");
}

// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `FileNotExistsException`
  - `CriterionNotExistsException`
- If the command is invalid (e.g., the command includes a parameter), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the criterion cannot be found in the criteria list), an `OperationCannotExecuteException` will be thrown.

[REQ15] `save <path>`

The requirement is implemented.

Implementation details

```
try {
    fs.saveVDisk(path);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The saveVDisk(String) method in the FileSystem class
public void saveVDisk(String path) throws ... {
    if (currentVDisk == null) {
        throw ...
    }
    try (ObjectOutputStream oos =
              new ObjectOutputStream(new FileOutputStream(path))) {
        oos.writeObject(currentVDisk);
    } catch (IOException e) {
        throw ...
    }
}
```

```
// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `NoMountedDiskOrWorkingDirectoryException`
  - `LocalFileSystemException`
- If the command is invalid (e.g., the command includes more than one parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the path is invalid), an `OperationCannotExecuteException` will be thrown.

[REQ16] `load <path>`

The requirement is implemented.

Implementation details

```
try {
    operationRecord.clearFileRelated();
    fs.loadVDisk(path);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The loadVDisk(String) method in the FileSystem class
public void loadVDisk(String path) throws ... {
    // ejectVDisk();
    try (ObjectInputStream ois =
             new ObjectInputStream(new FileInputStream(path))) {
        mountVDisk((VDisk)ois.readObject());
    } catch (ClassNotFoundException | IOException e) {
        throw ...
    }
}
```

```
// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
    - `LocalFileSystemException`
- If the command is invalid (e.g., the command includes more than one parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the path is invalid), an `OperationCannotExecuteException` will be thrown.

[REQ17] `quit`

The requirement is implemented.

Implementation details

```
operationRecord.clearAll();
fs.releaseResource();
System.exit(0); // safe exit
return null;

// The clearAll() method in the OperationRecord class
public void clearAll() {
    undo.clear(); // clear the undo stack
    redo.clear(); // clear the redo stack
}

// The releaseResource() method in the FileSystem class
public void releaseResource() {
    // ejectVDisk();
    criteria.clear();
}

// The commandValidityCheck() method
if (command.length != 1) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `LocalFileSystemException`
- If the command is invalid (e.g., the command includes a parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the path is invalid), an `OperationCannotExecuteException` will be thrown.

[BON1] `saveCri <path>`

The requirement is implemented.

Implementation details

```
try {
    fs.saveAllCriteria(path);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}

// The saveAllCriteria(String) method in the FileSystem class
public void saveAllCriteria(String path) throws ... {
    try (ObjectOutputStream oos =
              new ObjectOutputStream(new FileOutputStream(path))) {
        oos.writeObject(criteria);
    } catch (IOException e) {
        throw ...
    }
}

// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `NoMountedDiskOrWorkingDirectoryException`
  - `LocalFileSystemException`
- If the command is invalid (e.g., the command includes more than one parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the path is invalid), an `OperationCannotExecuteException` will be thrown.

[BON1] `loadCri <path>`

The requirement is implemented.

Implementation details

```
try {
    operationRecord.clearFileUnrelated();
    fs.loadCriteria(path);
    return ...
} catch (ModelException e) {
    throw new OperationCannotExecuteException(e.getMessage());
}
```

```
// The loadCriteria(String) method in the FileSystem class
public void loadCriteria(String path) throws ... {
    try (ObjectInputStream ois =
              new ObjectInputStream(new FileInputStream(path))) {
        criteria = (TreeMap<String, Criterion>) ois.readObject();
    } catch (ClassNotFoundException | IOException e) {
        throw new LocalFileSystemException(e.getMessage());
    }
}
```

```
// The commandValidityCheck() method
if (command.length != 2) {
    throw ...
}
```

Error conditions and error handling

- Errors in the Model part will throw `ModelException`.
  - `LocalFileSystemException`
- If the command is invalid (e.g., the command includes more than one parameters), an `InvalidCommandException` will be thrown.
- If the operation cannot be executed (e.g., the path is invalid), an `OperationCannotExecuteException` will be thrown.

[BON2] undo

The requirement is implemented.

Implementation details (see 2.3.8 and 2.3.9)

Implementation example (see 2.3.8 and 2.3.9)

```
// Undo a NewDoc operation
// The Service request for an undo operation
case "undo":
    return operationRecord.popForUndo().getInverseOperation();

// Get the inverse operation
@Override
public Operation getInverseOperation() {
    return new Remove(fs, document, OperationType.UNDO);
}
```

Error conditions and error handling

- If the command is invalid (e.g., there is no operation to undo), an `InvalidCommandException` will be thrown.

[BON2] `redo`

The requirement is implemented.

Implementation details (see 2.3.8 and 2.3.9)

Implementation example (see 2.3.8 and 2.3.9)

```
// Redo a Remove operation
// The Service request for a redo operation
case "redo":
    return operationRecord.popForRedo().getInverseOperation();

// Get the inverse operation
@Override
public Operation getInverseOperation() {
    return new PutBack(fs, file, type.getInverseType());
}
```

Error conditions and error handling

- If the command is invalid (e.g., there is no operation to undo), an `InvalidCommandException` will be thrown.

III. REFLECTION ON MY LEARNING-TO-LEARN EXPERIENCE

3.1 Learning-to-learn Experience

I learned a lot from this project. What I learned most about the design principles. The MVC Model was beneficial and helpful. Programs with MVC Models have clear structure, high cohesion, and low coupling between parts, which make them easy to understand and maintain.

Writing code and refactoring them was not easy. I refactored the whole project more than six times. Until the last ten days before the deadline, I finalized the Enhanced MVC Model and the complete *Fetch-Decode-Execute* Cycle of our system. Everything went smoothly from then. I think now I'm much more experienced, and I should be able to design excellent code structures more quickly and accurately in the future.

For the Java language and the algorithms, I've also learned a lot. For example,

- The regex expressions. I used it in the Console. It's really powerful and effective.
- The TreeMap container. It's a powerful data structure. I compared it with `std::map` in C++ and gained a deeper understanding of the standard libraries of modern programming languages.
- The recursion. I used a lot of recursions in `rSearch` and `rList`. One great challenge was printing the file tree. After finishing these recursive algorithms, I also gained a deeper understanding of the recursion.
- I used many Java syntax features that were new to me. For example, custom annotations, `java.io.Serializable`, and file input / output streams. After finishing this project, my understanding of the Java language also improved a lot.

In the end, this project also made me enjoy programming even more~!

YANG Xikun, Nov 19, 2024

Through this project, I have gained invaluable experience and insights, particularly in object-oriented programming (OOP).

One of the primary challenges I encountered was designing for modularity. The crucial question was how to structure interfaces, functions, and their hierarchies to ensure they are secure, polymorphic, and extensible. This required careful consideration and a thorough understanding of OOP principles such as:

- Encapsulation to protect data integrity
- Inheritance for code reuse and hierarchical relationships
- Polymorphism to enable flexible and maintainable code
- Interface segregation to maintain loose coupling(vital for system integrity)

Exception handling is another critical aspect of the project. As the last line of defense protecting program integrity, robust error handling is essential in sophisticated applications. The experience taught me that:

- Assumptions about error-free execution are absolutely unacceptable
- Different types of exceptions require specific handling strategies
- Proper error messages and logging are crucial for debugging
- Exception handling should be consistent across modules

Edge case testing proved instrumental in validating our implementation. To ensure compliance with both OOP principles and the Model-View-Controller (MVC) pattern, we:

- Developed comprehensive test cases covering boundary conditions
- Validated data flow between system components
- Tested inheritance hierarchies and polymorphic behavior
- Verified exception handling mechanisms
- Ensured proper separation of concerns according to MVC

This project has deepened my understanding of software architecture principles and reinforced the importance of thoughtful design in creating maintainable, robust applications.

YANG Jinkun, Nov 20, 2024

In the group project, I am responsible for constructing the JUnit test to check whether our code can run effectively and satisfy all the requirement. To make sure the test have covered all the operations, I need to fully understand the requirements — what exactly the CVFS is and the basic logic behind. To further achieve this, I have gone through the document repeatedly and discussed a lot with our team members. Also, to ensure the test is effective and valid, I've been working with the designer of this structure, our team member-Frank Yang to discuss the code and the implementations behind. During my construction of Junit test, I have met some problems. When I am checking the code, I felt that the implementation is wrong, or I just cannot understand. But I could not just simply change the code, so I have to discuss the code with my other team members and try our best to reach an agreement. Also, to try to better understand. Also, I am not really familiar with the JUnit test, so I have to go through the slides and the online tutorials about the Junit test.

By completing the project with my excellent group members, I have learned that collaborate is the key, especially when you work as the test engineer, you must have a thorough and clear understanding of the code design and structure, and that require you to have a good and frequent collaboration with the architecture designer in the team. So I think the collaboration and self-motivation is the key.

REN Yixiao, Nov 20, 2024

3.2 Plan to Improve Self-Learning Approaches

This project improved my self-learning skills. It involved not only using the Java Programming Language, but also configuring the IDE and Git, among other tools. Of course, I understand more about self-learning.

- Create needs. If I just memorize knowledge rigidly, I will soon forget it. I must create a learning environment, such as doing a project or just a small program, and learn new things. Within that, I will truly grasp the new knowledge.
- Learn to read the documentation. Many things require reading the documentation, such as `java.io.Serializable`. Documentation can be very complex and hard to understand, but it is the gold standard. Just like if you want to learn C++ deeply, you need to check the ISO documentation, learning any kind of programming requires reading the standard documentation.

YANG Xikun, Nov 20, 2024

To further improve my self-learning approaches, I think one of the most important things is develop the ability to gather useful information for you to learn. In the future, I should improve my ability to search for the valuable information whenever encounter some unfamiliar topic. Also, learning only about theory is not enough, I should also put more time in the practice, because knowing the theory doesn't mean you can apply the knowledge. Before the group project, I thought I know well about JUnit test, but only when I apply these knowledge in this project can I really understand.

After all, it is a really good experience to work with my teammates and work for the same goal.

REN Yixiao, Nov 20, 2024