Project Report

COMP2021 Object-Oriented Programming (Fall 2024)
Group 28

Members and contribution percentages:
WANG Yuqi: 33.3%
CUI Mingyue: 33.3%
CHEN Yangxuan: 33.3%

1. Introduction

This report describes the design and implementation of the Comp Virtual File System (CVFS) by group 28. It is part of the course COMP2021 Object-Oriented Programming at PolyU.

2. The Comp Virtual File System (CVFS)

2.1. Overall Design

In this project, we follow the Model-View-Controller (MVC) design pattern.

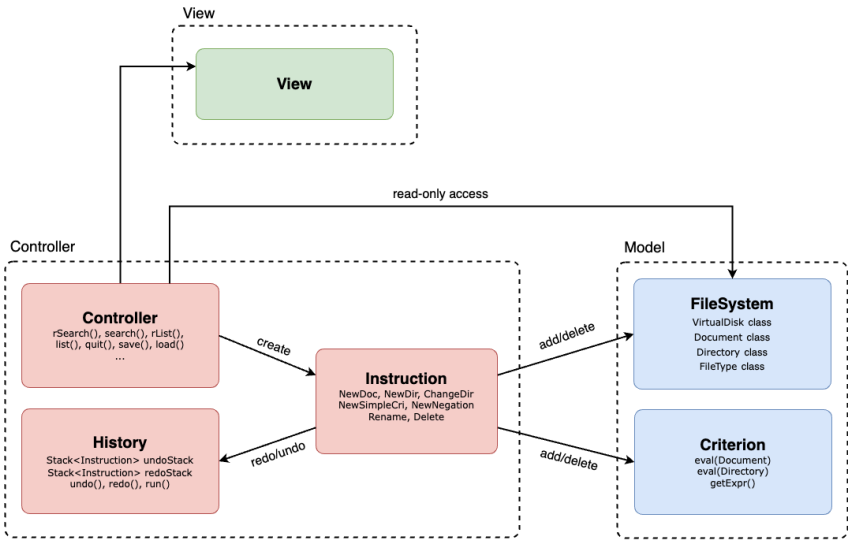

Figure 1: The overall design of the CVFS

The `Controller` parses user input and maintains the current state of the CVFS. It serves as a mediator between the `Model` and `View` components, sending parsed instructions to the `Model` and forwarding the results to the `View` for display. As illustrated in Figure 1, the Controller *creates* the a coresponding `Instruction` object based on the user input, which then modifies the `Model`.

The `Model` maintains a single instance of the `VirtualDisk` along with its sub-components (e.g., `Document`, `Directory`, `Criterion`). The `Model` operates independently and does not directly interact with the other two components. It receives instructions solely from the `Controller` and updates its internal state accordingly.

The `View` similarly maintains a one-way data flow, receiving but not transmitting data. It implements various methods to display instruction results in different formats.

2.2. Controller

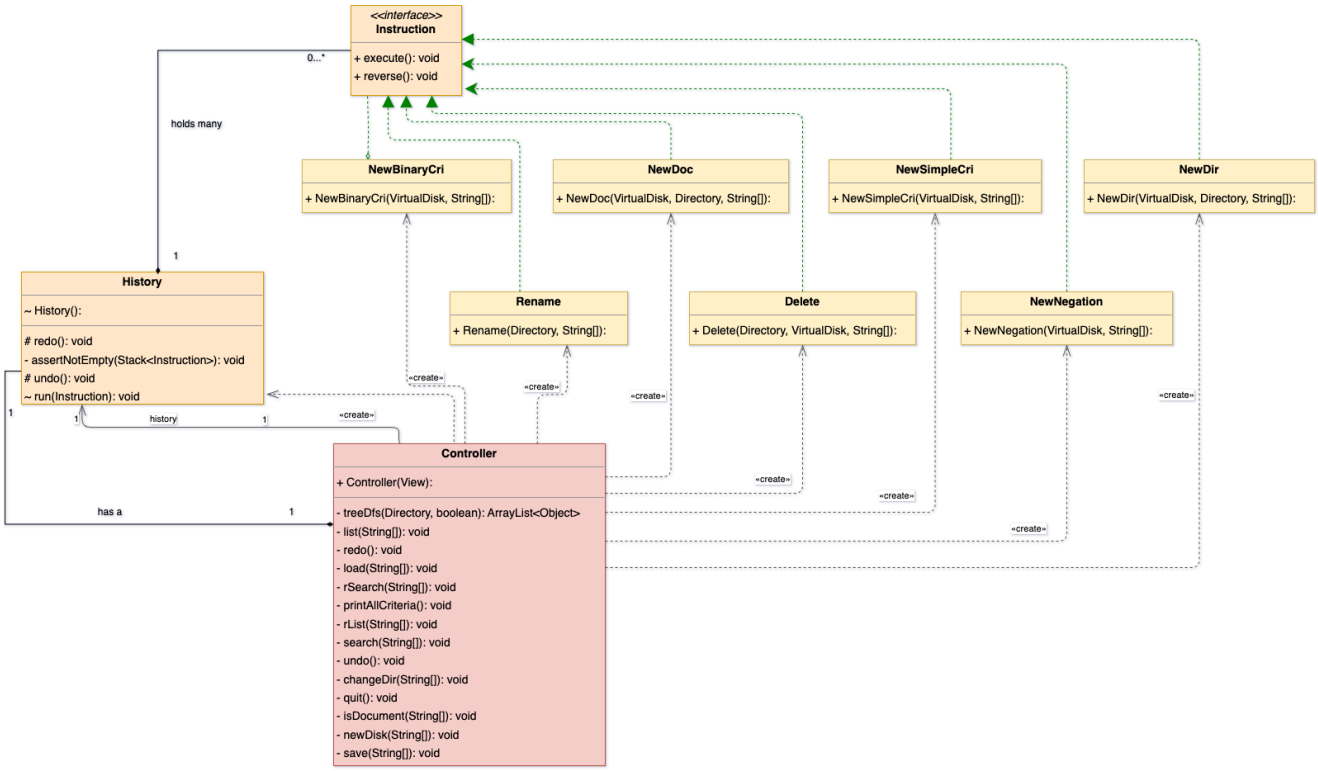

Figure 2: Close-up diagram of the `Controller` package

Figure 2 provides a close-up look of the `Controller` package. To better understand its construct, three things must be understood: **1) Irreversible Instructions**, **2) Reversible Instructions** **3) the History**.

2.2.1. Irreversible Instructions

Irreversible instructions (colored red) refer to user commands that are executed just once (e.g., `rSearch`, `rList`, `printAllCriteria`). In general, these commands do not make lasting change on the `Model`, hence does not make sense to be undone or redone. Therefore, for these types of instructions, we only need to create a method for each, and call them when needed.

2.2.2. Reversible Instructions

Reversible instructions (colored yellow) refers to user commands that **can** be redone or undone (e.g., `newDoc`, `newDir`, `newSimpleCri`). Unlike irreversible instructions, these aren't directly executed through a single method call. Instead, a corresponding `Object` that `implements` the `Instruction` interface is created. The motivation for such design is that, to reverse the effect of an instructions, we need to store the instruction itself along with its related information. We also need to encapsulate two actions into these instructions, namely `inverse()` and `reverse()`, which executes the instruction forward and in reverse, for redo and undo. This leads us to the following `Instruction` interface:

```
interface Instruction {
    // execute():  executes the instruction normally. For run() & redo()
    // reverse():  executes the inverse of the instruction. For undo().

    void execute() throws CVFSException;
    void reverse() throws CVFSException;
}
```

2.2.3. History

The `History` class (colored orange) serves the purpose of redoing and undoing instructions. It contains three main methods `run()`, `undo()`, and `redo()`. The `run()` instruction is called when an instruction is first created; it executes the instruction and pushes it into the `undoStack`. The `redo()` and `undo()`, as their name suggests,

```
public class History {
    private final Stack<Instruction> undoStack;
    private final Stack<Instruction> redoStack;

    void run(Instruction inst) throws CVFSException;
    void redo() throws CVFSException;
    void undo() throws CVFSException;
}
```

2.2.4. FULL PROCEDURE

- *Step 1: Converting raw string input into a command and a list of arguments*
- *Step 2: Execute commands the two types of instructions*
  - **For reversible instructions (e.g., `newDoc`, `newDir`, `newSimpleCri`)**:
    - Creates an `Instruction` object for the command
    - Executes and stores the instruction in history for future undo/redo operations
  - **For non-reversible instructions (e.g., `rSearch`, `rList`, `printAllCriteria`)**:
    - Directly calls the corresponding method
    - Sends results to view for display
  - **Examples of both**:
    - `newDoc`:
      1. `Controller` creates `NewDoc` object, which `implements Instruction`.
      2. `Controller` calls the runs the instruction, which interacts with `Model`
      3. `Controller` puts the `NewDoc` object into `History` for undo / redo
    - `research`:
      1. `Controller` calls the `rSearch()` method
      2. `rSearch()` performs recursive directory search on the given directory
      3. Interacts with the `Model` by retrieving directories via `model.Directory.getDir`
- *Step 3: Exception handling*

  1. Exception (e.g., invalid file names) occurred somewhere (possibly deep) within *Step 2*
  2. Checked exception propagates up the call stack until it reaches back to the `Controller`
  3. `Controller` catch the exceptions and send it to `View` to notify the user

2.3. Model

Figure 3 illustrates the construct of the `Model` package, which contains the components heart to the CVFS. The `Model` represents the static state of a single `VirtualDisk` instance, acting as the central repository that orchestrates interactions between CVFS model components such as `Criterion`, `Directory`, and `Document`.

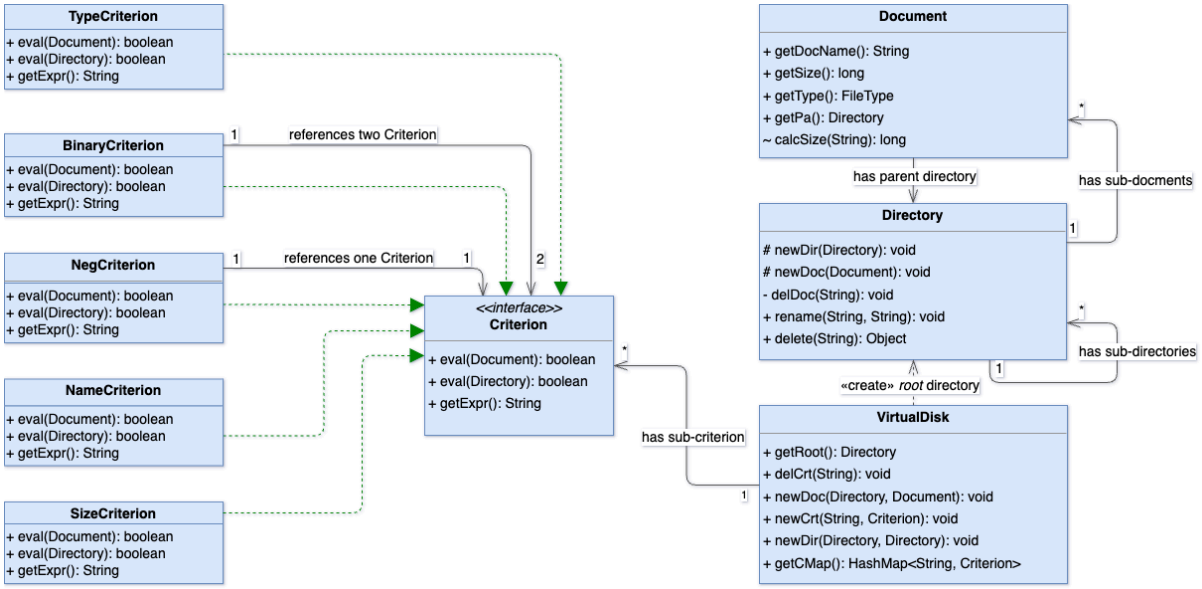

Figure 3: Close-up diagram of the `Model` package

2.3.1. Virtual Disk

The `VirtualDisk` maintains two primary elements: 1) a root `Directory` object and 2) a collection of `Criterion` objects. The root Directory serves as the entry point to the file hierarchy, containing multiple sub-Directory and sub-Document objects. These sub-directories can themselves contain any number of additional sub-Directory and sub-Document objects, forming a tree-like hierarchical structure. The collection of criterions is declared within `VirtualDisk` to enable serialization alongside the disk instance for local machine storage.

2.3.2. Directory

The `Directory` class contains **two sets of pointers**. The first set points to sub-directories, while the second set points to sub-documents. In implementation, they are basically two `HashMap<>` objects that map filenames to their corresponding file objects (i.e., `Directory` or `Document`):

```
private final HashMap<String, Directory> dirMap;
private final HashMap<String, Document> docMap;
```

2.3.3. Document

The `Document` class maintains a straightforward structure as a singleton holding document content. It incorporates a more sophisticated error handling mechanism due to the numerous edge cases associated with `newDoc` operations, details of which will be discussed in later sections.

2.3.4. Criterions

The system defines five distinct criterions: `TypeCriterion`, `NameCriterion`, `SizeCriterion`, `BinaryCriterion`, and `NegCriterion`. Each implements the `Criterion` interface:

```
public interface Criterion {
    boolean eval (Document doc);
    boolean eval (Directory dir);
    String getCrtName();
    String getExpr();
}
```

The unification of these criterions under a single `Criterion` interface offers two significant advantages. First, it enables *Runtime Polymorphism*, abstracting away the implementation differences between the five criterion types. Second, it facilitates *Compile-time Polymorphism* through enforced overloaded `eval()` methods. The following example demonstrates these benefits:

Example 1:

```
private boolean filter(Document doc, Criterion crt) {
    if (file instanceof Document) {
        Document doc = (Document) file;
        return crt.eval(doc);
    } else {
        Directory dir = (Directory) file;
        return crt.eval(dir));
    }
}
```

This example showcases how *Runtime Polymorphism* allows the creation of a method that evaluates any criterion type against a `file` object without concern for the specific criterion implementation. Additionally, *Compile-time Polymorphism* enables the use of the same `eval()` method for both `Document` and `Directory` objects, streamlining the code logic.

2.4. View

The `View` package consists solely of the `View` class, which implements various methods for displaying CVFS output. A key example is the `displayFileTree` method, which processes an `ArrayList` containing the file hierarchy (typically generated from `rList` or `list` operations).

```
>>> load test/testDisk.ser
load: loaded serialized disk from test/testDisk.ser

>>> rList
.
├─root1/     size: 1110
|   └─home/      size: 1070
|       └─tony/      size: 1030
|           ├─Downloads/ size: 40
|           ├─Document/  size: 382
|           |   ├─homeWifi.txt size: 56
|           |   ├─router.txt size: 102
|           |   └─homeDish.txt size: 184
|           └─Code/      size: 568
|               ├─Model.java size: 160
|               ├─Controller.java size: 192
|               ├─cvfs.java  size: 118
|               └─info.txt   size: 58
├─root2/     size: 40
└─root3/     size: 40
-------------------------
Total Files     Total Size
7               1190

>>> |
```

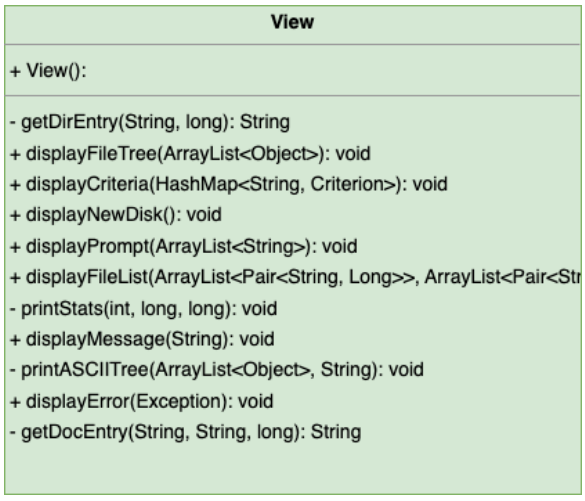

Figure 4: (Left) example output of the `displayFileTree` method
(Right) UML diagram of the single `View` class

2.5. Error Handling

Our project implements a streamlined approach to error handling by combining the Fail-Fast design pattern with Exception Propagation. In system design, Fail-Fast represents a strategy

where errors are detected and reported immediately. In the context of CVFS, Exception Propagation refers to the practice of forwarding exceptions up the call stack to the Controller, rather than handling them at where they originated.

In terms of implementaiton, a collection of custom assert (utility) methods are used. These methods can be invoked from **anywhere** within the CVFS codebase to check for errors. When conditions are violated, the assertion fails and the assertion methods throws a custom checked exception. For all CVFS components except the top-level Controller, we deliberately omit try-catch blocks and include only throws clauses in method signatures (unless the exception is unexpected, then we `try-catch` it and throws a `RuntimeException`).

This design enables automatic exception propagation up the call stack and handled at just one location. It effectively eliminats the need for local exception handling throughout the CVFS. To provide a more concrete understanding, consider the following example:

1. User enters command: `newDir home`
2. Method call sequence:

$$\text{Controller} \xrightarrow{\text{creates}} \text{NewDir object} \xrightarrow{\text{calls}} \text{disk.newDir(parentDir, curDir)}$$

3. Assertion method `assertEnoughSpace` called within `newDir`
4. Assertion failure triggers `FullDiskException`
5. Exception propagates back up the call stack:

$$\text{Controller} \xleftarrow{\substack{\text{throws}\\\text{exception}}} \text{NewDir object} \xleftarrow{\substack{\text{throws}\\\text{exception}}} \text{disk.newDir(parentDir, curDir)}$$

6. Exception is ultimately caught by `Controller`'s `try-catch` block
7. `Controller` extracts the error message and forwards it to `View`

3. Requirements Implementation

A clear understanding of Section 2.5 is essential before proceeding to this next section!

[REQ1] newDisk diskSize (Implemented ✅)

Implementation Detail

```
private void newDisk(String[] args) throws
    InvalidCommandException,
    InvalidValueException
{
    assertArgumentCount("newDisk", args, 1); // <-- error handling
    assertNumber("disk size", args[0]); // <-- error handling
    assertLimit("disk size", args[0], MAX_DISK_SIZE); // <-- error handling
    curDisk = new VirtualDisk(toNumber(args[0]));
    curDir = curDisk.getRoot();
    history.clear();
    view.displayNewDisk();
    cwd.clear();
}
```

The `newDisk` command is an **irreversible** instruction. Therefore, executing `newDisk` is simply a method call to `Controller.newDisk`. To create a new disk:

1. assign the `curDisk` pointer to a newly created `VirtualDisk` object with the specified size
2. update the current working directory (`curDir` and `cwd`) to the new root
3. clear the `history` as it pertains only to the previous disk
4. notify the user of successful disk creation via `View.displayNewDisk()`

Error Conditions

Error conditions:

1. User input has argument count $\neq 1$
2. User input `diskSize` is **not** numeric
3. User input `diskSize` is too large (e.g., Integer Overflow / Exceed Memory Limit)

Error Handling

These three potential error conditions are checked using the `assert` utility functions (discussed in Section 2.5):

1. `assertArgumentCount`: checks that there is exactly one argument
2. `assertNumber`: ensures `diskSize` is numeric
3. `assertLimit`: checks if `diskSize` $\leq$ `MAX_DISK_SIZE` (10^9)

Violations trigger either `InvalidCommandException` (argument count) or `InvalidValueException` (numeric validity, size limit), which propagate to the `Controller` for handling.

[REQ2] newDoc docName docType docContent (Implemented ✅)

Implementation Detail

```
class NewDoc implements Instruction {
    private final String name, content, type;
    private final VirtualDisk curDisk;
    private final Directory curDir;

    public NewDoc(VirtualDisk curDisk, Directory curDir, String[] args) throws
        InvalidCommandException
    {
        assertArgumentCount("newDoc", args, 3);
        this.name = args[0]; // filename
        this.type = args[1]; // type
        this.content = args[2]; // content
        this.curDisk = curDisk;
        this.curDir = curDir;
    }

    @Override
    public void execute() throws
      FileExistsException,
      InvalidFileException,
      FullDiskException,
      InvalidTypeException
```

```
    {
        Document doc = new Document(curDir, name, content, type);
        curDisk.newDoc(curDir, doc);
    }

    @Override
    public void reverse() throws FileNotFoundException {
        curDir.delete(name);
    }
}
```

The `newDoc` command is a **reversible** instruction. Therefore, each `newDoc` command creates a new `NewDoc` object that implements the `Instruction` interface (see Section 2.2.2). The execution flow is as follows:

1. `Controller` creates a new `NewDoc` object.
2. `NewDoc` object is passed to `History.run()`, which calls `execute()`
3. `execute()` creates a new `Document` object, then adds it to `curDisk` at `curDir`
4. The instruction object is stored in the `history.undoStack` for potential reversal

Error Conditions

1. User input argument count $\neq 3$ (i.e., docName, docType, docContent)
2. User input filename > 10 characters *or* contains special characters
3. User input file type is not one of "txt, java, css, html"
4. Document size exceeds remaining disk size
5. Document / Directory of same name already exists.

Error Handling

1. Argument count checked within the `NewDoc` constructor via `assertArgumentCount`

```
public NewDoc(...) ... {
    assertArgumentCount("newDoc", args, 3); // <-- error handling
    ...
}
```

2. Filename validity checked within the `Document` constructor via `assertValidFileName`

```
public Document(Directory pDir, String docName, String content, String type) throws
    InvalidTypeException,
    InvalidFileException
{
    assertValidFileName("document name", docName); // <-- error handling
    this.docName = docName;
    this.type = FileType.getFileType(type);
    this.size = calcSize(content);
    this.pDir = pDir;
}
```

3. File type validity also checked within the `Document` constructor via:

```
   this.type = FileType.getFileType(type);
```

The `getFileType` method throws `InvalidTypeException` if the given type does not match any of the predefined file types (i.e., txt, java, html, css):

```
public static FileType getFileType(String type) throws
    InvalidTypeException{
    switch (type) {
        case "txt" -> return FileType.TXT;
        case "java" -> return FileType.JAVA;
        case "html" -> return FileType.HTML;
        case "css" -> eturn FileType.CSS;
        default -> throw new InvalidTypeException(...);
      }
  }
```

4. Disk space availability is checked within `curDisk.newDoc(curDir, doc)`:

```
    public void newDoc(Directory baseDir, Document doc) throws
        FileExistsException,
        FullDiskException
    {
        assertEnoughSpace(doc); // <-- error handling
        baseDir.newDoc(doc);
    }
```

In which, the `assertEnoughSpace` method in `curDisk.newDoc` throws `FullDiskException` if the calculated size of the new document exceeds that of the free space.

5. Filename duplciation is checked within `baseDir.newDoc` of `curDisk.newDoc`

```
  protected void newDoc (Document doc) throws FileExistsException{
      String key = doc.getDocName();
      assertUniqueName(key); // <-- error handling
      docMap.put(key, doc);
      updateSize(doc.getSize());
      updateCnt(1);
  }
```

In which, the `assertUniqueName` method in `curDisk.newDoc` throws `FileExistsException` when a `Document` or `Directory` of the same name already exists.

[REQ3] newDir dirName (Implemented ✅)

Implementation Detail

```
class NewDir implements Instruction {
    private final String name;
    private final VirtualDisk curDisk;
    private final Directory curDir;

    public NewDir(VirtualDisk curDisk, Directory curDir, String[] args) throws
        InvalidCommandException
    {
        assertArgumentCount("newDir", args, 1); // <-- error handling
        this.name = args[0];
        this.curDisk = curDisk;
```

```
        this.curDir = curDir;
    }

    @Override
    public void execute() throws
        InvalidFileException,
        FullDiskException,
        FileExistsException
    {
        Directory dir = new Directory(name, curDir);
        curDisk.newDir(curDir, dir);
    }

    @Override
    public void reverse() throws FileNotFoundException {
        curDir.delete(name);
    }
}
```

The `newDir` implementation is very similar to that of `newDoc`. Both are **reversible** instruction. Therefore, `newDir` command also creates its own `NewDir` object that too implements the `Instruction` interface (see Section 2.2.2). The execution flow as follows:

1. `Controller` creates new `NewDir` object
2. `NewDir` object passed to `History.run()`, which calls `execute()`
3. `execute()` creates new `Directory` object, then adds it to `curDisk` at `curDir`
4. The instruction object is stored in the `history.undoStack` for potential reversal

Error Conditions

1. User input argument count $\neq 1$ (i.e., docName)
2. User input filename > 10 characters *or* contains special characters
3. Directory size exceeds remaining disk size (i.e., disk space < 40)
4. Document / Directory of same name already exists.

Error Handling

Error handling mechanism for `newDir` is nearly identical to `newDoc`, so they will be skimmed over. For more detail, check *Error Handling* section of [REQ2]):

1. Argument count checked within the `NewDir` constructor via `assertArgumentCount`
2. Filename validity checked within the `Directory` constructor via `assertValidFileName`

```
public Directory (String dirName, Directory pDir) throws InvalidFileException{
    assertValidFileName("directory name", dirName); // <-- error handling
    dirMap = new HashMap<>();
    docMap = new HashMap<>();
    this.dirName = dirName;
    this.pDir = pDir;
    this.size = (pDir == null) ? 0 : DIRECTORY_BASE_SIZE;
    this.cnt = 1;
}
```

3. Disk space availiability checked via `curDisk.newDir`

```
    public void newDir(Directory baseDir, Directory dir) throws
        FileExistsException,
        FullDiskException
    {
        assertEnoughSpace(dir); // <-- error handling
        baseDir.newDir(dir);
    }
```

4. Filename duplication checked within `baseDir.newDir` of the above code

```
    protected void newDir (Directory dir) throws FileExistsException{
        String key = dir.getDirName();
        assertUniqueName(key); // <-- error handling
        dirMap.put(key, dir);
        updateSize(dir.getSize());
    }
```

[REQ4] delete fileName (Implemented ✅)

Implementation Detail

```
class Delete implements Instruction {
    private final String name;
    private final Directory curDir;
    private final VirtualDisk curDisk;
    private Object file;

    public Delete(Directory curDir, VirtualDisk curDisk, String[] args) throws
        InvalidCommandException
    {
        assertArgumentCount("delete", args, 1); // <-- error handling
        this.name = args[0];
        this.curDir = curDir;
        this.curDisk = curDisk;
    }

    @Override
    public void execute() throws FileNotFoundException {
        file = curDir.delete(name); // <-- error handling in here
    }

    @Override
    public void reverse() {
        try {
            if (file instanceof Directory) curDisk.newDir(curDir, (Directory) file);
            else curDisk.newDoc(curDir, (Document) file);
        } catch (FullDiskException | FileExistsException e) {
            throw new RuntimeException("impossible error: " + e);
        }
    }
}
```

Since the `delete` command is a **reversible** instruction, it creates a dedicated `Delete` object that implements the `Instruction` interface. Here's the execution flow:

1. `Controller` creates `Delete` object
2. `Delete` object passed to `History.run()`

3. `History.run()` calls `execute()` method of `Delete`
4. `execute()` calls `delete` method of the current directory (i.e., `curDir.delete(name);`
5. `curDir.delete` returns the deleted file object
6. `Delete` object cache the deleted file as `Object file`
7. `Delete` object is pushed onto `history.undoStack` for potential reversal

The following is the implementation of `curDir.delete(String)`:

```
public Object delete (String key) throws FileNotFoundException {
    if (dirMap.containsKey(key)) {
        Directory dir = getDir(key);
        delDir(key);
        return dir;
    }
    if (docMap.containsKey(key)) {
        Document doc = getDoc(key);
        delDoc(key);
        return doc;
    }
    throw new FileNotFoundException(...);
}
```

When attempting to undo `delete`:

1. `reverse()` method of `Delete` checks the previously cached `file`
2. If `file` is `Directory` we call the `newDir` method
3. If `file` is `Document` we call the `newDoc` method

Error Conditions

1. User input argument count $\neq 1$ (i.e., fileName)
2. File of the specified name not found

Seemingly likely but **impossible** error conditions:

Name Collision During Undo While restoring a deleted file through undo, it might appear possible for a naming conflict to occur with a file created after the deletion. This cannot happen because undo operations proceed in reverse chronological order - any newer file with the same name would be removed first before reaching the deletion recovery.

Insufficient Space During Undo It may seem possible to run out of disk space while restoring a deleted file. However, since undo reverses all operations after the deletion first, any files created after the deletion would be removed before the recovery attempt. This guarantees that at least as much space is available as when the file was originally deleted.

Missing Parent Directory It might appear that the parent directory of a deleted file could be missing when trying to restore it. However, if the parent directory was deleted after the file deletion, that directory deletion would be undone first, ensuring the parent directory exists when recovering the file.

In short, the first-in last-out (FILO) nature of the undo mechanism inherently prevents these scenarios by restoring the exact system state that existed at the time of deletion.

Error Handling

1. Argument count checked via `assertArgumentCount` in the constructor of the `Delete` object
2. File existence is checked within `curDir.delete`. If neither `dirMap` nor `docMap` contains the specified filename, then we throw `FileNotFoundException`

[REQ5] rename oldFileName newFileName (Implemented ✅)

Implementation Detail

```
class Rename implements Instruction {
    private final String oldName, newName;
    private final Directory curDir;

    public Rename(Directory curDir, String[] args) throws InvalidCommandException{
        assertArgumentCount("rename", args, 2); // <-- error handling
        this.oldName = args[0];
        this.newName = args[1];
        this.curDir = curDir;
    }

    @Override
    public void execute() throws
        FileNotFoundException,
        FileExistsException,
        InvalidFileException
    {
        curDir.rename(oldName, newName); // <-- error handling inside
    }

    @Override
    public void reverse() {
        try {
            curDir.rename(newName, oldName); // <-- no error can occurr
        } catch (
          FileNotFoundException |
          FileExistsException |
          InvalidFileException e) {
            throw new RuntimeException("Impossible error: " + e);
        }
    }
}
```

Since `rename` is a **reversible** instruction, a `Rename` object that implements the `Instruction` interface is created. Here's the execution flow:

1. `Controller` creates `Rename` object
2. `Rename` object passed to `History.run()`
3. `History.run()` calls `execute()` of `Rename`
4. `execute()` calls the `rename` method of the current directory object `curDir`

Error Conditions

1. User input argument count $\neq 2$ (i.e., oldName, newName)
2. The `newName` does not satisfy naming criterion (≤ 10 alphanumeric characters)
3. There already exists a file named `newName`

4. There does not exist a file named `oldName`

> Seemingly likely but **impossible** error conditions:
>
> **Name Collision During Undo** While restoring the original name through undo, it might appear possible for a naming conflict to occur with a newly created file. However, this cannot happen because undo operations proceed in reverse chronological order - any newer file with the conflicting name would be removed first before reaching the `rename` undo.
>
> **File Not Found During Undo** It may seem possible for the file to no longer exists when attempting to rename. However, since undo reverses all operations after the `rename` first, any `delete` operation that deletes would have been recovered. This guarantees that the file defintely exists when executing.

Error Handling

1. Argument count checked with `assertArgumentCount` within the `Rename` constructor
2. File name validity checked via `assertValidFileName` in `curDir.rename`:

```
public void rename(String oldName, String newName) throws
    FileExistsException,
    FileNotFoundException,
    InvalidFileException
{
    assertValidFileName("new file name", newName); // <-- error checking
    if (dirMap.containsKey(oldName)) {
        setSubDirName(oldName, newName); // <-- more error handling in here
        return;
    }
    if (docMap.containsKey(oldName)) {
        setSubDocName(oldName, newName); // <-- more error handling in here
        return;
    }
    throw new FileNotFoundException(...); // <-- error
}
```

3. File name collision for `newName` checked within `setSubDirName` and `setSubDocname` via `assertUniqueName`

```
private void setSubDirName (String oldDirName, String newDirName) throws
    FileNotFoundException,
    FileExistsException
{
    assertDirExists(oldDirName); // <-- error checking
    assertUniqueName(newDirName); // <-- error checking
    Directory dir = dirMap.get(oldDirName);
    dir.setDirName(newDirName);
    dirMap.remove(oldDirName);
    dirMap.put(newDirName, dir);
}

private void setSubDocName (String oldDocName, String newDocName) throws
    FileNotFoundException,
```

```
        FileExistsException
    {
        assertDocExists(oldDocName); // <-- error checking
        assertUniqueName(newDocName); // <-- error checking
        Document doc = docMap.get(oldDocName);
        doc.setDocName(newDocName);
        docMap.remove(oldDocName);
        docMap.put(newDocName, doc);
    }
```

4. No file called `oldName` also checked checked within `setSubDirName` and `setSubDocName` via `assertDirExists` and `assertDocExists`.

[REQ6] changeDir dirName (Implemented ✅)

Implementation Detail

```
class ChangeDir implements Instruction{
    private final Controller control;
    private final Directory curDir;
    private final Directory newDir;

    public ChangeDir (Controller control, String[] args) throws
        InvalidCommandException,
        FileNotFoundException
    {
        assertArgumentCount("changeDir", args, 1);
        this.control = control;
        this.curDir = control.getCurDir();
        this.newDir = (args[0].equals("..")) ? curDir.getPa() : curDir.getDir(args[0]);

        if (newDir == null) {
            throw new FileNotFoundException("Cannot return: already at root");
        }
    }

    @Override
    public void execute() {
        ArrayList<String> cwd = control.getCwd();
        control.setCurDir(newDir);
        cwd.add(newDir.getDirName());
    }

    @Override
    public void reverse() {
        ArrayList<String> cwd = control.getCwd();
        control.setCurDir(curDir);
        int lastIndex = cwd.size() - 1;
        cwd.remove(lastIndex);
    }
}
```

`changeDir` is a **reversible** instruction, thus a dedicated `ChangeDir` object. The execution flow is as follows:

1. `Controller` creates `ChangeDir` object which implements `Instruction` interface

2. `ChangeDir` is passed into `history.run()`
3. `history.run()` calls the `execute()` method of `ChangeDir`
4. `execute()` changes the working directory

Error Conditions

1. If the target directory does not exist
2. If attempting to return the parent directory when already at `root` folder, potentially causing `NullPointerException`

Error Handling

1. Existence of target director is checked within `curDir.getDir()` method in the `ChangeDir` constructor. Specifically, the following line of code:

```
this.newDir = (args[0].equals("..")) ? curDir.getPa() : curDir.getDir(args[0]);
```

2. Preventing `NullPointerException` is done by checking if the returned parent directory by `getPa` is `null`. If `null`, we throw a `FileNotFoundException`

```
if (newDir == null) {
    throw new FileNotFoundException("Cannot return: already at root");
}
```

PRE-REQUISITE for [REQ7] & [REQ8]

3.0.1. `treeDfs` method

```
private ArrayList<Object> treeDfs(Directory curDfsDir, boolean recursive) {
    ArrayList<Object> res = new ArrayList<>();
    for (Directory dir: curDfsDir.getSubDirs().values()) {
        ArrayList<Object> subDirFiles;

        if (recursive) subDirFiles = treeDfs(dir, true);
        else subDirFiles =new ArrayList<>();

        Pair<Directory, ArrayList<Object>> subDir = new Pair<>(dir, subDirFiles);
        res.add(subDir);
      }
      res.addAll(curDfsDir.getSubDocs().values());
      return res;
  }
```

The `treeDfs` method recursively traverses `curDfsDir`, accepting a boolean parameter `recursive`. When `recursive` is `false`, it limits the search to the current directory without descending further. This design allows both `list` and `rList` in [REQ7] and [REQ8] to utilize the same DFS method.

The method returns an `ArrayList<Object>`, where each `Object` is either a `Pair<Directory, ArrayList<Object>>` or a `Document`. Using a `Pair` enables storing a `Directory` alongside another `ArrayList<Object>`, creating nested `ArrayList` structures of arbitrary depth to accurately represent a tree-like file hierarchy.

[REQ7] list (Implemented ✅)

Implementation Detail

```
private void list(String[] args) throws InvalidCommandException {
    assertArgumentCount("list", args, 0);
    view.displayFileTree(treeDfs(curDir, false));
}
```

Since `list` is an **irreversible** instruction, executing is simply calls the `list()` method of the `Controller`. The execution flow is as follows:

1. the `list` method first calls the `treeDfs` private method with `recursive = false`, so that it only searches one layer deep (**SEE** **Section 3.0.1**)
2. the `treeDfs` returns a list of all the files in the current direcotry
3. the file list object is sent to `displayFileTree` method of the `View`
4. the `displayFileTree` prints a one-layer ASCII tree and summary statistic (**SEE** **Figure 4**)

Error Conditions

When CVFS first loads, no disk are loaded, and current directory is `null`. But since this is an error condition common to nearly all commands except a few, it is discussed in the Section 3.1

Error Handling

No additional error handling needed.

[REQ8] rlist (Implemented ✅)

Implementation Detail

```
private void rList(String[] args) throws InvalidCommandException {
    assertArgumentCount("rList", args, 0);
    view.displayFileTree(treeDfs(curDir, true));
}
```

Since `rList` is an **irreversible** instruction, executing is simply calls the `rList()` method of the `Controller`. The execution flow is as follows:

1. the `rList` method first calls the `treeDfs` private method with `recursive = true` to DFS through **all** subdirectories (**SEE** **Section 3.0.1**)
2. the `treeDfs` returns the nested file hierachy of the current directory
3. the file hierachy object is sent to `displayFileTree` method of the `View`
4. the `displayFileTree` prints a beautiful, multi-layered ASCII tree and summary statistic (**SEE** **Figure 4**)

Error Conditions

When CVFS first loads, no disk are loaded, and current directory is `null`. But since this is an error condition common to nearly all commands except a few, it is discussed in the Section 3.1

Error Handling

No additional error handling needed.

> **PRE-REQUISITE for [REQ9] & [REQ10] & [REQ11]**
>
> A clear understanding of the `Criterion` interface (Section 2.3.4) is essential before proceeding to the following sections! Otherwise, it would be difficult to see how

[REQ9] newSimpleCri criName attrName op val (Implemented ✅)

Implementation Details

```
class NewSimpleCri implements Instruction {
    private final String crtName;
    private final VirtualDisk curDisk;
    private final Criterion crt;

    public NewSimpleCri(VirtualDisk curDisk, String[] args) throws
        InvalidCommandException,
        InvalidCriterionException,
        InvalidTypeException,
        InvalidValueException
    {
        assertArgumentCount("newSimpleCri", args, 4);
        assertValidCrtName(args[0]);
        this.curDisk = curDisk;
        this.crtName = args[0];
        String attrName = args[1];
        String op = args[2];
        String val = args[3];

        switch (attrName) {
            case "name" -> crt = new NameCriterion(crtName, op, val);
            case "type" -> crt = new TypeCriterion(crtName, op, val);
            case "size" -> crt = new SizeCriterion(crtName, op, val);
            default: throw new InvalidCriterionException(...);
        }
    }

    @Override
    public void execute() throws InvalidCriterionException {
        curDisk.newCrt(crtName, crt);
    }

    @Override
    public void reverse() {
        curDisk.delCrt(crtName);
    }
}
```

`newSipleCri` is a **reversible** instruction, hence a `NewSimpleCri` object that implements the `Instruction` interface is created. The execution flow as follows:

1. `Controller` creates a `NewSimpleCri` object
2. `NewSimpleCri` checks the type of criterion (name, type, or size)
3. `NewSimpleCri` creates the criterion objects accordingly
4. `NewSimpleCri` is passed into `history.run()`
5. `history.run()` executes the newSimpleCri by calling its `execute()` method
6. `execute()` passes the newly created criterion into `newCrt` method to place it store it within the VIRTUAL DISK

```
public void newCrt (String crtName, Criterion crt) throws ...{
    assertCrtNotExist(crtName);
    cMap.put(crtName, crt);
}
```

Error Conditions

1. User input incorrect number of arguments ($\neq 4$)
2. User input invalid criterion name (not 2 alphabets)
3. User input invalid attribute name (not one of name, type, size)
4. User input invalid operator or operand corresponding to the attribute

Error Handling

1. Incorrect argument count is checked via `assertArgumentCount` in `NewSimpleCri` constructor
2. Incorrect criterion name is checked via `assertValidCrtName` in `NewSimpleCri` constructor
3. Incorrect attribute name is handled in the `default` clause of `switch-case`
4. Invalid operator or operand:

    1. For name criterion, checks are performed within `NameCriterion` constructor. It throws `InvalidCriterionException` for either invalid operator or operand

    2. For size criterion, checks are performed within `SizeCriterion` constructor. It throws `InvalidCriterionException` if the operator is not one of $\{\leq, <, >, \geq, ==, \neq\}$, and throws `InvalidValueException` if the operand is not a numeric value OR that it exceeds `Long.MAX_VALUE`

    3. For type criterion, checks are performed within `TypeCriterion` constructor. It throws `InvalidCriterionException` if the operator is not "equals", and throws `InvalidTypeCriterion` if the operand file type is not one of {txt, html, css, java}

[REQ10] IsDocument (Implemented ✅)

Implementation Details

`IsDocument` is a pre-installed criterion. Therefore, implementing this command is as simple as adding a special `IsDocCriterion` that implements `Criterion` interface to every new disk. It can be retrieved in `rSearch` and `search` together with other crtierions.

```
public class IsDocCriterion implements Criterion{
    @Override public boolean eval (Document doc) { return true; }
    @Override public boolean eval (Directory dir) { return false;}
}
```

Error Conditions

When CVFS first loads, no disk are loaded, and current directory is `null`. But since this is an error condition common to nearly all commands except a few, it is discussed in the Section 3.1

Error Handling

No additional error handling needed.

[REQ11.1] newNegation criName1 criName2

Implementation Details

```
class NewNegation implements Instruction {
    private final String crtName;
    private final Criterion negCrt;
    private final VirtualDisk curDisk;

    public NewNegation(VirtualDisk curDisk, String[] args) throws
        InvalidCommandException,
        InvalidCriterionException
    {
        assertArgumentCount("newNegation", args, 2); // <-- error handling
        assertValidCrtName(args[0]); // <-- error handling
        this.crtName = args[0];
        Criterion subCrt = curDisk.getCrt(args[1]); // <-- error handling inside
        this.negCrt = new NegCriterion(crtName, subCrt); // <-- error handling inside
        this.curDisk = curDisk;
    }

    @Override
    public void execute() throws InvalidCriterionException {
        curDisk.newCrt(crtName, negCrt);
    }

    @Override
    public void reverse() {
        curDisk.delCrt(crtName);
    }
}
```

The `newNegation` command is **reversible**. Therefore, a special `NewNegation` object that implements the `Instruction` interface is created. Execution flow:

1. `Controller` creates `NewNegation` instruction object
2. `NewNegation` constructor creates new `NegCriterion` object
3. `NewNegation` is passed into `history.run()`
4. `history.run()` executes the instruction by calling `execute()`
5. `NegCriterion` is placed binded to the disk
6. `NewNegation` is pushed into `history.undoStack` for potential reversal.

Error Conditions

1. User input number of arguments $\neq 2$

2. User input criterion name (`crtName2`) that does not exist
3. User input criterion name (`crtName1`) already exists
4. User input negation criterion name (`crtName1`) invalid (not two alphabets)

Error Handling

1. Invalid argument count checked by `assertArgumentCount` in `NewNegation` constructor

2. Non-exist criterion name `crtName2` checked by `assertCrtExist` method in `curDisk.getCrt` which throws `InvalidCriterionException` when the criterion name does not exist.

```
public Criterion getCrt (String crtName) throws InvalidCriterionException {
    assertCrtExist(crtName); // <-- criterion existence checked here
    return cMap.get(crtName);
}
```

3. Criterion name collision checked by `assertCrtNameExist` in `curDisk.newCrt`

```
public void newCrt (String crtName, Criterion crt) throws
    InvalidCriterionException
{
    assertCrtNotExist(crtName); // <-- name collision checked here
    cMap.put(crtName, crt);
}
```

4. Invalid criterion name checked by `assertValidCrtName` in the `NewNegation` constructor.

[REQ11.2] newBinaryCri criName1 criName3 logicOp criName4 (Implemented ✅)

Implementation Details

```
class NewBinaryCri implements Instruction {
    private final VirtualDisk curDisk;
    private final String crtName;
    private final Criterion binCri;

    public NewBinaryCri(VirtualDisk curDisk, String[] args) throws
        InvalidCommandException,
        InvalidCriterionException
    {
        assertArgumentCount("newBinaryCri", args, 4); // <-- Error handling
        assertValidCrtName(args[0]); // <-- Error handling
        this.curDisk = curDisk;
        this.crtName = args[0];
        Criterion c1 = curDisk.getCrt(args[1]); // <-- Error check within
        Criterion c2 = curDisk.getCrt(args[3]); // <-- Error check within
        this.binCri = new BinaryCriterion(crtName, args[2], c1, c2);
    }

    @Override
    public void execute() throws InvalidCriterionException {
        curDisk.newCrt(crtName, binCri);
    }
```

```
    @Override
    public void reverse() throws InvalidCriterionException {
        curDisk.delCrt(crtName);
    }
}
```

The `newBinaryCri` command is **reversible**. Therefore, a special `NewBinaryCri` object that implements the `Instruction` interface is created. Execution flow:

1. `Controller` creates `NewBinaryCri` instruction object
2. `NewNegation` constructor creates new `NegBinaryCri` object
3. `NewBinaryCri` is passed into `history.run()`
4. `history.run()` executes the instruction by calling `execute()`
5. `NewBinaryCri` is placed binded to the disk
6. `NewBinaryCri` is pushed into `history.undoStack` for potential reversal.

Error Conditions

1. User input number of arguments $\neq 4$
2. User input criterion name (`crtName3` or `crtName4`) that does not exist
3. User input new criterion name (`crtName1`) already exists
4. User input criterion name (`crtName1`) invalid (not two alphabets)
5. User input invalid logic operator (`logicOp`)

Error Handling

1. Incorrect argument count checked by `assertArgumentCOunt` in `NewBinaryCri` constructor. Throws `InvalidCommandException`.
2. Non-exist names checked within `getCrt` (already discussed in [REQ11.1] `NewNegation`)
3. Name collision checked within `newCrt` (already discussed in [REQ11.1] `NewNegation`)
4. Invalid criterion name checked by `assertValidCrtName`
5. Invalid logic operator checked within `BinaryCriterion` constructor. Throws `InvalidCriteroinException` if operator is not one of {&&, ||}

```
    public BinaryCriterion(
        String crtName,
        String op,
        Criterion c1,
        Criterion c2
    ) throws
        InvalidCriterionException
    {
        if (!isValidOp(op)) throw new InvalidCriterionException(...);
        this.crtName = crtName;
        this.op = toOp(op);
        this.c1 = c1;
        this.c2 = c2;
    }
```

[REQ12] printAllCriteria (Implemented ✅)

Implementation Details

```
private void printAllCriteria () {
    view.displayCriteria(curDisk.getCMap());
}
```

Since `printAllCriteria` is irreversible, execution is a simple call of the `pinrtAllCriteria` method. The method as simple as retrieving the `HashMap<>` of criterions from the current disk and forward it to the `view` for console output.

Error Conditions

When CVFS initializes, no disk is loaded and the current directory is `null`. Since this error condition is common to nearly all commands except a few, it is addressed in Section 3.1 .

Error Handling

No additional error handling needed.

PRE-REQUISITE for [REQ13] & [REQ14]

3.0.2. `listDfs` method

```
private void listDfs(
    Directory curDfsDir,
    ArrayList<Pair<String, Object>> docs,
    ArrayList<Pair<String, Object>> dirs,
    String cwd, boolean recursive
) {
    for (Document doc: curDfsDir.getSubDocs().values()) {
          String docPath = String.format(
            "%s%s.%s",
            cwd, doc.getDocName(),
            doc.getType().toStrType()
          );
          docs.add(new Pair<>(docPath, doc));
      }
      for (Directory dir: curDfsDir.getSubDirs().values()) {
          String dirPath = String.format("%s%s/", cwd, dir.getDirName());
          dirs.add(new Pair<>(dirPath, dir));
          if (recursive) listDfs(dir, docs, dirs, dirPath, true);
      }
  }
```

The `listDfs` method recursively traverses `curDfsDir`, populating `docs` and `dirs` ArrayList with path-object. It constructs each document's path using `cwd`, the file name, and type (if `Document`). Recursively calls `listDfs` on the subdirectory if `recursive` is `true`. This approach enables both `search` and `rSearch` in [REQ13] and [REQ14] to sharethe same DFS method.

Two `ArrayList<Pair<String, Object>>` parameters are used to store documents and directories alongside their paths. Each `Pair` links a path string to either a `Document` or a `Directory`, effectively representing the hierarchical file structure. This organized storage captures the tree-like organization, facilitating easy access and manipulation of files and directories at any depth.

3.0.3. `filterList` method

```
private ArrayList<Pair<String, Long>> filterList(
    ArrayList<Pair<String, Object>> entries,
    Criterion crt)
{
    ArrayList<Pair<String, Long>> res = new ArrayList<>();
    for (Pair<String, Object> entry: entries) {
        Object file = entry.getSecond();
        String path = entry.getFirst();

        if (file instanceof Document) {
            Document doc = (Document) file;
            if (crt.eval(doc)) {
                res.add(new Pair<>(path, doc.getSize()));
            }
        } else {
            Directory dir = (Directory) file;
            if (crt.eval(dir)) {
                res.add(new Pair<>(path, dir.getSize()));
            }
        }
     }
    return res;
}
```

The `filterList` method processes the list (`docs` or `dirs`) poopulated by `listDfs`, filtering them based on a specified `Criterion`. It iterates through each `Pair<String, Object>` in `entries`, evaluates the criterion against either the `Document` or `Directory` object, and if the criterion is met, adds it to the result.

The method returns an `ArrayList<Pair<String, Long>>`, where each `Pair` consists of a path `String` and a `Long` representing the size of the corresponding `Document` or `Directory`.

[REQ13] search criName (Implemented ✅)

Implementation Details

```
private void search(String[] args) throws
    InvalidCommandException,
    InvalidCriterionException
{
    assertArgumentCount("search", args, 1); // <-- error handling
    Criterion crt = curDisk.getCrt(args[0]); // <-- error handling inside
    ArrayList<Pair<String, Object>> docs = new ArrayList<>();
    ArrayList<Pair<String, Object>> dirs = new ArrayList<>();
    listDfs(curDir, docs, dirs, "/", false);
    view.displayFileList(filterList(docs, crt), filterList(dirs, crt));
}
```

Since `search` is irreversible, it is just a simple method call. Its execution flows is as follows:

1. Retrieves the `Criterion` object with the user specifie:d name
2. Initializes two empty `ArrayList<>`, `docs` and `dirs`

3. Calls the `listDfs` (see Section 3.0.2) method with `recursive = false`, which popuates both `docs` and `dirs` with documents and directories directly contained in the current directory.
4. The populated lists `docs` and `dirs` are further processed by `filterList` (see Section 3.0.3) to remove any files that does not satisfy the given criterion.
5. Finally, the the filtered `docs` and `dirs` list are forwarded to `displayFileList` method in the `View` class to diplay to the user.

Error Conditions

1. User input number of arguments $\neq 1$ (i.e., criName)
2. User input criterion name that does not exist

Error Handling

1. Invalid argument count is handled by `assertArgumentCount`
2. Not exist criterion name checked within `curDisk.getCrt`

[REQ14] rSearch criName (Implemented ✅)

Implementation Details

```
private void rSearch(String[] args) throws
    InvalidCommandException,
    InvalidCriterionException
{
    assertArgumentCount("rSearch", args, 1);
    Criterion crt = curDisk.getCrt(args[0]);
    ArrayList<Pair<String, Object>> docs = new ArrayList<>();
    ArrayList<Pair<String, Object>> dirs = new ArrayList<>();
    listDfs(curDir, docs, dirs, "/", true);
    view.displayFileList(filterList(docs, crt), filterList(dirs, crt));
}
```

Since `rSearch` is irreversible, it is just a simple method call. Its execution flows is as follows:

1. Retrieves the `Criterion` object with the user specified name
2. Initializes two empty `ArrayList<>`, `docs` and `dirs`
3. Calls the `listDfs` (see Section 3.0.2) method with `recursive = true`, which popuates both `docs` and `dirs` with **all** files and subdirectories.
4. The populated lists `docs` and `dirs` are further processed by `filterList` (see Section 3.0.3) to remove any files that does not satisfy the given criterion.
5. Finally, the the filtered `docs` and `dirs` list are forwarded to `displayFileList` method in the `View` class to diplay to the user.

Error Conditions

1. User input number of arguments $\neq 1$ (i.e., criName)
2. User input criterion name that does not exist

Error Handling

1. Invalid argument count is handled by `assertArgumentCount`

2. Not exist criterion name checked within `curDisk.getCrt`

```
>>>  rSearch bb

Matching Documents:
/root1/home/tony/Document/homeDish.txt 184
/root1/home/tony/Code/Model.java       160
/root1/home/tony/Code/Controller.java  192

Matching Directories:
/root1/                                1110
/root1/home/                           1070
/root1/home/tony/                      1030
/root1/home/tony/Downloads/            40
/root1/home/tony/Document/             382
/root1/home/tony/Code/                 568
/root2/                                40
/root3/                                40
-----------------------------------------------
Total Files                            Total Size
11                                     4816
```

Figure 5: example output of `displayFileList` (called by `search` and `rSearch`)

[REQ15] save path (Implemented ✅)

Implementation Details

Our CVFS implements the save functionality via Java's built-in `Serializable` interface. By adding `implements Serializable` to every class in `Model` package, dumping an entire virual disk into a binary file is simple as follows:

```
try (FileOutputStream fos = new FileOutputStream(path.toFile());
     ObjectOutputStream oos = new ObjectOutputStream(fos)) {
    oos.writeObject(obj);
} catch (...) {
    ...
}
```

Error Conditions

While implementing the serialize functionality, File I/O has notoriously many error conditions. The following is a comprehensive list:

1. The provided path string is either `null` or contains only whitespace.
2. The path string does not conform to valid path syntax.
3. The parent directory of the specified path does not exist.
4. The parent directory lacks write permissions.
5. The path refers to an existing directory instead of a file.
6. A file already exists at the specified path.

Error Handling

```
public static Path getWritablePath(String pathStr) throws InvalidIOException {
    if (pathStr == null || pathStr.trim().isEmpty())
        throw new InvalidIOException(...); // 1. null or empty path
    Path path;
    try {
        path = Paths.get(pathStr);
    } catch (InvalidPathException e) {
        throw new InvalidIOException(...); // 2. invalid path syntax
    }
    Path parent = path.getParent();
    if (parent != null) {
        if (!Files.exists(parent))
            throw new InvalidIOException(...); // 3. parent not exist
            if (!Files.isWritable(parent))
                throw new InvalidIOException(...); // 4. parent not writeable
        }
        if (Files.exists(path)) {
            if (Files.isDirectory(path))
                throw new InvalidIOException(...); // 5. path points to directory
            else
                throw new InvalidIOException(...); // 6. file already exists
        }
        return path;
    }
}
```

The `getWritablePath` method addresses these issues by validating the input path and its context. It ensures that the path is neither `null` nor empty, confirms the path format is valid, and checks that the parent directory exists and is writable. Additionally, it verifies that the target path does not already exist and does not point to a directory. If any of these conditions fail, the method throws an `InvalidIOException` with a relevant message, thereby preventing invalid or unsafe file operations during the serialization process.

[REQ16] load path (Implemented ✅)

Implementation Detail

Mirroring the `save` command, the `load` command simply reads serialized file via `ObjectInputStream` and `FileInputStream`.

```
try (FileInputStream fis = new FileInputStream(path.toFile());
     ObjectInputStream ois = new ObjectInputStream(fis)) {
    return ois.readObject();
} catch (...) {
    ...
}
```

Error Conditions

While implementing the read functionality, File I/O has notoriously many error conditions. The following is a comprehensive list:

1. The provided path string is either `null` or contains only whitespace.

2. The path string does not conform to valid path syntax.
3. The file does not exist at the specified path.
4. The path refers to a directory instead of a file.
5. The file lacks read permissions.

Error Handling

```
public static Path getReadablePath(String pathStr) throws InvalidIOException {
    if (pathStr == null || pathStr.trim().isEmpty())
        throw new InvalidIOException(...); // 1. null or empty path
    Path path;
    try {
        path = Paths.get(pathStr);
    } catch (InvalidPathException e) {
        throw new InvalidIOException(...); // 2. invalid path syntax
    }
    if (!Files.exists(path))
        throw new InvalidIOException(...); // 3. file not found
    if (Files.isDirectory(path))
        throw new InvalidIOException(...); // 4. path points to directory
    if (!Files.isReadable(path))
        throw new InvalidIOException(...); // 5. not readable
    return path;
}
```

The `getReadablePath` method addresses these issues by validating the input path and its context. It ensures that the path is neither `null` nor empty, confirms the path format is valid, checks that the file exists at the specified location, verifies that the path refers to a file rather than a directory, and ensures that the file is readable. If any of these conditions fail, the method throws an InvalidIOException with a relevant message, thereby preventing invalid or unsafe file operations during the read process.

[REQ17] quit (Implemented ✅)

Implementation Details

When `quit` method is called by the `Controller`, it sets the `isRunning` flag to false, which results in the outer-most console I/O `while-loop` to short-circuit and end.

```
    private void quit() {
        assertArgumentCount("quit", args, 0);
        isRunning = false;
    }
```

Error Conditions

1. Incorrect number of arguments ($\neq 0$)

Error Handling

1. Checked via `assertArgumentCount`

[BON1] save/load search criteria (Implemented ✅)

Implementation Details

Save and load criteria is already achieved in our standard implementation. Since our search criterions are stored in a `HashMap<String, Criterion>` within a `VirtualDisk` object, and because both `Criterion` and Java `HashMap` implements the `Serializable` interface, search criterions will always be serialized / deserialized with everything else

Error Conditions

No additional error conditions apart from the ones discussed in `save` and `load`.

Error Handling

None.

[BON2] undo/redo instructions (Implemented ✅)

Implementation Details

The implementation of all **reversible** instructions via the unified `Instruction` interface (see Section 2.2.2) and the `History` class (see Section 2.2.3) is heart to realizing the undo / redo functionality.

To reiterate, the `Instruction` interface enforces an `execute` and `reverse` method for all classes that implements it. As their name suggests, `execute` runs the instruction as intended, whereas `reverse` safely undo its effects. This allows us to abstract away the differences between **reversible** instructions and treat them as just **one type** of instruction, making the `History` class possible.

```
public class History {
    private final Stack<Instruction> undoStack;
    private final Stack<Instruction> redoStack;

    void run(Instruction inst) throws CVFSException;
    void redo() throws CVFSException;
    void undo() throws CVFSException;
}
```

The `History` class comprise two primary variable and three methods.

`run()` Initially, the newly created `Instruction` object is passed to the `run()` method. `run()` will attempt to execute the instruction; if succeeded, the `Instruction` object is pushed into the `undoStack`; otherwise, the invalid instruction is disregarded. **Note**: whenever `run()` is called, `redoStack` is cleared.

`undo()` To undo the last action, 1) `pop()` the top `Instruction` object from the `undoStack` 2) executes the `reverse()` method of the `Instruction` 3) pushes the instruction into `redoStack`

`redo()` To redo the last undone action, 1) `pop()` the top `Instruction` object from `redoStack` 2) call the `execute()` method of the `Instruction` to execute it again 3) pushes the instruction back to `undoStack`.

Error Conditions

1. User input invalid number of arguments ($\neq 0$)
2. `redo()` when `redoStack` is empty
3. `undo()` when `undoStack` is empty
4. `redo()` in unknown context can be dangerous. For example,

> Consider this command sequence:
>
> $$\text{newDir A} \rightarrow \text{newDoc B} \rightarrow \text{undo} \rightarrow \text{rename A B}$$
>
> After undoing "newDoc", executing "rename" creates a potential conflict. If we attempt to `redo` the `newDoc` operation:
>
> $$\text{newDir A} \rightarrow \text{newDoc B} \rightarrow \text{undo} \rightarrow \text{rename A B} \; (\rightarrow \text{redo?})$$
>
> This sequence would create a critical conflict since directory A has been renamed to B. Attempting to `redo` and restore document B would result in a name collision with the renamed directory. For this reason, `redo()` operations should only be permitted immediately after an `undo()`

5. `undo()`/`redo()` after `newDisk` may cause `NullPointerException`

Error Handling

1. Invalid argument count handled via `assertArgumentCount`

```
private void undo(String[] args) throws CVFSException {
    assertArgumentCount("undo", args, 0);
    history.undo();
}

private void redo(String[] args) throws CVFSException{
    assertArgumentCount("redo", args, 0);
    history.redo();
}
```

2. `pop()` empty `redoStack` checked via `assertNotEmpty()`

```
// in `History` class
protected void redo() throws CVFSException {
    assertNotEmpty(redoStack);
    Instruction inst = redoStack.pop();
    inst.execute();
    undoStack.push(inst);
}
```

3. `pop()` empty `undoStack` checked via `assertNotEmpty()`

```
// in `History` class
protected void undo() throws CVFSException {
    assertNotEmpty(undoStack);
    Instruction inst = undoStack.pop();
    inst.reverse();
    redoStack.push(inst);
}
```

4. patched by **clearing** `redoStack` everytime a new instruction is ran

5. Everytime a `newDisk` is executed, both `undoStack` and `redoStack` are cleared.

3.1. [COMMON ERROR CONDITION]

3.1.1. Common Error Condition

Initially, CVFS has no disk loaded. In this situation, no instructions should be allowed to execute. Otherwise, `newDir` for example would cause `NullPointerException` due to current working directory not initialized yet.

3.1.2. Error Prevention

A conditional check is posititiond at the very begining of the `Controller`:

```
if (curDisk == null) {
    switch (cmd) {
        case "newDisk", "quit", "load" -> break;
        default -> throw new InvalidActionException(...);
    }
}
```

4. Reflection on My Learning-to-Learn Experience

by Wang Yuqi:

1) reflect on learning-to-learn experience

Working on this project has been remarkeably fruitful. As someone who mains competitive programming, which is more about short-form code that are intellectually challenging, coding projects of this magnitude was a fresh experience. It challenges me to explore uncharted waters and learning to learn. I would categorize my learning-to-learn experience into external and internal.

External learning is the time where I learned to seek external resources for help. One noteworthy example is when researching how I learned to use the `Serializable` interface of Java. Determined to fully understand its underlying mechanisms to avoid any potential pitfalls, a dedicated a whole day just to unravel the low-level implementation of the Serializable interface. My initial understanding of `Serializable` was surface-level – considering it as just a marker interface that somehow qualifies objects for serialization. But it was far more sophisticated.

One of the nuanced technical aspects I encountered was the concept of class identifiers. Initially, a quick search online suggested that the “best practice” is to include a custom Seri-

alVersionUID in each class. Curious about the reasoning behind this, I delved deeper. I learned that SerialVersionUID is a unique identifier for each Serializable class in Java, used during the deserialization process to ensure that a loaded class corresponds exactly to a serialized object. If the SerialVersionUID of the class does not match that of the serialized object, a InvalidClassException is thrown. This mechanism primarily serves as a version control system for classes

This understanding led me to reconsider my own use case. Instead of blindely following the "best practice", I decided to let Java handle class identifciaiton, as it inherently allows `InvalidClassException` to occurr even with minor changes to the code. Since CVFS is not designed to have a version control system, this seemingly unsafe practice would paradoxically prevent potential pitfalls.

Internal learning, on the other hand, refers to the way I brainstormed and experimented with various object-oriented programming (OOP) design patterns. Without external input, the simple act of prototyping allowed me to gain hands-on experience and insights into the pros and cons of different design patterns. In hindsight, I find prototyping to be a valuable learning tool that is highly beneficial for rapidly garnering engineering experience.

2) describe plan to improve

Looking forward, I feel more confident in my ability to learn in the face of new challenges. To take my self-learning ability to the next-level, I plan to expand on my currrent foundation of external and internal learning and merge them together. While external learning focuses on taking in unfamiliar topics and internal learning focuses on reinforcing existing knowledge, a combined approach would be to directly learn new concepts with concrete hands-on experimentation. For example, instead of merely reading about the HTML, a more concrete way of learning would be to implement an HTTP client and server from scratch.

by Cui Mingyue, Chen Yangxuan

1) reflect on learning-to-learn experience

Firstly, I gained a deep understanding of the principles of Object-Oriented Programming (OOP), particularly encapsulation and polymorphism. These principles helped me design a code structure that was easy to extend and maintain.

In creating UML diagrams. Initially, I struggled with how to effectively represent the structure of my code. However, as I delved deeper into UML, I learned to clearly illustrate the relationships and interactions between classes. I found that the UML diagrams automatically generated by IDEA did not always accurately reflect my code logic, leading me to learn how to modify and enhance these diagrams using draw.io. This not only improved my visualization skills but also deepened my understanding of the complexities involved in system design.

Additionally, I focused on mastering the design and implementation of a Command Line Interface (CLI). By implementing various commands (such as creating and deleting files and directories), I enhanced my understanding of file system operations. I also learned how to handle user input errors and exceptions to improve the robustness of the program.

Finally, I developed skills related to teamwork and project management. I communicated effectively within the group to ensure that each member's contributions were recognized and that we could successfully complete the project stages.

2) describe plan to improve

- Action: Define specific, measurable goals for each learning session, such as mastering a new programming concept or completing a certain number of coding challenges.
- Outcome: This will provide direction and motivation for my learning activities.

Diversify Learning Resources:

- Action: Explore various online platforms (like Coursera, edX, and YouTube) to access different types of content, including videos, articles, and interactive coding exercises.
- Outcome: Exposure to diverse perspectives and teaching styles will deepen my understanding of complex topics.

Establish a Regular Study Schedule:

- Action: Create a consistent weekly schedule that allocates specific time blocks for focused study, practice, and reflection.
- Outcome: This routine will help build discipline and ensure regular progress in my learning journey.

Engage in Active Learning:

- Action: Instead of passively consuming information, I will engage in active learning techniques such as coding projects, peer teaching, and participating in coding communities.
- Outcome: Active engagement will reinforce my knowledge and improve retention.

Reflect on Learning Experiences:

- Action: After completing each learning module or project, I will take time to reflect on what I learned, what worked well, and what could be improved.
- Outcome: Regular reflection will help me identify effective strategies and areas for growth.

Seek Feedback and Collaboration:

- Action: Collaborate with peers or mentors to gain feedback on my work and discuss challenging concepts.
- Outcome: Collaborative learning will provide new insights and enhance my understanding through discussion and shared experiences.