Project Report

COMP 2021 Object-Oriented Programming (Fall 2024)
Group 8

Members and contribution percentage:

HUANG Lingru 33.3%
SHEN Linghui 33.3%
TAN Rou Xin 33.3%

1 Introduction

This document aims to describe the design and implementation of the Comp Virtual File System (CVFS) by Group 8. The project is part of the course, COMP 2021 Object-Oriented Programming, at the Hong Kong Polytechnic University.

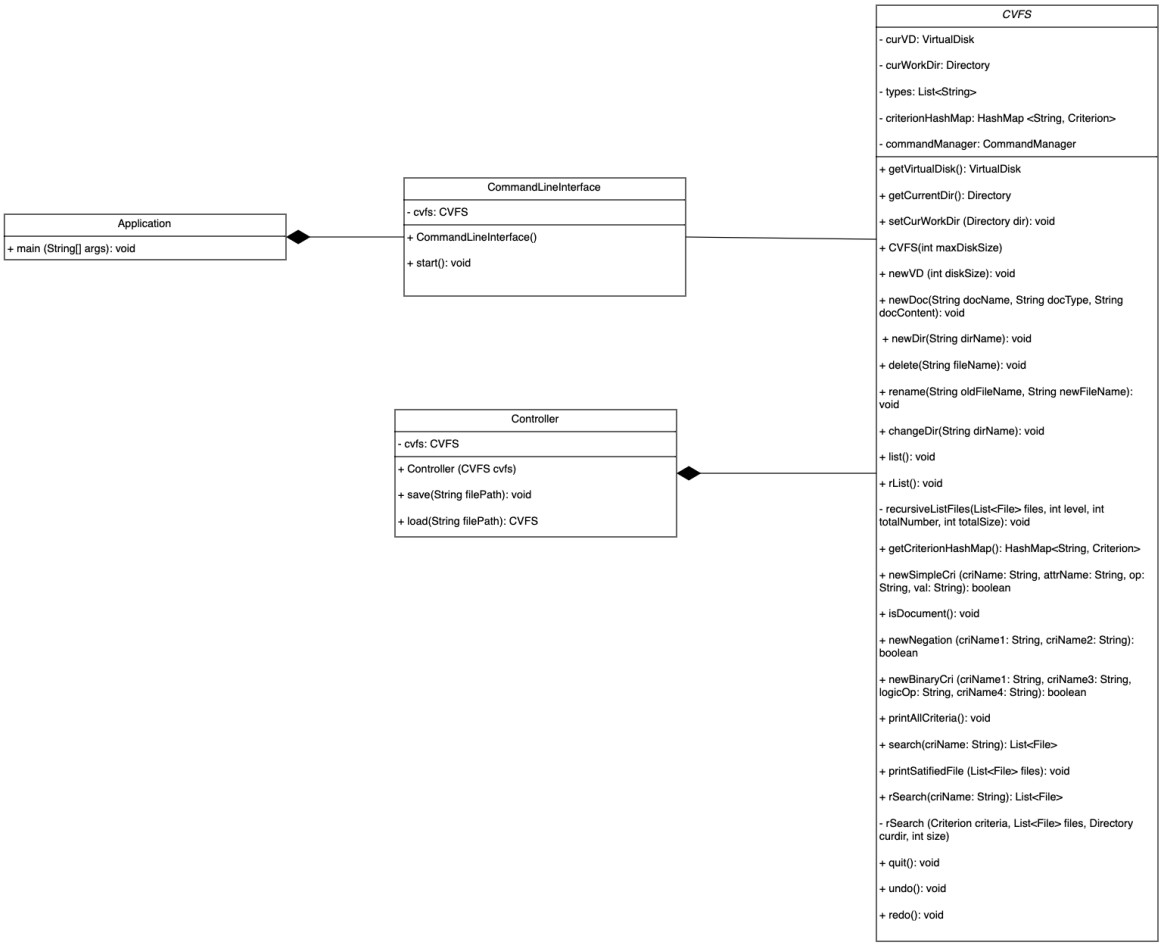
2 The COMP Virtual File System (CVFS)

This section will be separated into 2 parts. 2.1 will include the overall design of the CVFS, and 2.2 will be the implementation details according to the requirements.

2.1 Design

https://drive.google.com/file/d/1dH9q9D_cR52LpU_Yj-4-Caz7vn4REvve/view?usp=sharing

2.1.1 The Command Line Interface

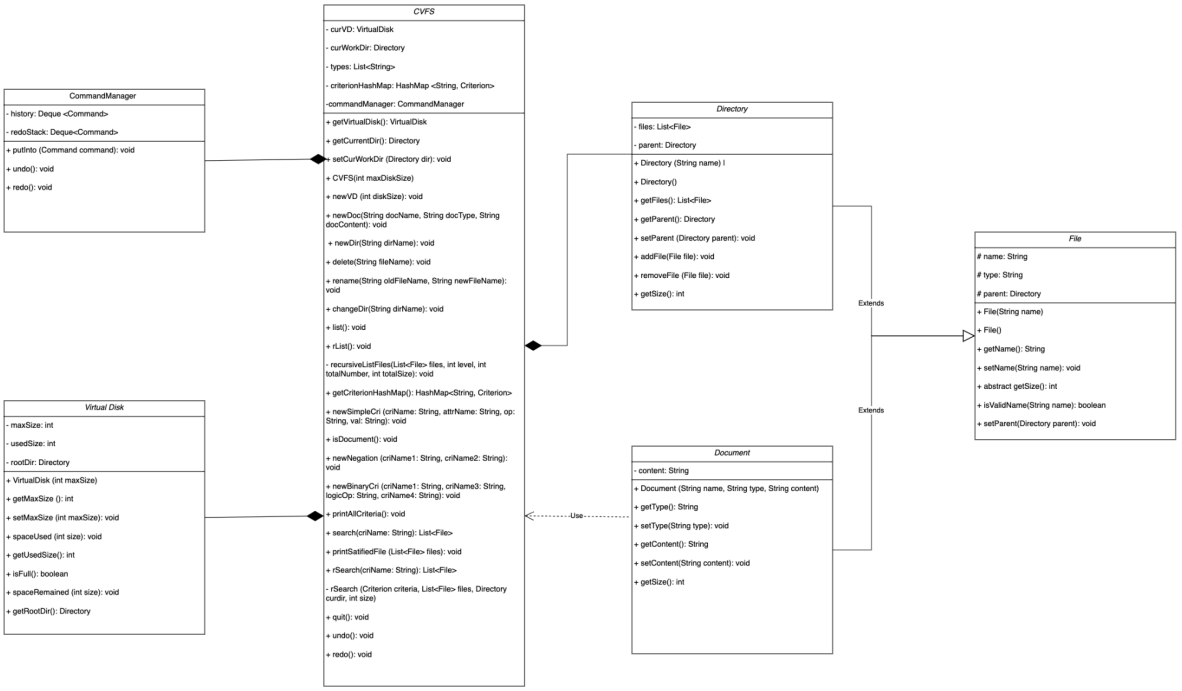


The Command Line Interface represents an interface to interact with the COMP Virtual File System, also known as CVFS in short. Started by the class `Application` and passed in a CVFS object, It sets up an initial message and continuously accepts user input until the user decides to quit. Upon receiving user input, the system will call out various methods and provide feedback accordingly. All the exceptions thrown in other methods will be caught and

handled in `CommandLineInterface.class`. Class `Controller` helps users interact with CVFS by saving the entire CVFS object into a specific local path or loading CVFS into the system.

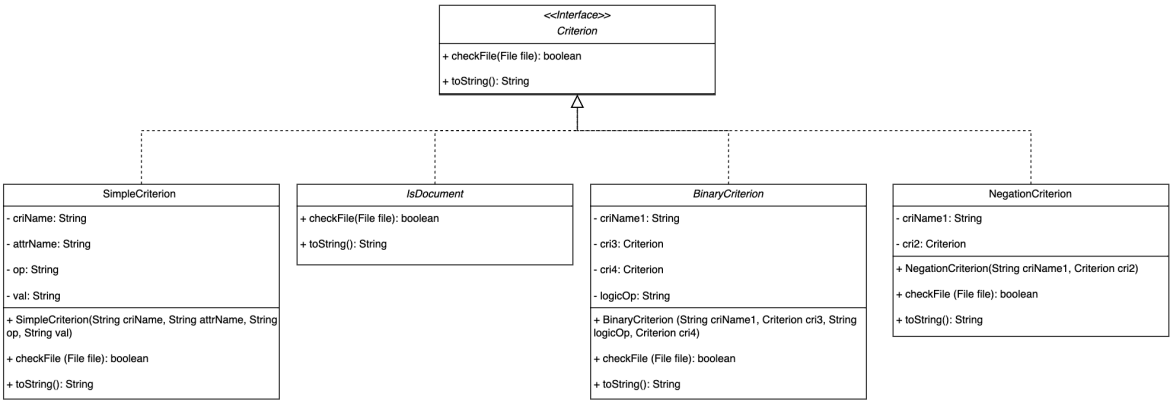
```
public class Application {
   public static void main(String[] args){
        CVFS cvfs = new CVFS(10000);
        // Initialize and utilize the system
        CommandLineInterface cli = new CommandLineInterface(cvfs);
        cli.start();
   }
}
public class CommandLineInterface {
   private CVFS cvfs;

   public CommandLineInterface(CVFS cvfs) {
        this.cvfs=cvfs;
   }

   public void start() {
        Scanner scanner = new Scanner(System.in);
        boolean running = true;

        System.out.println();

       System.out.println("**************************************************
       ********************************************************************
       ***************");
        System.out.println("                                    Welcome to the
Command-Line Based COMP Virtual File System!
");
        System.out.println("                                  The system
provides you with diverse commands for file management.");
        System.out.println("                                The CVFS system is
completed by HUANG Ling Ru, SHEN Ling Hui, Tan Rou Xin");

System.out.println("************************************************************
************************************************************************
*");
        System.out.println();
        System.out.println("Available commands: newVD, newDoc, newDir, delete,
rename, changeDir, list, rList\n"+
                "                    newSimpleCri, criIsDoc, newNegationCri,
newIsDocCri, newBinaryCri, search, rSearch\n"+
                "                    save, load, quit.");
        System.out.println("Now you can input your commands here (Case doesn't
matter):");
       while(running){try{.....} catch(){...}
       }
}
```

2.1.2 The Structure of The COMP Virtual File System (CVFS)



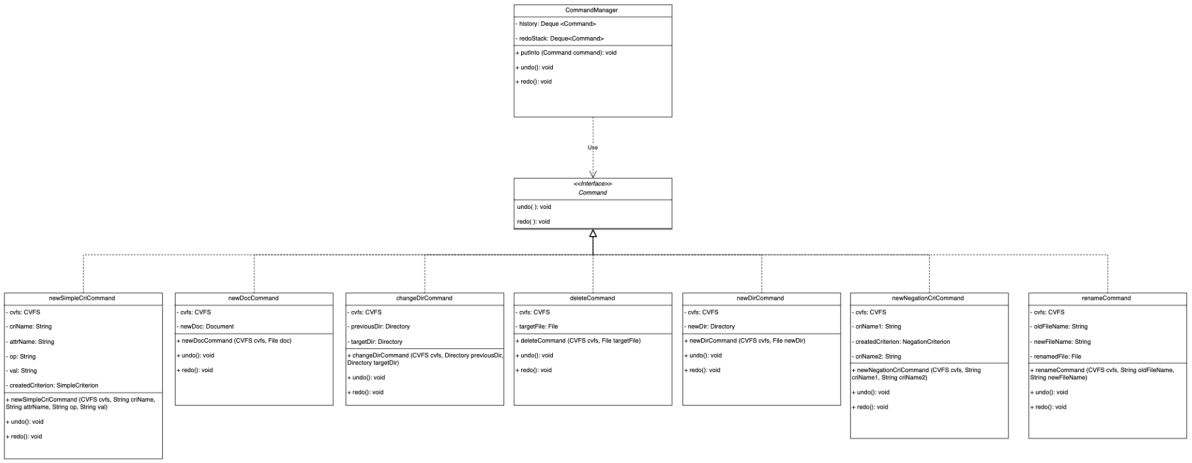
The main class performs the majority of actions. It can initialise a virtual disk with size, with one CVFS working on one virtual disk only at a time. Within the virtual disk, file stimulation could happen. There are two types of files, document-typed and directory-typed. The system is able to create new files, delete files, rename files, change the directory of the files, listing and recursively listing the files.

2.1.3 Criterion Design



The system also supports file selection in terms of simple criteria, composite criteria and assuring document-typed criteria. All criteria constructed can be listed to search and recursively search from the entire system.

2.1.4 Undo Redo Command Design - Bonus 2



The system also supports reversing the unwanted commands of changing the directory, deleting files, creating files, renaming files, and creating composite and simple criteria that the user has done earlier. In the same way, the system also supports redoing the commands.

2.2 Implementation of Requirements

[REQ 1]

1) The requirement is implemented.

2) A new disk is created with a specified maximum disk size. The newly created disk would be in the current working directory, which is also the root directory. Any available working disk will automatically be closed by the system.

3) No error situations.

```
public void newVD(int diskSize){
   this.curVD=new VirtualDisk(diskSize);
   this.curWorkDir=curVD.getRootDir();
}
```

[REQ 2]

1) The requirement is implemented.

2) The user is able to create a new document with a specified document name, document type and content. The program is then responsible for adding a new document to the current directory within the system. With the size of the new document, the program will update the spaces used by the virtual disk.

   The document name is only limited to 10 alphanumeric characters, which means to choose 10 characters from A to Z, a to z and 0 to 9, but at least include one English letter. Document type is limited to a few choices of “txt”, “java”, “html” or “css”.

3) Error situation 1: [File Name Validation]

The name of the newly created document has been restricted to not more than 10 alphanumeric letters with at least one English letter.
`IllegalArgumentException` will be thrown for invalid names.

Error situation 2: [File Type Validation]
The program allows only file types (txt, java, html, and CSS). If a file type is invalid, an `IllegalArgumentException` will be thrown.

Error situation 3: [Disk Space Check]
In case the disk does not have enough space to keep the newly created content, an `IllegalStateException` will be thrown.

Error situation 4: [Logic validation]
In case if the name of the new document formed has the same name as the files that were already in the current working directory. `IllegalArgumentException` will be thrown.

```
public void newDoc(String docName, String docType, String docContent){
    if(docName.length()>10 || !File.isValidName(docName)){
        throw new IllegalArgumentException("Invalid file name");
    }
    if (!types.contains(docType)) {
        throw new IllegalArgumentException("Invalid document type.
Allowed types are: " + String.join(",", types));
    }
    if (curVD.isFull()) {
        throw new IllegalStateException("Disk is full");
    }
    Document newDoc= new Document(docName,docType,docContent);
    curVD.spaceUsed(newDoc.getSize());
    curWorkDir.addFile(newDoc);
}
```

[REQ 3]

1) The requirement is implemented.

2) A new directory will be created with a specified directory name in the current working directory. Space used by the new directory in the virtual disk will also be updated.

   The directory name is only limited to 10 alphanumeric characters, which means to choose 10 characters from A to Z, a to z and 0 to 9, but at least include one English letter.

3) Error situation 1: [Directory Name Validation]
   Same as [File Name Validation] at REQ 2.3

   Error situation 2: [Disk Space check]
   Same as [Disk Space Check] at REQ 2.3

Error situation 3: [Logic validation]
Same as [Logic validation] at REQ 2.3

```
public void newDir(String dirName){
    if(dirName.length()>10 || !File.isValidName(dirName)){
        throw new IllegalArgumentException("Invalid file name");
    }
    if (curVD.isFull()) {
        throw new IllegalStateException("Disk is full");
    }
    Directory newDir=new Directory(dirName);
    curVD.spaceUsed(newDir.getSize());
    curWorkDir.addFile(newDir);
}
```

[REQ 4]

1) The requirement is implemented.

2) The file can be deleted with the specification of the file name. When the file name is received by the program, the program will iterate through all files in the current working directory and assign the file as the targeted file. The targeted file will be removed, and its file size will be cleared. A confirmed message "`File deleted successfully`." will be printed after the completion of deletion.

3) [File Existence Check]
In case if file name given was not found, an `IllegalArgumentException` will be thrown.

```
public void delete(String fileName){
    File targetFile = null;
    for (File file : curWorkDir.getFiles()) {
        if (file.getName().equals(fileName)) {
            targetFile = file;
            break;
        }
    }

    if (targetFile == null) {
        throw new IllegalArgumentException("File does not exist");
    }

    curWorkDir.removeFile(targetFile);
    curVD.spaceRemained(targetFile.getSize());
    System.out.println("File deleted successfully.");
}
```

[REQ 5]

1) The requirement is implemented.

2) The file can be renamed with the specification of the file name. When the old file name is received by the program, the program will iterate through all files in the current working directory and assign the file as the targeted file. The targeted file will be renamed. A confirmed message "`File renamed successfully.`" will be printed after the completion of the rename.

   The new file name is only limited to 10 alphanumeric characters, which means choose 10 characters from A to Z, a to z and 0 to 9, but at least include one English letter.

3) [File Existence Check]
   In case the old file name is not found, an `IllegalArgumentException` will be thrown.

   Error situation 2: [File Name Validation]
   The new name has been restricted to not more than 10 alphanumeric characters with at least one English letter. `IllegalArgumentException` will be thrown for invalid names.

   Error situation 3: [Logic validation]
   In case if the name of the new document formed has the same name as the files that were already in the current working directory. `IllegalArgumentException` will be thrown.

```
public void rename(String oldFileName, String newFileName){
    File targetFile = null;
    for (File file : curWorkDir.getFiles()) {
        if (file.getName().equals(oldFileName)) {
            targetFile = file;
            break;
        }
    }
    if (targetFile == null) {
        throw new IllegalArgumentException("File does not exist");
    }
    if (newFileName.length() > 10 || !File.isValidName(newFileName)) {
        throw new IllegalArgumentException("Invalid new file name");
    }
    targetFile.setName(newFileName);
    System.out.println("File renamed successfully.");
}
```

[REQ 6]

1) The requirement is implemented.

2) This method supports a change in the directory. If two dots (. .) were found in the current directory, the parent directory, if available, would be the current new working directory. Else, the current working directory remains in the root directory. For the rest

of the old directory without two dots (. .), the system will iterate through all files in the current directory to look for a directory with a specified name as the new directory. A confirmed message will be printed after successfully renaming the directory.

3) [Directory Existence Check]
In case the old directory name is not found, an `IllegalArgumentException` will be thrown.

```
public void changeDir(String dirName){
    if (dirName.equals("..")) {
        if (curWorkDir.getParent() != null) {
            curWorkDir = curWorkDir.getParent();
            System.out.println("Changed directory to the parent
directory.");
        } else {
            System.out.println("Already in the root directory.");
        }
    } else {
        for (File file : curWorkDir.getFiles()) {
            if (file instanceof Directory &&
file.getName().equals(dirName)) { //这里
                curWorkDir = (Directory) file;
                System.out.println("Changed directory to " + dirName);
                return;
            }
        }
        throw new IllegalArgumentException("Directory not found");
    }
}
```

[REQ 7]

1) The requirement is implemented.

2) The system will get all files from the current working directory. For all document-type files, the system will return its name, type and size. Whereas for all directory-typed files, the system will return its name and size. Total number of files and total size of all files would be finalised by increment of every file read. The message of the total number and total size of the file will be printed.

3) No error situations.

```
public void list(){
    List<File> files = curWorkDir.getFiles();
    int totalNumber = 0;
    int totalSize = 0;
    System.out.println("Files in directory " +
curWorkDir.getName()+":");
    for (File file : files) {
        if (file instanceof Document) {
            Document doc = (Document) file;
```

```
            System.out.printf("%s %s %d bytes%n", doc.getName(),
doc.getType(), doc.getSize());
        } else if (file instanceof Directory) {
            Directory dir = (Directory) file;
            System.out.printf("%s %d bytes%n", dir.getName(),
dir.getSize());
        }
        totalNumber++;
        totalSize += file.getSize();
    }

    System.out.println("Total files in the working directory: " +
totalNumber);
    System.out.println("Total size of the working directory: " +
totalSize + " bytes");
}
```

[REQ 8]

1) The requirement is implemented.

2) The method `rList()` sets up initial parameters to keep the record of the level of recursion, total number and total size of files. The method `recursiveListFiles()` iterates through all files for the documents and directories through recursion. For all files read, the system will recursively call the method for appropriate recursion level incrementation, updating the total number and total size used and printing information of the files. Given document-typed files, name, type and size will be printed. Whereas for directory-typed files, name and size will be printed. At the root level when level = 0, the system will print the final number and size of the total files.

3) No error situations.

```
public void rList(){
    List<File> files = curWorkDir.getFiles();
    int totalNumber = 0;
    int totalSize = 0;
    int level=0;
    recursiveListFiles(files, level, totalNumber, totalSize);
}

private void recursiveListFiles(List<File> files, int level, int
totalNumber, int totalSize){
    for (File file : files) {
        String indentation = "  ".repeat(level);
        if (file instanceof Document) {
            Document doc = (Document) file;
            System.out.printf("%s%s %s %d bytes%n", indentation,
doc.getName(), doc.getType(), doc.getSize());
            totalNumber++;
            totalSize += doc.getSize();
        } else if (file instanceof Directory) {
```

```
                Directory dir = (Directory) file;
                System.out.printf("%s%s %d bytes%n", indentation,
        dir.getName(), dir.getSize());
                totalNumber++;
                totalSize += dir.getSize();
                recursiveListFiles(dir.getFiles(), level + 1, totalNumber,
        totalSize);
            }
        }

        if (level == 0) {
            System.out.println("Total files: " + totalNumber);
            System.out.println("Total size: " + totalSize + " bytes");
        }
    }
```

[REQ 9]

1) The requirement is implemented.

2) This method creates a new criterion with four *String* parameters: name as `criName`, attributes as `attrName`, operators as `op` and value as `val`. Restriction of parameters according to the requirement is written in the class `SimpleCriterion implements Criterion, Serializable`. The system will add a successfully created criterion into `criterionHashMap` which is defined earlier as a field in the *CVFS* class.

3) [Criterion Name Check] `IllegalArgumentException` thrown due to unsuitable parameters requirement:
   - *criName* is not 2 letters
   - *attrName* is not either “name”, “type” or “size”
   - With *attrName* type of “name”, *op* does not include “contains” or *val* does not end and starts with double quotes
   - With *attrName* type of “type”, *op* does not include “equals” or *val* does not starts and ends with double quotes
   - With *attrName* type of “size”, *op* does not either include “<”, “>”, “<=”, “>=”, “==” or “!=” or *val* is not an integer.
   - Same *criName* exist

   The first 5 restrictions are retrieved inside the constructor of class `SimpleCriterion` and throw `IllegalArgumentException` with a specific message to method `newSimpleCri (`class `CVFS)`. The last restriction for renaming will be retrieved and thrown in `newSimpleCri (`class `CVFS)`.

```
//class SimpleCriterion implements Criterion, Serializable
private String criName;
private String op;
private String val;
```

```
public SimpleCriterion(String criName, String attrName, String op, String
val) throws IllegalArgumentException{
   if(!criName.matches("[a-zA-Z]{2}"))
       throw new IllegalArgumentException("Criterion name must contains
exactly two English letters.");

if(!(attrName.equals("name")||attrName.equals("type")||attrName.equals("size"
)))
       throw new IllegalArgumentException("Attribute name must be either
name, type or size.");
   if(attrName.equals("name")){
       if(!op.equals("contains")) throw new
IllegalArgumentException("Attribute is name, Op must be contains.");
       if(!(val.startsWith("\"") && val.endsWith("\""))) throw new
IllegalArgumentException("Attribute is name, val must be a string in the
double quote.");
   }
   if(attrName.equals("type")){
       if(!op.equals("equals")) throw new IllegalArgumentException("Attribute
is type, Op must be equals.");
       if(!(val.startsWith("\"") && val.endsWith("\""))) throw new
IllegalArgumentException("Attribute is type, val must be a string in the
double quote.");
   }
   if(attrName.equals("size")){

if(!(op.equals(">")||op.equals("<")||op.equals(">=")||op.equals("<=")||op.equ
als("==")||op.equals("!=")))
           throw new IllegalArgumentException(("Attribute is size, Op must be
>, <, >=, <=, ==, or !=."));
       if(!val.matches("\\d+")) throw new IllegalArgumentException("Attribute
is size, val must be an integer.");
   }


   this.criName=criName;
   this.attrName=attrName;
   this.op=op;
   this.val=val;
}
--------------------------------------------------------------------------
//class CVFS implements Serializable
      public void newSimpleCri(String criName, String attrName, String op,
      String val){
             if(criterionHashMap.get(criName)!=null) throw new
      IllegalArgumentException("Criterion " + criName + " already exists.");
             criterionHashMap.put(criName, new SimpleCriterion(criName,
      attrName, op, val));
      }
```

[REQ 10]

1) The requirement is implemented.

2) A isDocument() method is used. It simply puts a criterion called “IsDocument” into the hashmap, calling the empty constructor in class IsDocument to create a new instance. The check function will be interrupted later.

3) No error situation.

```
//class IsDocument implements Criterion, Serializable
      (empty constructor)
-------------------------------------------------------------------------
//class CVFS implements Serializable
      public void isDocument(){
            criterionHashMap.put("IsDocument", new IsDocument());
      }
```

[REQ 11] - newNegation

1) The requirement is implemented.

2) The system will first get an existing criterion named `criName2` in criterionHashMap, and reference it by `cri2`. It will then call the constructor in `NegationCriterion` class to create a criterion as a negation to `cri2`, referenced by `criName1`, and put it into `CriterionHashMap`.

3) [Criterion Name Check] `IllegalArgumentException` thrown due to issue below:
   - criName1 does not contain 2 letters.
   - criName2 is not found in the system.
   - Duplicate criName1

   Similar to SimpleCriterion, the first criterion name constraint will be thrown in the constructor of class `NegationCriterion`, and the last 2 constraints will be thrown in the `newNegation` method (class `CVFS`).

```
// class NegationCriterion implements Criterion, Serializable
      private String criName1;
      private Criterion cri2;

       public NegationCriterion(String criName1, Criterion cri2){
            if(!criName1.matches("[a-zA-Z]{2}"))
                  throw new IllegalArgumentException("Criterion name must
                  contains exactly two English letters.");
            this.criName1=criName1;
            this.cri2=cri2;
       }
-------------------------------------------------------------------------
//class CVFS implements Serializable
      public void newNegation(String criName1, String criName2) {
             Criterion cri2 = criterionHashMap.get(criName2);
             if (cri2 == null) throw new IllegalArgumentException("Criterion
      " + criName2 + " doesn't exist.");
```

```
            if(criterionHashMap.get(criName1)!=null) throw new
        IllegalArgumentException("Criterion " + criName1 + " already
        exists.");
            criterionHashMap.put(criName1, new NegationCriterion(criName1,
        cri2));

        }
```

[REQ 11] - newBinaryCri

1) The requirement is implemented.
2) Similar to Negation, the system will first get 2 existing criteria named `criName3,criName4` in criterionHashMap, and reference it by `cri3, cri4` respectively. It will then call the constructor in `BinaryCriterion` class to create a new criterion as `cri3 op cri4`, referenced by `criName1`, and put it into `CriterionHashMap`.

3) [Criterion name check] `IllegalArgumentException` thrown due to issue below:
   - criName1 does not contain 2 letters.
   - *logicOp* is not “&&” or “||”.
   - criName3 or criName4 is not found in the system.
   - Duplicates criName1.

   Similar to Negation, the first 2 constraints are thrown in class BinaryCriterion, and the last 2 exceptions are thrown in method `newBinaryCri` (class `CVFS`).

```
//class BinaryCriterion implements Criterion, Serializable
      public BinaryCriterion(String criName1, Criterion cri3, String
      logicOp, Criterion cri4){
         if(!criName1.matches("[a-zA-Z]{2}"))
             throw new IllegalArgumentException("Criterion name must
      contains exactly two English letters.");
         if(!(logicOp.equals("&&")||logicOp.equals("||")))
             throw new IllegalArgumentException("LogicOp is either && or
      ||");
         this.criName1=criName1;
         this.cri3=cri3;
         this.logicOp=logicOp;
         this.cri4=cri4;
      }
-------------------------------------------------------------------------
//class CVFS implements Serializable
      public boolean newBinaryCri(String criName1, String criName3, String
      logicOp, String criName4){
             Criterion cri3=criterionHashMap.get(criName3);
             Criterion cri4=criterionHashMap.get(criName4);
             if(cri3==null) throw new IllegalArgumentException("Criterion "
      + criName3 + " doesn't exist.");
             if(cri4==null) throw new IllegalArgumentException("Criterion "
      + criName4 + " doesn't exist.");
```

```
            if(criterionHashMap.get(criName1)!=null) throw new
      IllegalArgumentException("Criterion " + criName1 + " already
      exists.");
            criterionHashMap.put(criName1, new BinaryCriterion(criName1,
      cri3, logicOp, cri4));
          }
```

[REQ 12]

1) The requirement is implemented.
2) `toString()` method is defined `Criterion Interface`, and *overridden* by all the classes implementing `Criterion` to facilitate the usage in `printAllCriteria` method.
   `printAllCriteria` first checks the size of `criterionHashMap`. It will print "No criterion accessible" if no occupied space size is seen (no criterion in `criterionHashMap`). The system then iterates through the contents in `criterionHashMap` and prints information on all criteria found.

3) No error situation.

```
//class CVFS implements Serializable
      public void printAllCriteria(){

         if(criterionHashMap.size()==0) System.out.println("No criterion
      accessible.");
         for (String criName : criterionHashMap.keySet()) {
             System.out.println(criName + ": " +
      criterionHashMap.get(criName).toString());
         }
      }
-------------------------------------------------------------------------
      //class SimpleCriterion implements Criterion, Serializable
      public String toString(){
         return attrName +" " + op +" "+ val;
      }
-------------------------------------------------------------------------
      //class NegationCriterion implements Criterion, Serializable
      public String toString(){
         return "NOT" + " " + cri2.toString();
      }
-------------------------------------------------------------------------
      //class BinaryCriterion implements Criterion, Serializable
      public String toString(){
         return cri3.toString() + " " + logicOp + " " + cri4.toString();
      }
```

[REQ 13]

1) The requirement is implemented.

2) `checkFile(File file)` is defined in `Criterion Interface`, and *overridden* by all the classes implementing `Criterion` to facilitate the usage in `search` method. In this step, we view ***both*** `Document` and `Directory` as `File` to check whether they fit certain criteria (name, size, type), and didn't go inside the subdirectories if any. (As the requirement is to "list all the files ***directly*** contained in the working directory which fit the criterion".)

   An empty array with variable size was designed to place specific criteria that meet the requirements *criName* in. The system will iterate through all files in the current working directory, calling the method `checkFile` in the corresponding class to check whether the file fits the requirements. With the criterion input, the method will return the list of filtered files ***directly*** contained in `workDir` that meet the criterion, as well as output the total number of these files and their cumulative size.

3) [Criterion Name Check]
   If the criterion-referenced by `criName` doesn't exist in the criterionHashMap, we throw `IllegalArgumentException` with a specific error message.

```
//class CVFS implements Serializable
public List<File> search(String criName) throws IllegalArgumentException{
   List<File> files = new ArrayList<>();
   int size=0;
   Criterion criterion =  this.getCriterionHashMap().get(criName);
   if(criterion==null) throw new IllegalArgumentException("The criterion
doesn't exist.");
   for(File file : this.curWorkDir.getFiles()){
       if(criterion.checkFile(file)){
           files.add(file);
           size+=file.getSize();
       }
   }
   int number = files.size();
   System.out.println("Total number of files: " + number + ";" + " Total size
of files: " + size + ".");
   return files;
}
--------------------------------------------------------------------------
//class SimpleCriterion implements Criterion, Serializable
      public boolean checkFile(File file) {
         switch (attrName) {
             case "name":
                 String val2=val.substring(1, val.length() - 1);
                 return file.getName().contains(val2);
             case "type":
                 if(file instanceof Directory) return false;
                 String val3=val.substring(1, val.length() - 1);
                     Document document =(Document) file;
                     return document.getType().equals(val3);
             case "size":
                 int size = Integer.parseInt(val);
```

```
                int fileSize = file.getSize();
                switch (op) {
                    case ">":
                        return fileSize > size;
                    case "<":
                        return fileSize < size;
                    case ">=":
                        return fileSize >= size;
                    case "<=":
                        return fileSize <= size;
                    case "==":
                        return fileSize == size;
                    case "!=":
                        return fileSize != size;
                }
            }
            return false;
        }
```

```
--------------------------------------------------------------------------------
//class IsDocument implements Criterion, Serializable
public boolean checkFile(File file){ //override
   return file instanceof Document;
}
--------------------------------------------------------------------------------
//class NegationCriterion implements Criterion, Serializable
public boolean checkFile(File file) { //override
   return !cri2.checkFile(file);
}
--------------------------------------------------------------------------------
//class BinaryCriterion implements Criterion, Serializable
public boolean checkFile(File file){//override
   switch(logicOp){
       case("&&"):
           return cri3.checkFile(file) && cri4.checkFile(file);
       case("||"):
           return cri3.checkFile(file) || cri4.checkFile(file);
   }
   return false;
}
```

[REQ 14]

1) The requirement is implemented.

2) The public method will first get the criterion in hashmap, and call the private helper method to search for documents that satisfy the criterion. We check documents as usual. If the subdirectory is ***empty***, we view it as a file to check, else if the subdirectory contains other files, we ***look inside*** to check the inner files ***recursively***. With the criterion input, the method will return the list of filtered files ***directly and indirectly*** (in the inner directories) contained in `workDir` that meet the criterion, as well as output the total number of these files and their cumulative size.

3) [Criterion name check]
If the criterion-referenced by `criName` doesn't exist in the `criterionHashMap`, we throw `IllegalArgumentException` with a specific error message.

```
//class CVFS implements Serializable
    public List<File> rSearch(String criName)throws
    IllegalArgumentException{
       List<File> files = new ArrayList<>();
       Criterion criterion =  criterionHashMap.get(criName);
       if(criterion==null) throw new IllegalArgumentException("The
    criterion doesn't exist.");
       int size = rSearch(criterion, files, this.curWorkDir, 0);
       int number = files.size();
       System.out.println("Total number of files: " + number + "; Total
    size of files: " + size + ".");
       return files;
    }
    private int rSearch(Criterion criteria, List<File> files, Directory
    curDir,int size){
       if(curDir.getFiles().size()==0) return 0;
       for(File file : curDir.getFiles()) {
           if (file instanceof Directory && ((Directory)
    file).getFiles().size() != 0)
               size += rSearch(criteria, files, (Directory) file, 0);
           else if (criteria.checkFile(file)) {
               files.add(file);
               size += file.getSize();
           }
       }

       return size;
    }
```

[REQ 15]

1) The requirement is implemented.

2) The system will first create a `FileOutputStream` and `ObjectOutputStream` to write the CVFS that the user wants to save into a specific file path. Upon completion, both streams will be closed to make sure all data has been written. The system will print a confirmed message upon successful operation.
We create a new class called `Controller` to save and load the whole CVFS instance. All the related classes `implement Serializable` to ensure the successful completion of save and load.

3) For any error throughout the saving operation, an exception will occur by printing "Error saving virtual disk: " with the error message.

```
    public void save(String filePath) {
       //Save the working virtual disk into the file at path.
       try {
           FileOutputStream fileOut = new FileOutputStream(filePath);
```

```
            ObjectOutputStream objectOut = new ObjectOutputStream(fileOut);
            objectOut.writeObject(cvfs);
            objectOut.close();
            fileOut.close();
            System.out.println("CVFS saved successfully.");
        } catch (Exception e) {
            System.err.println("Error saving virtual disk: " +
e.getMessage());
        }
```

[REQ 16]

1) The requirement is implemented.

2) The system will first create a `FileInputStream` and ObjectInputStream to read the CVFS that the user wants to load into the current working directory from a specific file path. This method is designed as static and returns a CVFS instance. The system will print a confirmed message upon successful operation.

3) For any error throughout the saving operation, an exception will occur by printing “Error saving virtual disk: “ with the error message.

```
public static CVFS load(String filePath) {
    //Load a virtual disk from the file at path and make it the
working virtual disk.
    try {
        FileInputStream fileIn = new FileInputStream(filePath);
        ObjectInputStream objectIn = new ObjectInputStream(fileIn);
        System.out.println("CVFS loaded successfully.");
        return (CVFS) objectIn.readObject();
    } catch (Exception e) {
        System.err.println("Error loading virtual disk: " +
e.getMessage());
        return null;
    }
}
```

[REQ 17]

1) The requirement is implemented.

2) The system will terminate. We also make flag `running` to `false` in `CommandLineInterface.java`

3) No error situations

```
public void quit(){
    //Terminate the execution of the system.
    System.exit(0);
}
```

[Bonus 1]

1) The requirement is implemented.

2) As all the related classes `implement Serializable` (including classes in package criterion), and we create a class controller to contain the save and load method towards the **whole** cvfs instance, the criteria inside it will also be saved and loaded successfully.

3) Error handling is contained in req15, 16.

[Bonus 2]

1) The requirement is implemented.

2) We create a new package called command. In class CommandManager, we utilise 2 stack structures to manipulate user's commands. We design an interface Command, and 8 classes, newDocCommand, newDirCommand, newNegationCriCommand, newSimpleCriCommand, renameCommand, deleteCommand, changeDirCommand implements the interface, overriding the redo() and undo() methods.

   6 relative methods in class CVFS will create a new command and put it into a commandManager, which is initialized in the constructor of CVFS. An example in newDoc method is shown below. And two methods redo() and undo() are also created in CVFS, directly calling the corresponding methods in commandManager.

```
Command command = new newDocCommand(this, newDoc);
commandManager.putinto(command);
```

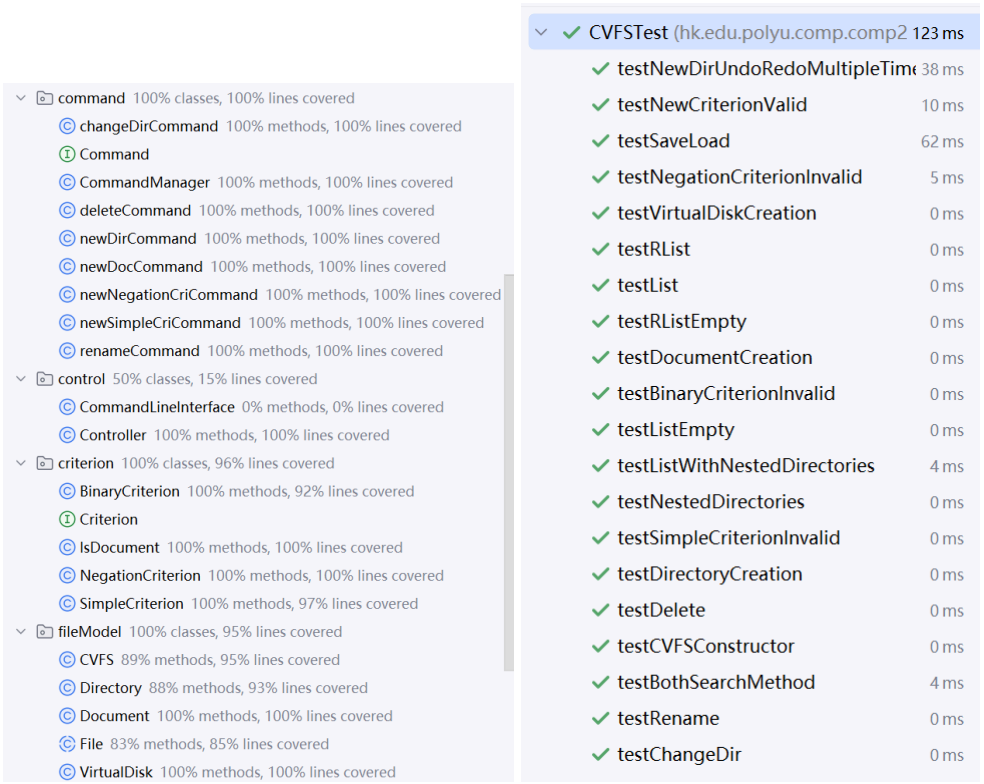
   The more detailed implementation process is shown in the code

3) No error situations.

2.3 Test

Please note that some errors may occur in testing in different devices is due to the difference of newline character (`println`) defined by the different operating systems (MacOS: `\r\n` ; Windows: `\n`)!!!

We design the test for all classes and methods in these classes(only excluding CommandLineInterface.java and Application.java).
Line coverage: >90%



3 Reflection on MyLearning-to-Learn Experience

3.1 Reflect on our learning-to-learn experience in the project

During the first discussion of our project, we decided to first construct a simple structure of the project so that the requirements of source code are divided based on the relatedness. Parts were separated into Req 1 to 8, Req 9 to 14 and Req 15 to 17, based on the overview of the system, criterion and interaction of the system with the local file. This was planned to maximise our team collaboration to ensure the best effectiveness.

Nevertheless, we came to know that we were too naive as the simple structure did not contain enough information and details to ensure the smoothness of the code running after combining all of our codes together. We concluded that code smoothness could not be implemented by doing it separately. Hence, the continuous work division would be separated into the compilation of code with the structure that was done earlier, working on the bonus questions and writing the project report.

What we learned in class was insufficient to complete the full program. For example, we encounter difficulties when completing save and load tasks. We didn't initially realize that successful compression requires making the associated class implement Serializable, and needs empty constructors in some classes. We search on the Internet to find and fix the bugs.

3.2 Plan to improve our self-learning approaches

A better documentation and detailed plan alongside with more communication throughout the coding process could help in improving our code stimulation in the first place. If

throughout documentation for the code and the structure could be updated once any members have new ideas, it could be discussed on-time to avoid larges mistakes which would lead to major issues in the code compilation later. Other members could also receive the latest changes for the project so changes could be made on time instead of after the entire completion of the code.

3.3 Use of GenAI

1. We used AI to help us build an initial framework for the system, which we then combined with our own ideas to make the construction of the CVFS system more neater and clearer. The original structure has changed a lot in the subsequent completion process.

2. Some small techniques which are not explicitly delivered in class. We interpret with GenAI to facilitate our coding. For example, the technique to check whether a string is 2 English letters.
   ```
   if(criName1.matches("[a-zA-Z]{2}"))
   ```

3. It is used to solve trivial and easily overlooked problems, such as the correct use of line breaks.

4. Used for the save and load requirements. We are not familiar with saving and loading to the local path, thus ask GenAI for help and interrupt it to fix bugs.