

CSE 332S Lab 5: OOP Design

Miscellaneous details:

Due Date: Sunday, May 9th at 11:59 PM CT, there will be no extensions given on this lab! **If you are submitting late, you must email me and let me know.**

Grade breakdown: This lab accounts for 20% of your final semester grade, 25% total for lab 5 and studios 16-21

Group details: You will work in the same groups as you did for Studio 16-21. Groups of 1-3 students are allowed.

Getting started: Continue working in your repository for studios 16-21 and lab 5. Open and work in the "Lab5.sln" project.

Note: All of the header and source files you will modify throughout studios 16-21 and lab 5 are stored in a common location, the SharedCode folder inside of your repository. All of the 6 studio solutions link to the same files, so when you edit a file in studio 16, those files will be updated for studio 17 as well. No setup should be necessary. One side effect of this is that as you modify files in later studios, some tests from previous studios will fail or break. This is ok. You do not need to worry about the tests from previous studios once that studio is complete.

Lab overview:

This lab will build upon studios 16-21. You should complete all studios before starting the lab. Throughout studios 16-21 and continuing into lab 5, you will build a software simulation of a file system, some simple file types that may be stored in a file system, and a user interface similar to a command prompt or terminal that allows a user to interact with the file system and files it contains. Throughout the design and implementation of the file system, OOP design principles and patterns learned throughout the semester will be used to ensure the file system is easy to extend with new functionality in the future, easy to modify without major code refactoring, and easy to configure with different functionality as needed. Here is a quick overview of what you have done in studio already:

1. **Studio 16: Creating a set of related classes via interface inheritance.** A file system stores files of many different types. In this studio, you created an interface that declares the basic functionality all files share (read, write, append, getSize, getName). You then defined a couple of concrete file types that inherit this interface and define it appropriately for the given file type (TextFile, ImageFile (studio 17)). This creates a set of concrete file classes that share a common interface (AbstractFile).
2. **Studio 17: Programming to an interface.** Now that you have some concrete file types, you need a way to store and manage access to files. This is the file system's job. This

studio introduces a new interface describing the functionality all file systems share (createFile (moved in studio 18), addFile, openFile, closeFile, removeFile) and introduces an implementation of this interface (SimpleFileSystem). The SimpleFileSystem may store and manage access to many different file types. To support this, the file system is programmed to use the AbstractFile interface. A file system stores AbstractFiles and interacts with AbstractFiles. As concrete classes that inherit and define the AbstractFile interface are subclasses of AbstractFile, objects of those classes may be used freely with the file system. New file types can easily be added to the file system by creating new concrete file classes that inherit the AbstractFile interface.

3. Studio 18: Single responsibility principle and the abstract factory design pattern.

This studio separates the task of creating files from the file system object. A file system should simply be responsible for storing and managing access to files, not creating them. A new object, a file factory, will be responsible for creating files instead. The abstract factory pattern is used to support extensibility and flexibility in our design. The AbstractFileFactory interface declares an interface for creating files and concrete file factory objects define that interface to handle the creation of objects of different concrete file types. The abstract factory pattern provides extensibility by making it easy to create new concrete file factory classes (as in studio 20). Each concrete file factory may enforce restrictions on what types of files it can create, or can vary how the files are actually created. The pattern supports flexibility as a client that creates files using the AbstractFileFactory interface may be configured with any concrete factory class that inherits from AbstractFileFactory. This allows us to easily configure a client to change what types of files it may create or how those files are created.

4. Studio 19: Adding new functionality to existing file types via the visitor pattern.

This studio introduces the visitor pattern. The visitor pattern allows us to add new functionality to an existing set of related classes, where each concrete class may require a different implementation. For instance, a file may be displayed in different ways. Maybe we want to display the bytes contained in the file directly, without any special formatting; maybe we want to display metadata about the file only; or, maybe we want to display the file in a format specific to that concrete file type. Rather than adding a virtual member function to the AbstractFile interface for each of these display methods (such as readBytes(), readMetadata(), readFormatted()) and overriding those methods in an appropriate way in each derived file class, we can instead use the visitor pattern to accomplish this without cluttering the AbstractFile interface. After completion, you will now have a couple of concrete visitors. One prints the contents of a file in a format specific to that file type, the other displays the metadata of the file.

5. Studio 20: Password protected files using the Proxy pattern. This studio introduces the Proxy design pattern to start to build support for password protected files. A proxy object can be stored in place of a real file in the file system. Any attempts to read, write, or visit a file will require the user to enter the correct password for the file before gaining access.

6. Studio 21: Adding a user interface and the command pattern. Up to this point, testing/interacting with the file system has been done via writing code in main. This

studio creates a user interface (CommandPrompt) that allows a user to interact with the file system by issuing commands. To implement this functionality, the command pattern is used. Each action the user can request is implemented as a ConcreteCommand object. When a user provides input, the CommandPrompt object invokes the appropriate command based on the user's input. The touch command is introduced in this studio. Touch is a command that creates a new file and adds the file to the file system.

What you will implement in lab 5: For lab 5, you will be creating a few additional commands to increase the functionality of the program. The commands you will modify or implement are listed below, you will find further details on the specification and requirements of each command later in this document. In lab 5, you will:

1. Create and implement a command called "ls" to list all of the files stored in the file system
2. Create and implement a command called "remove" to remove a file from the filesystem
3. Modify the "touch" command to support creating password protected files
4. Create and implement a command called "cat" that concatenates or writes user input into a file
5. Create and implement a command called "ds" that displays the contents of a file based on the file's type
6. Create and implement a command called "copy" that makes a copy of an existing file and stores it in the file system with a different name (**prototype pattern**)
7. Create and implement a command called "rename" that renames an existing file in the file system (**composite pattern, strategy pattern**)

And finally, the details...

Modify, or create and implement the following commands as described below. All commands you add should be ConcreteCommand objects as part of the command pattern. You should name your classes exactly as they are typed below when highlighted in blue. Header and source files for your classes should already be created and linked in the solution.

1. **Create and implement LSCommand (15 points):** The LSCommand will be invoked by the user by typing "ls" into the command prompt. It should output to the terminal the names of all files currently in the file system. To support this, you should first add a function to the AbstractFileSystem interface called "getFileNames" that takes no parameters and returns a `std::set<string>` containing the names of all files in the file system. Implement this function in SimpleFileSystem. You can assume all file names are less than 20 chars total (you can enforce this in "touch" if you would like). The output of the "ls" command should look as below (assume we have 5 files in the file system currently: file.txt, image.img, other.txt, file2.txt, image2.img):

```
$ ls          // the command given by the user
file.txt      file2.txt      // output of the command
image.img     image2.img
```

other.txt

More specifically, 2 files should be printed per line and the file names should be evenly spaced and aligned appropriately.

The “ls” command should also support an additional option, “-m”, which will display a single file per line along with the metadata associated with that file as below (sizes are made up):

```
$ ls -m
file.txt      text      15
file2.txt     text      10
image.img     image      4
image2.img    image      9
other.txt     test       0
```

When the command is executed, it should return 0 if the command executes successfully, or some non-zero value if the command fails for some reason.

Hint: You may already have a visitor that can help with this. Given a file name, how would you obtain a pointer to the file so that you can “visit” it?

Other Hint: When the CommandPrompt executes a command, what does it pass to the command’s execute function if the user enters “ls”? What about “ls -m”? Use the debugger to figure this out if you need to.

2. **Create and implement RemoveCommand (5 points):** This command should remove the file with the provided name from the file system. If the file is unable to be removed for some reason, the command should return a non-zero value indicating this. Otherwise, it should return 0 if the file was removed successfully. The command should be invoked by the user with “rm <filename>”, such as below:

```
$ ls
file.txt      image.img    // file system contains these
files
$ rm file.txt // rm one of the files
$ ls
image.img     // file.txt was successfully removed
$              // ready for the next command from the user
```

3. **Modify your TouchCommand from studio 21 to support creating password protected files (10 points):** Update “touch” with an option to create a file that is password protected. When touch is invoked with the “-p” option, a password protected

file should be created rather than a regular file. You must use your Proxy from studio 21 to support this. The touch command should create the file, prompt the user for a password, and then set up the proxy to protect the file. Then add the proxy to the file system. An example is below:

```
$ touch file.txt      // creates file.txt and adds it, no password
$ touch file1.txt -p  // -p comes after the filename
What is the password? // prompt the user for a password
1234                  // user input: a not very safe password
$
```

After implementing cat and ds below, you can verify the password protection is working correctly (cat and ds will write/read the file).

4. **Create and implement CatCommand (15 points):** Add a new command to your program called cat. If you are unfamiliar with linux command line utilities, cat is a utility that is useful for concatenating files. The cat command can be used to write to a file as well, which will be the purpose of our cat command. Cat can be invoked from the command prompt as follows:

```
$ cat <filename> [-a]
```

<filename> will be replaced with the name of a real file in the file system. The '[' brackets around "-a" indicate "-a" is optional. So, a real invocation of the command may look like:

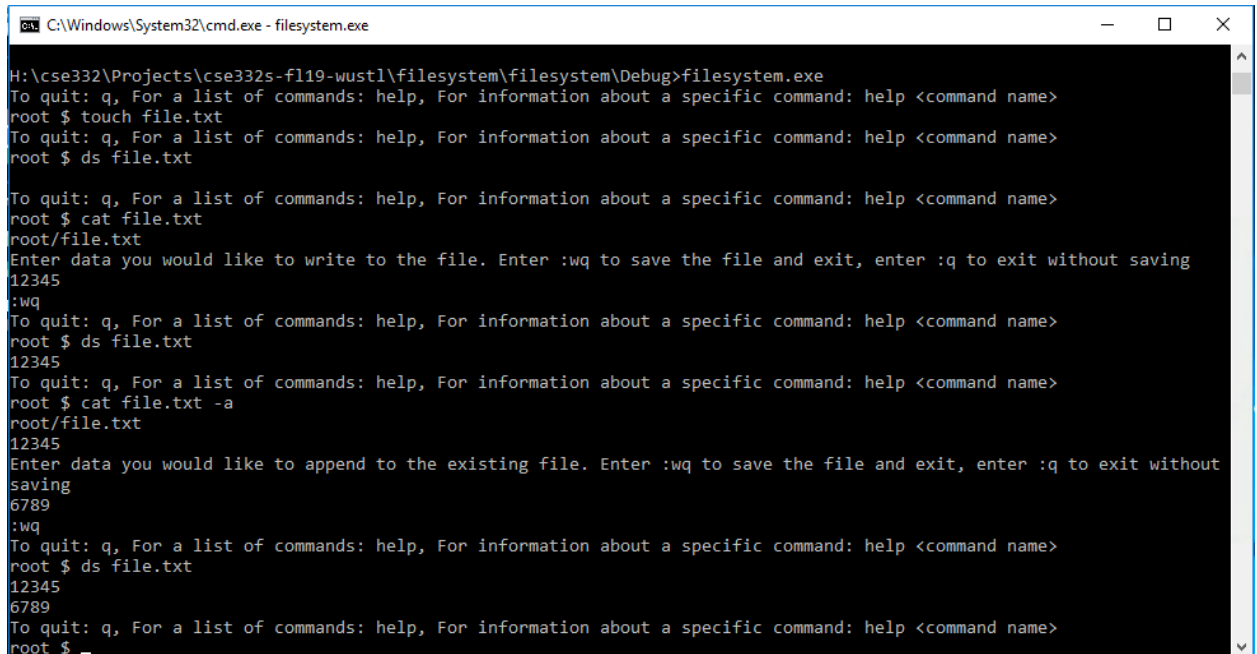
```
$ cat file.txt      // or
$ cat file.txt -a
```

The cat command should be defined to do the following: the -a option stands for append. If the -a option is given, the current contents of the file should be displayed (the bytes only, not the formatted output) followed by a new line. The user should then be prompted to input data to append to the file, to input ":wq" to save and quit, or to input ":q" to quit without saving. The users input should be read from cin line by line. If the line is not ":wq" or ":q", the data should be saved temporarily. Remember getline() will trim off the newline character ('\n'), make sure to reinsert a new line character between each line of user input when saving it. If the user enters ":q", the cat command should return and no data should be written to the file. If the user enters ":wq", the input provided up until the user entered ":wq" should be appended to the file.

If the user invokes cat without the -a option, the functionality should be the same as above except the current contents of the file should not be displayed to the user before prompting for input and when the user provides ":wq" as input, the contents of the file should be overwritten with the data supplied by the user, rather than appended to the

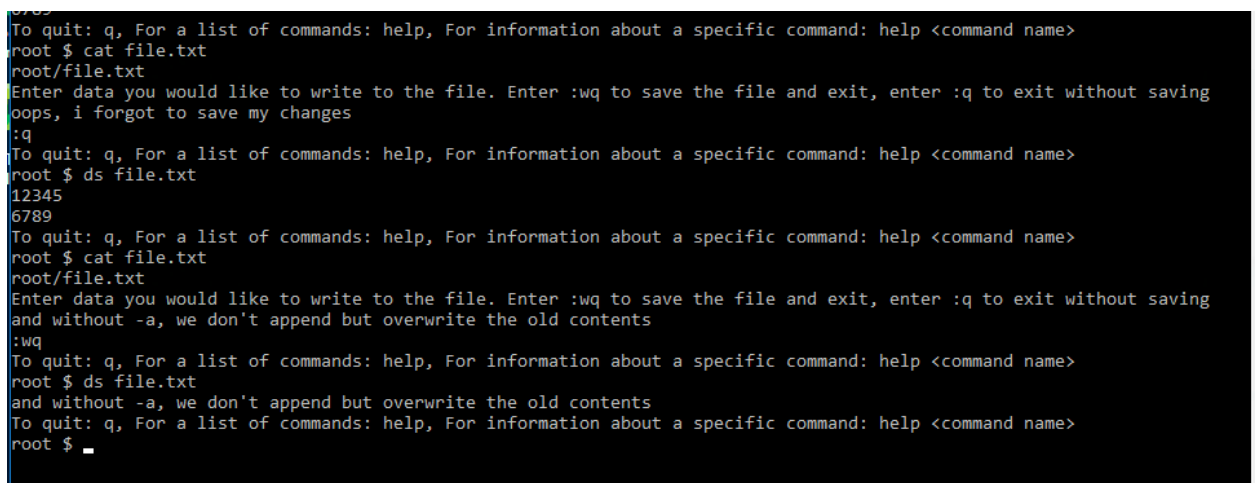
end of the file. Some example invocations and possible output are below (note “ds” is a command you will implement later, it displays a file):

Note: These images show the functionality from last semester where we supported directories and hierarchical file storage. The output may look a bit different (root \$ instead of \$ only, root/file.txt instead of file.txt, etc..). Ignore those minor differences and the images show the correct functionality.



```
C:\Windows\System32\cmd.exe - filesystem.exe
H:\cse332\Projects\cse332s-fl19-wustl\filesystem\filesystem\Debug>filesystem.exe
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ touch file.txt
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ds file.txt
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cat file.txt
root/file.txt
Enter data you would like to write to the file. Enter :wq to save the file and exit, enter :q to exit without saving
12345
:wq
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ds file.txt
12345
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cat file.txt -a
root/file.txt
12345
Enter data you would like to append to the existing file. Enter :wq to save the file and exit, enter :q to exit without saving
6789
:wq
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ds file.txt
12345
6789
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $
```

And continued on..



```
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cat file.txt
root/file.txt
Enter data you would like to write to the file. Enter :wq to save the file and exit, enter :q to exit without saving
oops, i forgot to save my changes
:q
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ds file.txt
12345
6789
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cat file.txt
root/file.txt
Enter data you would like to write to the file. Enter :wq to save the file and exit, enter :q to exit without saving
and without -a, we don't append but overwrite the old contents
:wq
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ds file.txt
and without -a, we don't append but overwrite the old contents
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $
```

As with other commands, 0 should be returned if the command executes successfully, otherwise a non-zero value should be returned indicating an error occurred.

5. **Create and implement DisplayCommand (10 points):** Add a new command, Display. Display is invoked with “ds”. Display opens a file and displays its contents, either

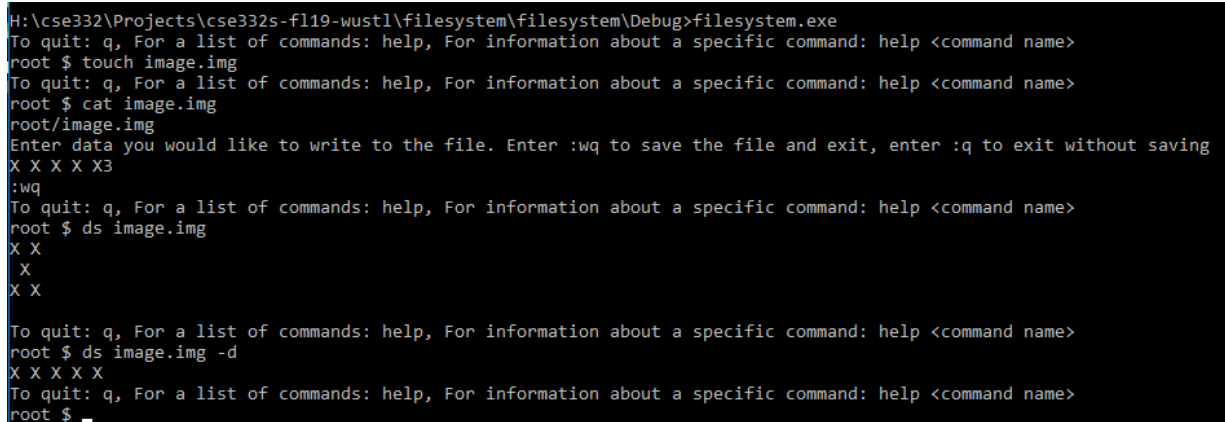
formatted or not (when given the “-d” option for data only). An example invocation could be:

```
$ ds image.img          // formatted
```

Or..

```
$ ds image.img -d       // unformatted
```

And here is a screenshot of it in action:



```
H:\cse332\Projects\cse332s-fl19-wustl\filesystem\filesystem\Debug>filesystem.exe
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ touch image.img
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cat image.img
root/image.img
Enter data you would like to write to the file. Enter :wq to save the file and exit, enter :q to exit without saving
X X X X X3
:wq
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ds image.img
X X
X
X
X X

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ds image.img -d
X X X X X
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $
```

If display executes successfully, 0 should be returned. Otherwise, a non-zero value should be returned indicating an error occurred.

6. **Create and implement CopyCommand (15 points):** The copy command will copy a file that exists in the file system and add the copy to the file system with a different name. It is invoked with the following command structure:

```
cp <file_to_copy> <new_name_with_no_extension>
```

Where file_to_copy is replaced with a real file that exists in the file system and new_name_with_no_extension is replaced with the name a user would like the copy to be called (minus the extension). The copy will be of the same type as the original file, so the file extension should match as well. When the copy is made, the correct extension should be added to the file name given by the user. The original file, and the copy of the file should be unique file objects. **To receive credit for this command**, you must implement the **prototype pattern** in order to copy a file object. If any errors occur while copying or adding the copy to the file system, a message notifying the user the command failed should be displayed and the copy of the file should be deleted. Here is a transcript of copy in action (kind of, I have not implemented this version of the lab yet, so some output is missing but this shows the general commands to run):

```
$ touch file.txt        // create file.txt
$ cat file.txt          // write to it
```

```

123456
:wq
$ cp file.txt file_copy // create a copy
$ ds file_copy.txt // .txt added by the cp command
123456 // ensure it is a copy
$ cat file_copy.txt
Hello
:wq
$ ds file.txt
123456 // ensure the copy is a unique file, writing to
// the copy does not change the original

```

If the file is copied successfully and added to the file system, executing this command should return 0 for success. Otherwise, a non-zero value should be returned.

How should a password protected file be copied? The copy should be password protected as well. So, not only does the file need to be copied via the prototype pattern, but the proxy for the file needs to be copied as well.

Hint: You will need to change the name of the copy to its new name, however there is no setName function in the AbstractFile interface. You should not add a setter for the file name. Instead, it may be helpful to pass a string holding the new file name to the clone function as part of the prototype pattern.

7. **Create and Implement support for MacroCommands (10 points):** Macro commands will allow us to construct commands out of other commands. Executing a macro command simply executes each of the commands it is composed of in order. To implement this, you should first create a class called **MacroCommand** that inherits the AbstractCommand interface and maintains as member variables a std::vector of AbstractCommand objects (the primitive commands the macro command will use) and a pointer to an **AbstractParsingStrategy** (This will be described shortly). In our project, the macro command's execute function takes in an input string, however we can't just pass that same string to each individual command the macro command is composed of, as they expect different input. The job of a ParsingStrategy will be to take the input provided to the macro command's execute function and transform it into a vector<string>, where each string in the vector is the input that should be supplied to the corresponding individual command. MacroCommand's execute function should be implemented as follows:
 - a. Ask the ParsingStrategy object to parse the input provided to the macro command's execute function, which will return a vector<string>. See below.
 - b. For each individual command the MacroCommand is composed of, call the individual command's execute function with the corresponding input from the vector<string> returned by the ParsingStrategy. If any individual command

returns an error, return an error. Otherwise, if all individual commands execute successfully, return 0 for success.

As a MacroCommand is a composite object in the composite pattern, it needs to provide a function for adding individual commands to its vector of commands. Add a function to its public interface called `addCommand` that takes a pointer to an AbstractCommand and pushes it to the end of its vector of commands. Add a function to set the macro command's parsing strategy as well. This should be called `setParseStrategy` and should take a pointer to an AbstractParsingStrategy as a parameter.

Declare the `AbstractParsingStrategy` interface. An AbstractParsingStrategy has a single public pure virtual member function called `parse` that takes a `std::string` parameter and returns a `std::vector<string>` by value.

- 8. Add the rename command (10 points):** To receive credit, this command must be implemented as a MacroCommand. The rename command will change the name of an existing file (you cannot change it directly, this must be implemented as a MacroCommand or you will get 0 credit). Rename can be invoked by the user with "rn" as below:

```
$ rn <existing_file> <new_name_with_no_extension>
```

Where `<existing_file>` is replaced with the name of an existing file and `<new_name_with_no_extension>` is replaced with the name the user would like the file to be called (a file extension like .txt or .img will be tacked on to the name provided by the CopyCommand). Renaming a file should do two things: First, copy the file with the copy command giving the copy the correct new name. Second, remove the original file with the remove command. To implement a MacroCommand with this functionality you will need to:

- Implement a concrete class called `RenameParsingStrategy` that defines the interface provided by AbstractParsingStrategy. The interface should be defined appropriately so that given an input string like:

```
<existing_file> <new_name>
```

The parse function will return a vector containing the strings:

```
<existing_file> <new_name> // in index 0, used by the copy command
```

```
<existing_file> // in index 1, used by the remove command
```

- Update main to create a MacroCommand object configured with a RenameParsingStrategy object as its AbstractParsingStrategy and a CopyCommand as well as a RemoveCommand object as its command objects. Add the MacroCommand to the CommandPrompt so it will be invoked when the user provides "rn" as input

- 9. Updating main and testing (10 points):** update main to configure the command prompt with each of the above commands. Test your commands thoroughly. In your ReadMe.txt

file, document the tests you ran as well as any errors/bugs you encountered while working. Also, at the top of your Readme.txt, list each group member's name and describe how the work was split between the group members. Note: Make sure to avoid memory leaks, double deletions, etc.

10. **Extra credit (5 points):** Implement one of the following commands

- a. GrepCommand - invoked with:

```
$ grep <string_to_search_for>
```

Grep should search for the provided string in all files stored in the file system. Grep should output a list of all file names that contain the given string. Image files should not be searched, only text files. You should implement a new visitor to accomplish this.

- b. A command that is invoked with "chmod" followed by a file name and a '+' or a '-'. If a '-' is provided, the file should be made read only. The proxy pattern should be used for this. If a '+' is provided, the file should be made writable again.

```
$ chmod file.txt -      // makes file.txt read only
$ chmod file.txt +      // makes file.txt writable again
```

- c. Come up with another MacroCommand and implement it and add it to your CommandPrompt

- i. Some useful command ideas I can think of:

1. Cat + ds - edit a file and then display it afterwards to see the edits.
2. Touch + cat - create a file and edit it immediately
3. Even crazier: touch + cat + ds