CSE 332S Lab 5: OOP Design

Miscellaneous details:

Due Date: Thursday, December 12th at 11:59 PM

Grade breakdown: This lab accounts for 20% of your final semester grade

Group details: You may work in groups of up to 3 students total (you + 2 others). As with lab 4, one of you should create a team when accepting the lab 5 assignment. This will create a repository for your team. The remaining team members should accept the assignment, but should join the team created previously by rather than creating a new team. This will ensure all team members have access to the private repository created for your group.

Getting started: Here is the link to the lab 5 assignment. Please review the Group details above before getting started. Ensure each group member joins the correct team as well. Send me an email(shidalj@wustl.edu) including your team name and a list of all group members.

Lab overview:

This lab will build upon studios 16-21. You should complete all studios before getting started on the lab. To encourage this, no extensions will be offered for this lab unless all studios (16-21) are completed by December 5th. 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: Directories and the composite pattern.** Currently, the simple file system implementation is limited. It contains a single level of files only. File systems are much more powerful, efficient, and easier to manage when stored in a hierarchical way. This is accomplished with directories. Directories are a concrete file type that may contain other files. The contents of a directory are just a list of file names the directory contains (and in a real file system, pointers to where those files are stored on disk). This studio introduces directory files and models them using the composite pattern. The contents of a directory file are as described earlier, a list of file names the directory contains. Rather than storing a pointer to the file in the contents of the directory directly, pointers to files contained in the directory are stored as children of the directory. Parent information is also added to each concrete file type. This will allow us to not only traverse the file system from top down, but also from a file up to its parent directory. A new concrete file

system (HierarchicalFileSystem) is implemented as well. The HierarchicalFileSystem is able to manage a file system containing directories by supporting functionality for parsing a path to a file such as:

root/directory1/directory2/file.txt

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 executing a set of commands as well as storing the current directory a user is working in (the current working directory, CWD). 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.
7. **A few commands I implemented for you in the base code:** The lab 5 base code you will receive is my implementation of the studios as well as a few other commands (your answers to the studio questions will be read thoroughly to ensure you completed the studios and don't simply copy the base code). Aside from touch, I have implemented a few additional commands for you. Those commands are described here:
    a. RemoveCommand - removes a file from the file system
    b. LSCommand - lists the contents of the CWD
    c. CDCommand - changes directories. ".." represents the parent directory. So, the command "cd .." will update the CWD to the parent of the current CWD

What you will implement in lab 5: For lab 5, you will be modifying a couple of the commands I have written for you to extend their functionality a bit. You will also 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. Modify the LSCommand to take an additional option that displays metadata information about each file in the CWD
2. Modify the HierarchicalFileSystem to enforce that non-empty directories cannot be removed. Then modify the RemoveCommand to implement an option that recursively removes all files contained within a directory and then removes the directory
3. Create and implement a command called "cat" that concatenates user input onto a file
4. Create and implement a command called "ds" that displays the contents of a file based on the file's type
5. Create and implement a copy command. Copy makes a copy of an existing file and stores it somewhere else in the file system
6. Create and implement a command for creating symbolic links.

And finally, the details…
Modify, or create and implement the following commands as described below.

While working, you **must obey the following restrictions** in order to avoid a large deduction:

- you may not update the access control level of any functions in the AbstractFile interface. getChild, addChild, removeChild, getParent, setParent are functions that are meant to be accessible by the file system objects only. The AbstractFile class should not friend any command classes, visitor classes, or factory classes to ensure this. The AbstractFile class should not friend the Command Prompt class as well.
- All commands you add should be ConcreteCommand objects as part of the command pattern.

1. **Modifying the LSCommand:** when given the option "-l" the LSCommand should display a "long" listing for each file in the current working directory (CWD). For example, if the CWD is "root" and a file named "file.txt" with a size of 15 chars (15 bytes) is in the root directory along with a file named "image.img" (with a size of 4 bytes) and a directory named "dir" that is empty, you should see the following output:
```
root $ ls        // the command given by the user
file.txt         // the output
image.img
dir

root $ ls -l
file.txt   text        15
image.img  image       4
dir        directory   0
```
   Hint: you may already have a visitor that can help with this.

2. **Modifying Remove and the HierarchicalFileSystem:** Removing a directory from the file system should not be allowed if the directory contains other files. Modify the HierarchicalFileSystem class's removeFile method so that an error is returned if the file being removed is a directory that is not empty. You may find `dynamic_cast` useful here to help identify if an AbstractFile * is pointing to a DirectoryFile object.

   Now modify the RemoveCommand so that when given an option "-r", the file being removed and all the files it contains are removed from the file system. The file system won't allow you to remove non-empty directories, so you will need to recursively open files that are contained within a directory and remove each of them first. Once the directory is empty, it can be removed from the file system without error. If any file in the directory can not be removed for any reason, the directory should not be removed either. However, all other files in the directory should still be removed if possible. As an example, let’s say we have a directory “dir” stored in the root directory. Inside of “dir” we have a file called “file.txt” and a file called “image.img”. The output for the following commands should be similar to:

```
root $ rm dir               // command
unable to remove non-empty directory         // output
command failed
```

Let’s say the file “file.txt” is currently open by another application (we can simulate this by modifying the main function to create dir, file.txt, and image.img manually, adding them to the file system, and then opening file.txt before calling run on the commandPrompt object).

```
root $ rm dir -r            // add the recursive option
unable to remove non-empty directory
command failed

root $ cd dir               // but if we examine the directory dir
root/dir $ ls dir
file.txt                    // we see image.img was not open, so it was removed
```

Now if “file.txt” and “image.img” are both closed.

```
root $ rm dir -r            // recursively remove dir
root $ ls                   // dir is gone, including all files it contained
```

Hints and suggestions: You may find a visitor helpful here. It is easy to know exactly what file type is being visited by a visitor. Also, don’t forget that you can store additional state(member variables) within a concrete visitor if needed. This may be useful in determining if all children of a directory were successfully removed.

3. **Adding the CatCommand:** Add a new command to your program called cat. When a user invokes the cat command from the command prompt, your computer should meow wildly. How you implement this is up to you.

   No, I’m kidding. 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:

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

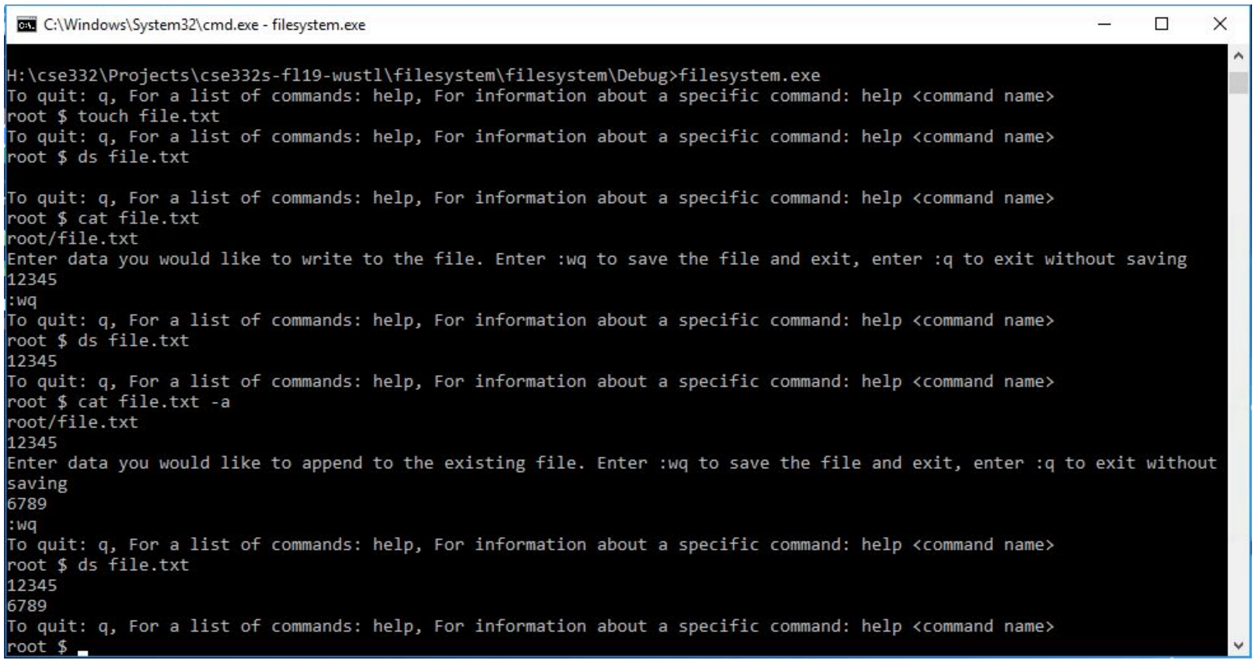
   <filename> will be replaced with the name of a real file in the CWD. The ‘[‘ braces around “-a” indicate “-a” is optional. So, a real invocation of the command may look like:

   ```
   root $ cat file.txt      // or
   ```

```
root $ cat file.txt -a
```

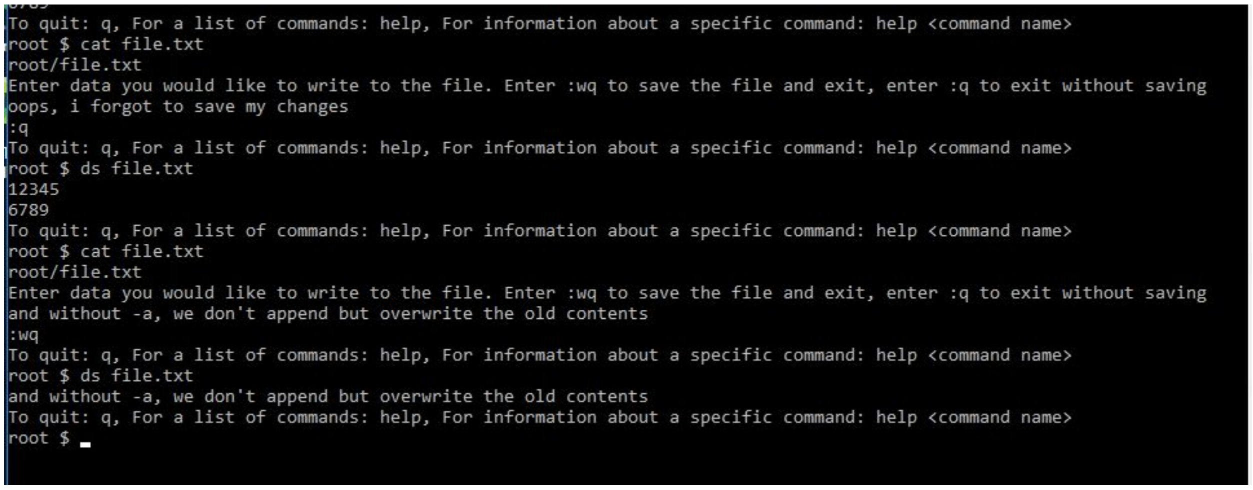
The cat command should be defined to do the following: the -a option stands for append. If the -a option is given, the current contents of the file should be displayed (the bytes only, not the formatted output) followed by a new line. The user should then be prompted to input data to append to the file, 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 examples invocation and possible output are below(note "ds" is a command you will implement later, it displays a file):



```
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 $
```

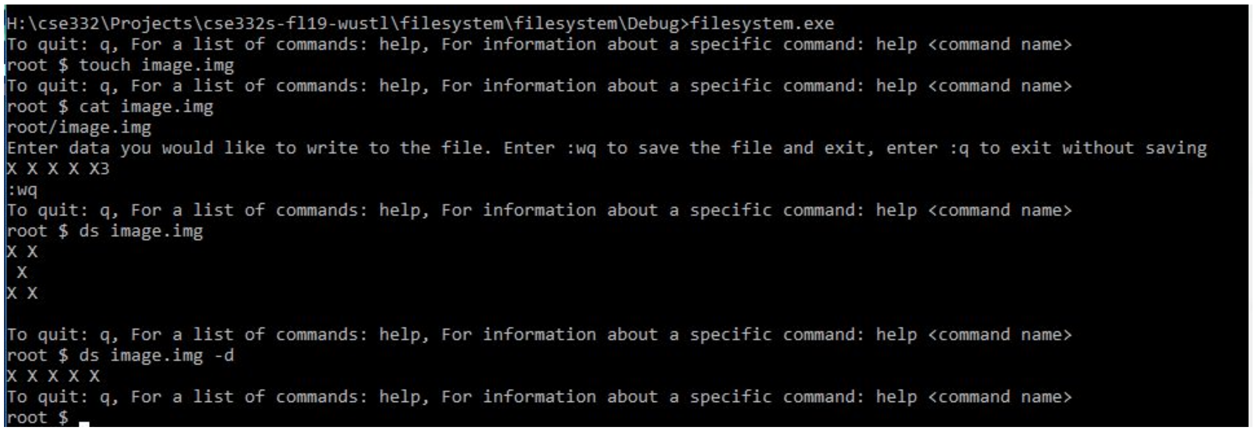
4. **Adding the DisplayCommand:** Add a new command called Display. Display is invoked with “ds”. Display opens a file and displays its contents, either formatted or not (when given the “-d” option for data only). An example invocation could be:

   root $ ds image.img   // formated

   Or..

   root $ ds image.img -d   // unformatted

   And here is a screenshot of it in action:

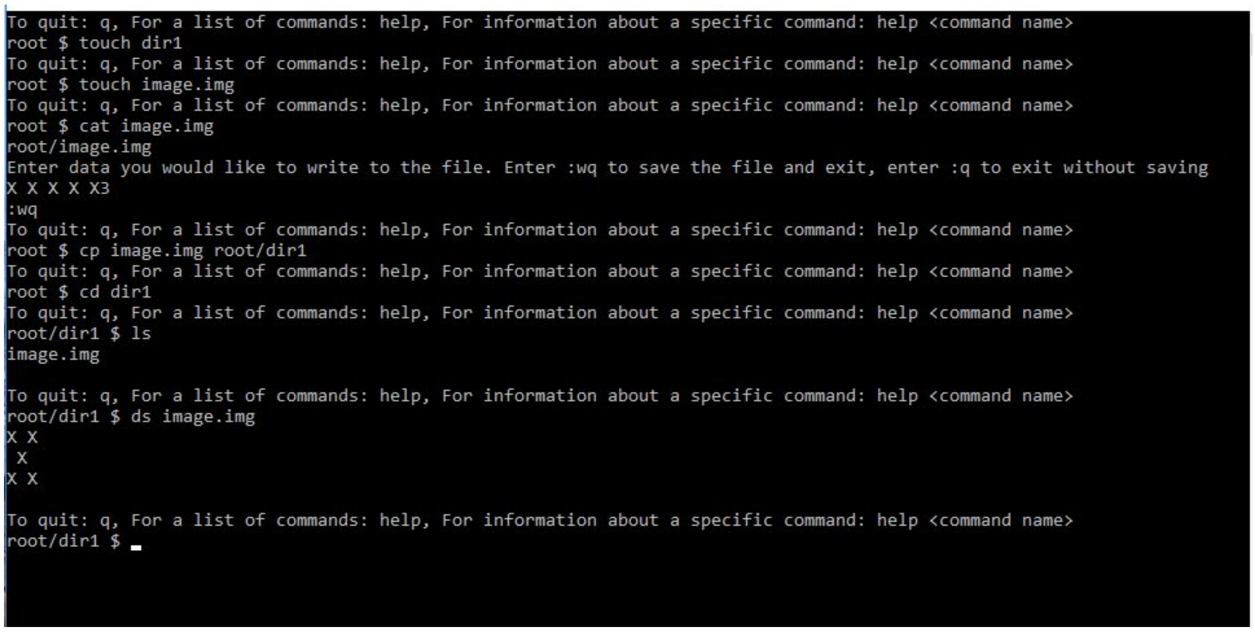


```
H:\cse332\Projects\cse332s-fl19-wustl\filesystem\filesystem\Debug>filesystem.exe
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ touch image.img
To quit: q, For a list of commands: help, For information about a specific command: help <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

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 $
```

5. **Adding the Copy Command:** The copy command will copy a file that exists in the file system and add the copy to the file system in a different location. It is invoked with the following command structure:

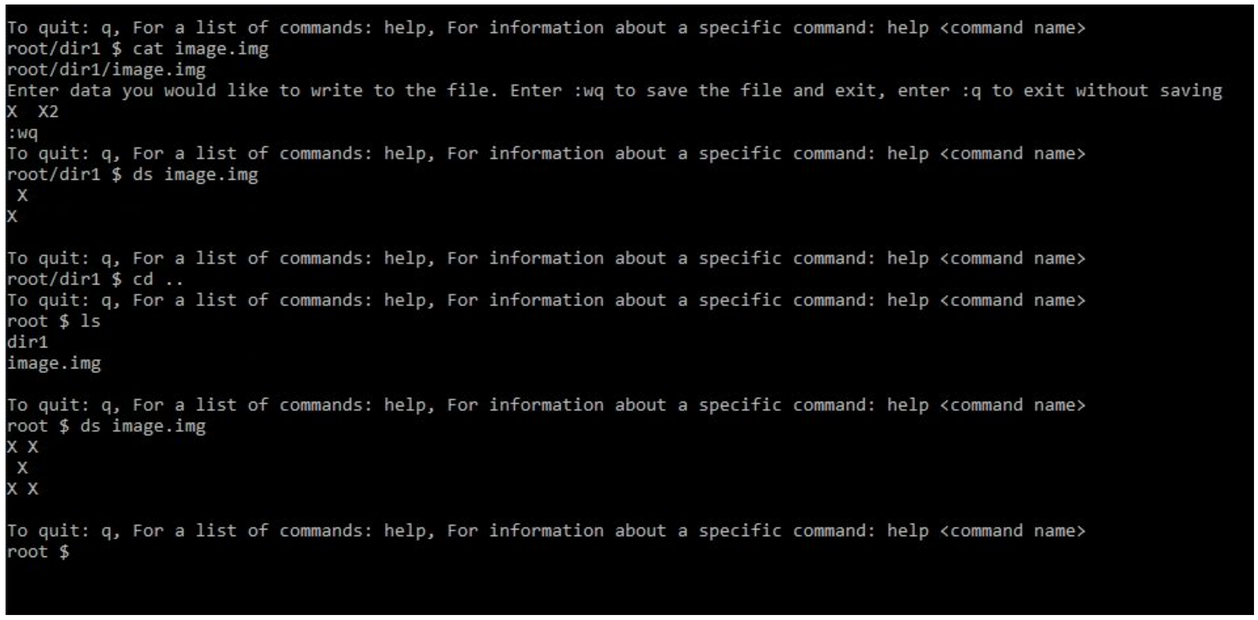
   cp <file_to_copy> <full_path_to_destination>

   Where file_to_copy is replaced with a real file that exists in the CWD and full_path_to_destination is replaced with a path to the directory where the copy of the file should be added. 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. Directories should not be copyable, trying to clone a directory should simply return a nullptr. 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:



```
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ touch dir1
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 $ cp image.img root/dir1
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cd dir1
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ ls
image.img

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ ds image.img
X X
 X
X X

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $
```

Now to ensure a complete copy of the file was made, we will change the contents of the copy and see it does not update the original in any way:



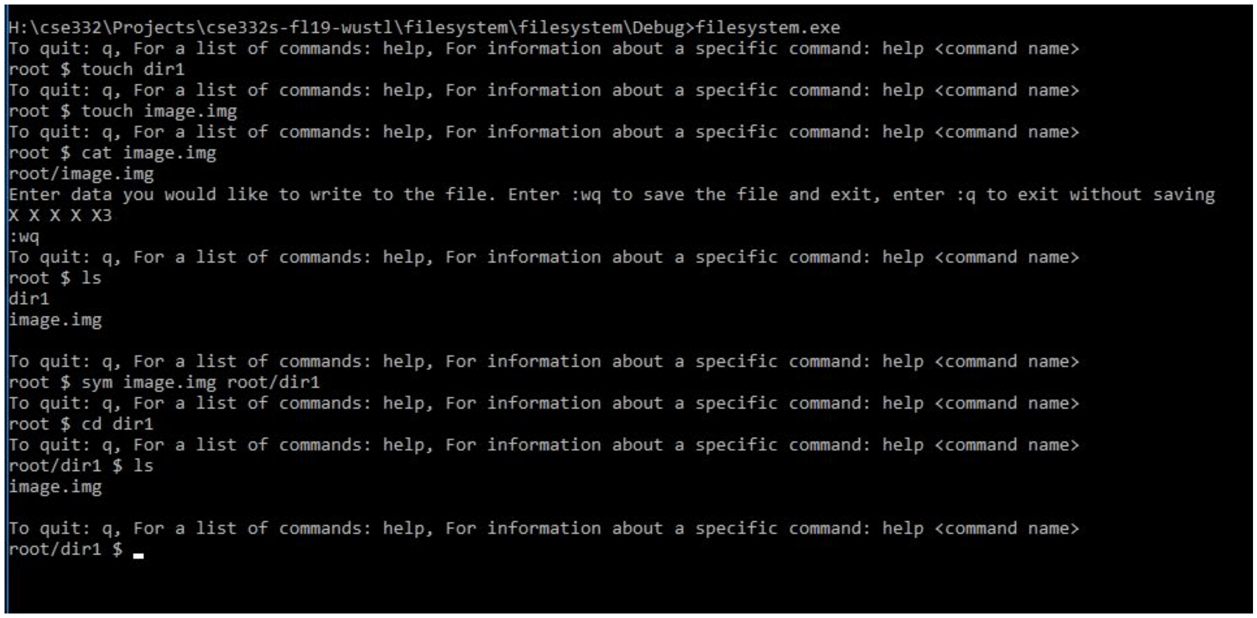
```
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ cat image.img
root/dir1/image.img
Enter data you would like to write to the file. Enter :wq to save the file and exit, enter :q to exit without saving
X  X2
:wq
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ ds image.img
 X
X

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ cd ..
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ls
dir1
image.img

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

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $
```

Hint: when cloning a file, make sure to set the clones parent to be a nullptr. Once the clone method finishes executing, the setParent function will no longer be accessible.

6. **Adding symbolic links:** Lastly, you will add a command that will create a symbolic link to an existing file and add the symbolic link to the file system at a given location. A symbolic link is similar to a copy of the file, however the symbolic link will point to the already existing file, rather than making a unique copy of the file. As there may be multiple symbolic links to a file, the file object itself can only be deleted when all links to the file are removed from the file system. This implies reference counting, or maintaining a link count to a file, much like shared pointers behave. **To receive credit for this command**, you must implement the **Proxy pattern** to handle this. When a symbolic link to a file is created, the file should be replaced in the file system with a proxy object that points to the file. The proxy object will also point to a dynamically allocated "link count" for the file. Each time a symbolic link is created, a copy of the proxy object should be stored in the file system, and the link count should be incremented. Each time a proxy to the file is removed from the file system, the proxy object should decrement the link count. The proxy object that decrements the link count to 0 should delete the file object and the link count for that file. The symbolic link command should be invoked with the following command structure:

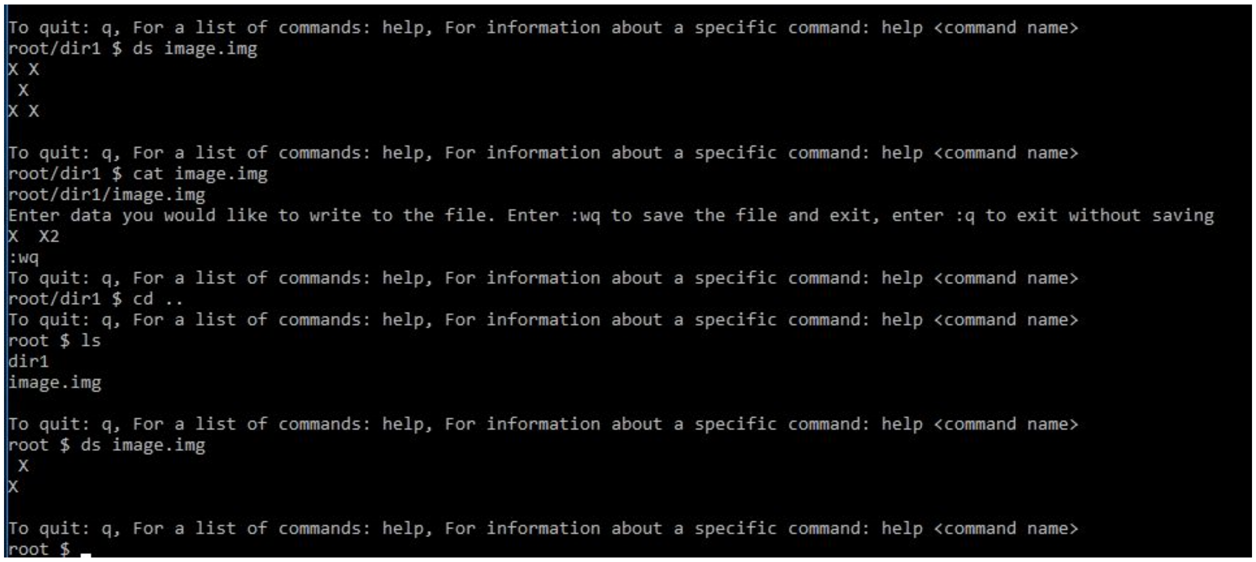
   ```
   sym <filename> <full_path_to_destination>
   ```

   Where filename and full_path_to_destination are as described in the copy command description. Here is a transcript of "sym" in action:

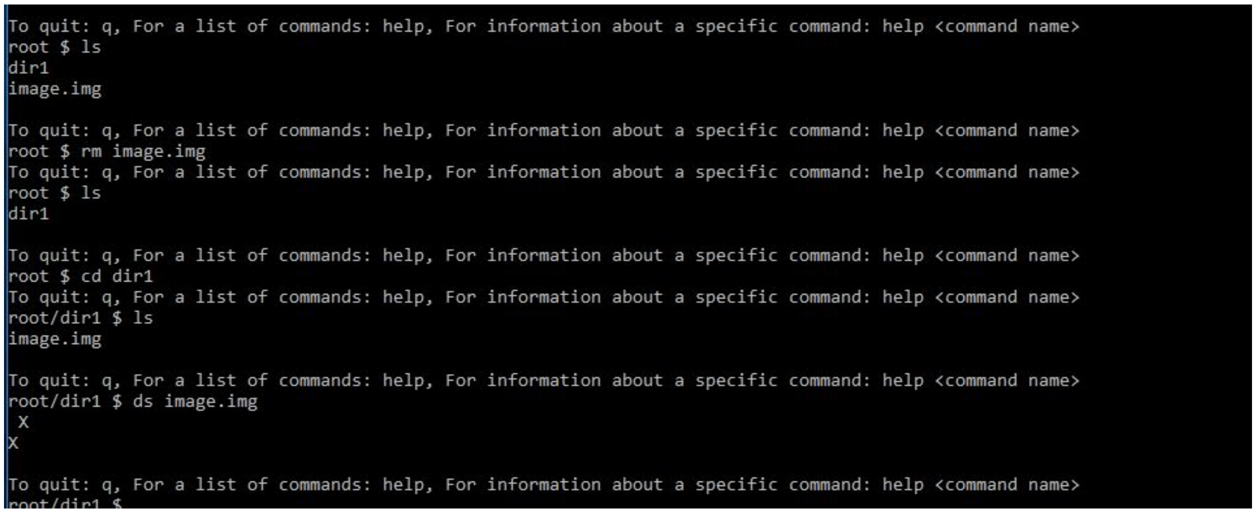
```
H:\cse332\Projects\cse332s-fl19-wustl\filesystem\filesystem\Debug>filesystem.exe
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ touch dir1
To quit: q, For a list of commands: help, For information about a specific command: help <command 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 $ ls
dir1
image.img

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ sym image.img root/dir1
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cd dir1
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ ls
image.img

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ 
```

   We see the symlink is created here, and below we modify the file via the sym link and see that the original is modified as well.



```
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ ds image.img
X X
 X
X X

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ cat image.img
root/dir1/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  X2
:wq
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ cd ..
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ls
dir1
image.img

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

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $
```

And now we show that removing the original from the file system does not destroy the file and we can still access it via other links to the file.



```
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ls
dir1
image.img

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ rm image.img
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ ls
dir1

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root $ cd dir1
To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ ls
image.img

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $ ds image.img
 X
X

To quit: q, For a list of commands: help, For information about a specific command: help <command name>
root/dir1 $
```

Hints: The proxy class may be friended by AbstractFile so that it has access to protected methods of the file interface. Directories should not be copyable or have symbolic links created to them. You may find *dynamic_cast* useful to determine if an AbstractFile * points to a Proxy object.

7. **Updating main and testing:** 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 members name and describe how the work was split between the group members.

8. **Extra credit (5 points):** Use a design pattern to add a piece of new functionality of your choosing. Once you decide on a piece of functionality you would like to add, you should have it approved by me. Extra credit will only be given for groups that have their idea approved. You may ask for approval for an idea up until the lab deadline. Some ideas for additional functionality to add may be:
    i. Implement support for macro commands using composite pattern. This would allow for simple commands to be chained together to create more complex commands.
    ii. Implement a builder using the builder pattern that handles constructing of the CommandPrompt. This would simplify the main function significantly.
    iii. Copy-on-write - use the proxy pattern to implement copy-on-write
    iv. Use the proxy pattern to enforce permissions for a file such as making a file read-only.
    v. Log commands and support undo or replay