CS 355, Assignment 3

The Sneezix File System (UPDATED 3/31/2020)

Individual only

Modified, now due: Wednesday, April 8th, 2020, 12:00 noon (mid-day, not night)

100 points

Problem Statement

I'm giving you a JAR file (available on Canvas) with some existing structure for a Java program that implements a simple file system along the lines of the quiz exercise we went through in class. The quiz document is also posted on Canvas for your general reference. You must complete the tasks as specified below.

Tasks

1. Import the JAR file into Eclipse or IntelliJ IDEA, maintaining the package structure (five packages). Run the program (ShellView.java is the main program) to get a general sense of how it works, noting that a command both generates console output for each command as well as (potentially, but not yet) updating the display of the overall file system on the right (the file system display will start working correctly when you implement task 2). Look at the overall package structure and the contents of the existing class files to understand how the program is structured, and consider what design patterns have already been used in implementing this system. Note that the system uses a Command Factory (a design pattern not yet discussed in class) to hold and supply commands. You'll need to have a basic understanding of the Command Factory class in order to add new commands, but this isn't too difficult – basically new commands need to be added to the command table in the initCommandTable method of this class. The following commands are at least partially implemented:
    a) mkdir xyz – make a new directory named xyz at the current level (current level defaults to the root directory)
    b) mkfile filename size "contents" – make new file named xyz with the specified size and contents set to the string "contents" at the current level
        a. e.g. mkfile test.txt 20 "file contents here"
    c) pwd - display the current working directory to console output
    d) ls - show all information (name, size, contents) about each file system object in the current directory to console output. Display should be in alphanumeric order.
    e) cd xyz – change directory to subdirectory xyz directly below the current level (partially implemented)

   The system also runs a filesystem.ini command batch file (also available on Canvas) upon startup – download this file, place it in an accessible location, and change the file path specification in your Shell class to point to this file.

2. Composite design pattern – right now the file system hierarchy consists of a general FileSystemObject class, with File and Directory classes inheriting from the FileSystemObject class. However, it is single-level at this point; there is no recursive structure.
    a) Modify the Directory class to allow a directory to hold multiple File or Directory objects by adding a data member representing the children of the directory as an ArrayList of the correct type for the Composite pattern.
    b) Make additional modifications to the Directory class as commented within that class to implement functionality for the children.
    c) Uncomment lines in the following command classes to use the children functionality you've just added:
        a. LsCommand
        b. MkdirCommand
        c. MkfileCommand
    d) At this point, your system should support the first four commands above, though working at the root level only. You can test this out manually, and your filesystem.ini file should now execute all implemented commands, creating a file structure (though not all commands in the filesystem.ini will execute correctly at this point.)

3. Strategy and Iterator design patterns - Implement basic functionality for the following additional commands for creation, deletion, and movement between file system objects:
    a) rm xyz – remove file xyz at the current level
    b) rmdir xyz – remove directory xyz at the current level. At this point, rmdir should automatically remove the directory and all children recursively
    c) cd xyz – change directory to subdirectory xyz directly below the current level. This is partially implemented, but additional work will have to be done. There is a helper method in the FileSystem class called setCurrentWorkingDirectory that has some initial code – you can use or change as you wish. Also, this command is commented out in the initCommandTable() method for the CommandFactory.
        a. Also: cd .. – change directory to parent directory – implement this functionality
        b. Also: cd ~ - change directory to root (normally this changes to the user's home directory, but given that we're

not support multiple users yet, we'll go to the root directory here) – implement this functionality

d) cat xyz - display the contents (a field of the File class) of a file xyz to console output

e) du - show the total recursive size of all file system objects in the current directory to console output (note: directories themselves count as 0, but they indirectly are the size of all files recursively stored underneath them)

   a. For the du command, you **OPTIONALLY ma**y use the Iterator pattern to implement the traversal of the file system**, but you can implement this command in any way that you want.**

You must use the Strategy pattern to implement each of these basic commands (think: each command is a strategy for implementing a particular task), and each command should implement the standard interface given.

You do not have to deal with error conditions for each command – just make sure the functionality works for the base desired operations.

Note that if an unknown command is attempted to be executed, the ErrorCommand object is returned by the Command Factory class and is executed to generate an error message.

4. Proxy design pattern - Support the addition of symbolic links to the file system (i.e. an entry at one location in the file system can act as a reference to another directory or file somewhere else in the system.) We'll simplify this for now, and make the proxy only be able to refer to another object in the same directory. You'll need to support the commands:

   a) mklink xyz <existing location>; where the new link xyz and the existing location (file or directory) has to be in the current directory.

   b) rmlink xyz – remove the link xyz (though do NOT remove the associated file object)

   c) A link should be displayed with a star (*) after the name, just as a directory is displayed with a slash (/) after the name, in the file system display. If you want, you can add something more realistic (e.g. linkname* -> filesystemobjectpointedtoname)

   You must use the Proxy design pattern to implement links, with the link acting as a proxy to the real file object being linked to.

5. **OPTIONAL (extra credit):** Right now, the file system display on the right is a separate action undertaken when the Execute button is pressed

in the ShellView class. Change this to make the file system display use the Observer design pattern to make this work.

Auxiliary Tasks:

6. You may design in and implement additional utility methods and classes to support this system as needed.

Design Constraints

7. The above tasks should be implemented in a way that minimizes the use of "structural-ifs", both for file system objects and commands.
    - **a) Structural-ifs test the type of an object and act accordingly. Regular procedural if statements for non-type-checking conditions are fine.**
    - b) Think carefully about how you want to store and work with both concepts (**the file system objects and the commands**).

Notes

- While we're primarily studying design, and I'll evaluate primarily on that, I want you to see some of the real-world issues that occur when working with patterns. The implementation of the patterns and associated support classes/utilities may not be clean when patterns are combined. There can be interesting interactions here between the different file system object types and the different commands we want to implement. However, don't get hung up in the implementation details – work to implement the functionality in some form, and do that as simply as possible while still making use of the pattern ideas.

- Remember that the main goal of a design patterns guru is to see where patterns naturally occur, to use them appropriately, and to adapt them and work with them as needed.

Submission Requirements

- You are to turn in the following through Canvas
    1. a JAR or ZIP file containing your program source code
    2. a Visio file containing a UML class diagram showing the structure of this program, including proper use of IS-A relationships (inheritance or interface usage), HAS-A relationships (can be shown as general associations or aggregate associations (open diamond and line, for coupling generated from class-level data members), and USES-A relationships (dotted line, for coupling generated from one method rather than from data for the whole class). Convert the Visio diagram to PDF format for submission, as you did in Assignment 1.
    3. A short summary document (Word or PDF) discussing your work, including but not limited to:

   1. How did the design pattern-based structure of the program affect the ease or difficulty of doing the above tasks?
   2. What issues, if any, do you see in the combination and interaction of the design patterns here?

Extra Credit (up to 10%)

- ONLY IF all required work is completely implemented, you can get additional credit for the following (NOTE: you must both do the work and indicate that you've added this extra credit functionality in your summary document above.
   1. **Using the Observer design pattern for the file system update, as noted above.**
   2. Allow absolute or full paths before file/directory/link names; e.g.
      - i. mkdir /<path>/xyz – make a new directory named xyz at the specified full path
      - ii. mkfile /<path>/xyz – make a new file named xyz at the specified full path
   3. Implement other basic Unix-style commands.
   4. Support the addition of single-user protection; i.e. each file system object can have read and/or write privileges (default on creation: read-only). The cat command applied to a file without read permission should fail. The rm or rmdir operation applied to a file system object without write permission should fail. The cd command should not be allowed if the target directory does not have read permission. The various mkXXXX commands should fail if trying to create an object in a directory without write permission. The ls command (if it was implemented) would also display an 'r' if read privilege is set, and a 'w' if write privilege is set. You'll also have to change your mkfile and mkdir commands to set default protections on each object created.
   5. To support manipulation of file system object privileges, you could now support the operation:
      - i. chmod <+ or -><r and/or w> xyz (add or remove read and/or write privilege to file system object xyz)
   6. Support a command history that can be navigated through in either direction (you may add to the GUI to support this; e.g. you might want to use some forward and back buttons/icons). There is a Command design pattern (research on your own), but you can implement this in any way possible.

Sample Filesystem.ini Contents

```
pwd
```

```
mkdir bin
mkdir home
mkdir tmp
cd bin
mkfile ps 30 “process listing utility”
mkfile ls 25 “file system object listing utility”
mkfile cat 15 “file content listing utility”
rm cat
cd ..
cd home
mkdir cs355
cd cs355
mkfile prog1 12 “prog1 contents”
mkfile prog2 27 “prog2 contents”
cd ..
mkdir misc
cd misc
mkfile resume.doc 38 “my vita”
mkfile test.txt 5 “test file”
rm test.txt
mkdir new
rmdir new
cd ..
cd cs355
pwd
mklink prog3 prog2
cat prog3
rmlink prog3
cat prog2
cd ..
cd ..
ls
cd bin
rm ps
cd ..
du
cd bin
cd ~
pwd
```