Studio 21: CSE 332S File System Assignment Part 6

Key Topic: Command Pattern

In previous studios, we created a family of classes that represent different file types (AbstractFile and different concrete classes that define that interface) and a set of classes that represent different file system types (AbstractFileSystem and the concrete classes that inherit and define this interface). In this studio, you will build a client for your file system to allow a user to send commands to the file system and use files in an interactive way. You will implement a "command prompt" class to handle user input and use the command pattern to facilitate different types of commands the user may execute.

Unit tests have been provided for you to test your code before moving on to the next part of the assignment. ***You must follow the naming conventions (highlighted in blue) described in the instructions below in order to pass the tests!***

All of the following parts must be completed.

1. **Do not continue unless you have completed Studio 20, complete Studio 20 instead.** Use Zoom to create a meeting and invite your group members as in studio 16. I would suggest only 1 group member writes code during the studio, ideally a different team member than the last studio. At the end of the studio, you will commit and push those changes so that the other group members can pull the changes from the shared repository and merge them into their own local repository. Review the git workflow information from Studio 16 if needed.

   Open "*Studio21.sln*" and switch to the "Solution Explorer" tab. Open the provided ReadMe.txt file and in it, record your answers as you work through this studio. For the answer to this question, record all team members' names.

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.

2. As described in lecture, we will use the Command pattern to implement a user interface for our file system. Let's start by declaring the interface a ConcreteCommand will define.

In the AbstractCommand.h file in the SharedCode folder, declare a class called *AbstractCommand*.

AbstractCommand should declare 2 public and pure virtual member methods and should declare a virtual destructor and ask the compiler to define it for us (virtual ~AbstractCommand() = default). The 2 member functions should be:

a. int execute(std::string) - the parameter will hold any information that is passed to the command
b. void displayInfo() - this is essentially a usage message for the command, it will display to the user how the command should be invoked

As the answer to this question, why is it important to declare a virtual destructor in the base class?

3. We will now create a CommandPrompt class. The command prompt handles user input/output. It is the user interface for our file system. It is also the Invoker in the command pattern. The command prompt will be configured with concrete commands and will invoke them when requested by the user. In the provided CommandPrompt.h and CommandPrompt.cpp files (create these files if they don't exist already as described in the note from step 2), declare and define a class called "CommandPrompt" with the following member variables:

a. A std::map that maps a std::string to a pointer to an AbstractCommand. This map will store the command objects the CommandPrompt is configured with
b. A pointer to an AbstractFileSystem which will store the file system object the command prompt is configured with
c. A pointer to an AbstractFileFactory which will store the file factory object the command prompt is configured with

And the following member functions, defined as described below:

d. A public default constructor that initializes both pointer member variables to be nullptr
e. The compiler defined destructor will be fine
f. A public method void setFileSystem(AbstractFileSystem*) - sets the AbstractFileSystem pointer member variable equal to the parameter passed to this method
g. A public method void setFileFactory(AbstractFileFactory*) - sets the AbstractFileFactory pointer
h. A public method int addCommand(std::string, AbstractCommand*) - inserts the parameters passed to this function as a pair into the map of commands, returns success if the insertion was successful, returns a non zero value otherwise.
i. A protected method listCommands() that iterates over each command in the map and prints the command's name

j. A protected method std::string prompt() that prompts the user to input:
   i. A valid command
   ii. “q” to quit
   iii. “help” for a list of commands
   iv. Or “help <command name>” for details about a specific command

   The prompt method should then insert on a new line a single ‘$’ character followed by a couple of spaces (do not insert endl after the $ and spaces. The goal is to emulate a typical command prompt where the user will type a command after the ‘$’). The prompt method should then call getline to read a line of input from the user into a string and return the string. The output should be similar to:

```
Enter a command, q to quit, help for a list of commands, or
help followed by a command name for more information about
that command.
$  <user input goes here>
```

k. A public method int run() that is defined to repeatedly, in a *while(1)* loop, call prompt and store the result into a string and proceed as follows:
   i. Check if the input is “q”, if it is quit and return an appropriate non zero value.
   ii. If the input is “help”, call the listCommands method
   iii. If the input is not one of the above two cases, check that the input is only 1 word long(look for a ‘ ‘ in the input, if none is found it is 1 word). If so:
   iv. Search the map of commands for the command that matches the input. If a command is found, invoke the command by calling “execute” on it with an empty string as argument. If the command returns an error, print to the user the command failed. If the command did not exist, print a message to the user notifying them of this
   v. If the input is longer than 1 word, wrap the input in a istringstream and extract the first word into a string.
      1. If the first word is “help”, extract a second string. This string is the command to display more information about. Search for this command in the map of commands and call displayInfo() on it, or print a message to the user if the command does not exist.
      2. Otherwise, the first word is a command name. Search for the command in the map of commands and call execute on it if it exists, passing the remaining input(everything after the first ‘ ‘) as the string parameter. If the command does not exist or the command returns an error, display an appropriate message to the user.

As the answer to this question, think about “dependency injection”. A command prompt requires a file system, a file factory, and command objects. However, the command prompt itself does not know any of the concrete classes it interacts with. Instead, it is injected with concrete command, file system, and file factory objects. Discuss how this makes the command prompt flexible and reusable. Think about different file system implementations and different file factory implementations required to create the file types required by a different file system implementation. Can the command prompt be easily configured to use these instead of the simple file system and its associated factory object?

4. Declare and define a ConcreteCommand class called TouchCommand in the provided TouchCommand.h and TouchCommand.cpp files in the SharedCode folder (create these files if needed). Touch is a command that creates a file. It will be invoked by the user inputting “touch <filename>”. Touch will then create a file called “filename” and add it to the file system. TouchCommand inherits and defines the interface declared by AbstractCommand. TouchCommand will have 2 member variables, a pointer to an AbstractFileSystem and a pointer to an AbstractFileFactory. Declare and define a constructor that takes 2 parameters (AbstractFileSystem * and AbstractFileFactory *) and uses them to initialize the 2 member variables. The compiler defined destructor works for this class. Override the methods inherited from AbstractCommand as follows:

    a. displayInfo simply tells the user how to invoke the touch command and describes the command. So, print to the user something like “touch creates a file, touch can be invoked with the command: touch <filename>”

    b. execute should be defined to ask the file factory to create a file with the provided filename (which should be passed into the function as its parameter). If the file is created successfully, it should be added to the file system object by calling addFile. Call addFile with the name of the file and the pointer to the file that was returned by the file factory. If the file was not created successfully, return an appropriate error. If the file was created successfully but was not added to the file system correctly, call delete on the file that was created and return an appropriate error.

    In main(), dynamically allocate a SimpleFileSystem, a SimpleFileFactory, and a TouchCommand object (configured with the file system and file factory objects). Create a variable of type CommandPrompt and configure it with the file system, file factory, and touch command object you created. Call run() on the command prompt object and test quitting, asking for help, asking for help on the touch command, and executing the touch command. After quitting and returning from the run function, you should still have direct access to the FileSystem object in main(). You can verify “touch” worked correctly by trying to open the file it created. As the answer to this question, describe the tests you ran.

In lab 5, you will implement several more commands including a RemoveCommand for removing files, a LSCommand for listing the contents of the current directory, a CopyCommand for copying files, commands for displaying and writing to files, etc..

5. Finally, test your code using the given unit tests to ensure you are ready to move on to the next part of the File System assignment. *The instructions to do this are in the first part of this assignment.*