Studio 19: CSE 332S File System Assignment Part 4

Key Topic: Behavioral Design Patterns

Last studio, we simplified our file system object by removing the responsibility of creating files from the file system. This studio, we will introduce functionality to read files and display them in ways other than described in the read() function of the concrete file classes. We will use the visitor pattern to accomplish this.

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 18, complete Studio 18 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 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 "*Studio19.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 17-21 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. In a typical system, reading a file simply returns the files data. It is up to an application to decide how to interpret that data and display it. For instance, currently read is defined by *ImageFile* to display the image to the console, however what if we just wanted to read the contents of the file directly. Then we could display the file contents in any way we wanted, we could edit the file contents and then rewrite the file, etc. Currently this isn't supported. Let's start to implement this functionality by first updating the read function to simply return the contents of a file, not display the file to cout.

   a. Update the AbstractFile read function to return a vector<char>.
   b. Then update the read function in each concrete class to return its contents. (comment out your code for displaying the file, we will use it later...)
   c. In main, try creating and then reading a file. Store the files contents into a local variable declared in main. Then make an edit to the file's contents and rewrite the file with the modified contents. As the answer to this question, describe the tests you ran.

3. We can now read the contents of the file, however we no longer have functionality to display the file. We could add a display function to the AbstractFile interface and let each concrete file class define it in an appropriate way. However, this adds an extra responsibility to the file classes. Also, what if a file can be displayed in multiple ways? We would have to add a virtual member function for each unique way a file can be displayed to the AbstractFile interface. This will clutter a file's interface and make the class much more complex. The visitor pattern allows us to add new functionality to a class without significant modification to its interface. Let's implement the visitor pattern now.
   a. The first step is to create an interface for a Visitor. Open the header file named "AbstractFileVistitor.h", create an abstract base class called "AbstractFileVisitor". The interface should include a pure virtual member method for visiting each concrete file type. The methods will be called "visit_<file type>" and take a pointer to the appropriate file type as an argument. The methods should have a void return type. (replace <file_type> above with each real file type, for example: visit_TextFile)
   b. Next, update the interface of your files to allow them to be visited by a visitor. Add a pure virtual member method to the AbstractFile interface called "accept". Accept takes a single argument of type pointer to AbstractFileVisitor and returns void.
   c. Declare and define "accept" in each concrete file class to call the appropriate "visit" method via the visitor passed to the method, passing "this" as the argument to the call.

   As the answer to this question, describe the sequence of communication between a visitor and the object it visits. Try creating an "interaction diagram" on paper or the board for this pattern.

4. Open the header and source files called "BasicDisplayVisitor.h" and "BasicDisplayVisitor.cpp" respectively. Create a concrete visitor class called BasicDisplayVisitor, define each visit function as it was previously defined in the read method of the concrete file classes, or as described in studio 16(TextFile) and 17(ImageFile).

    a. Test your program: In main, create a few files and a variable of type BasicDisplayVisitor. Call accept on the file objects and pass the address of the BasicDisplayVisitor variable to the call.

As the answer to this question, discuss the concept of “delegation” within your group. Describe how the visitor pattern uses delegation to add additional functionality to a family of classes without cluttering their interface.

5. Suppose we wanted to add functionality to display a file’s metadata (the file’s name, size, and type). Think about how we can accomplish this with the visitor pattern. Create a new concrete visitor to add this functionality to your program and test it. Open the header and source files “MetadataDisplayVisitor.h” and “MetadataDisplayVisitor.cpp” respectively. You should name the class MetadataDisplayVisitor. Please only record types as “text” or “image” so that the unit tests will pass.
   Hopefully this shows you the main advantage of the visitor pattern. However, as the answer to this question, think about a situation where we have a lot of unique visitors. Now, think about what classes in our program would need to be modified if we decide to add a new concrete file type that should be visitable. Based on this, describe a negative consequence of the visitor pattern. When might it be a bad idea to use this pattern?

6. 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 previous part of this assignment.*