Studio 18: CSE 332S File System Assignment Part 3

Key Topic: Creational Design Patterns

Last studio we created an AbstractFileSystem class, which declares the interface for creating files and managing access to those files. We then created a single concrete class that defined the AbstractFileSystem interface (SimpleFileSystem). Today, we will work towards making our design more flexible and reusable by reducing the responsibilities of our file system object. The **SOLID** design principles (https://en.wikipedia.org/wiki/SOLID) are a set of 5 design principles intended to make OOP software more flexible, understandable, and maintainable. These principles are used by many industry organizations. The "S" in SOLID stands for the "Single responsibility principle" (https://en.wikipedia.org/wiki/Single_responsibility_principle). This principle says a class should have only a single responsibility, meaning only changes to the project specification regarding that responsibility should result in changes to the class. Other responsibilities should be delegated to different objects. Currently, a file system object has 2 responsibilities in our program, creating files as well as controlling access to those files. In this studio, we will remove the file creation responsibility from our file system objects.

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 17, complete Studio 17 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 "*Studio18.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. Currently, the file system object creates file objects. This is done by the createFile method. createFile is an example of a factory method. It is declared in the interface of the AbstractFileSystem class, however it is not defined. Instead, each concrete file system class that inherits from AbstractFileSystem overrides and defines the method to create objects of the specific file types used by that file system implementation. Discuss the factory method with your group. Think about how easy/difficult it would be to create a new file system implementation that creates different file types than the SimpleFileSystem (assume those concrete file classes are already defined). Now, think about what code would need to be modified if we create a new concrete file type that is used by many different file system implementations. Notice that a concrete file system must know exactly what types of concrete files it creates. In other words, a concrete file system depends on the concrete file types it creates. Later, we will remove this dependency and a file system object will be completely unaware of the concrete type of the files it manages. As the answer to this question, briefly summarize your discussions.

3. Our file system will be responsible for managing access to files, not creating new files. Files themselves will be created outside of the file system and then added to it for management. This fits the “Single responsibility principle”, managing access to files and creating files are separate responsibilities. We will eventually move file creation to a client object (a terminal or user interface for the filesystem). For now, we will use the main function as our client. Our client may be configured with any concrete file system object and each concrete file system type may support different concrete file types. We need a way to support this. The **abstract factory pattern** works well for this. In this pattern, a factory is a separate object that is responsible for creating objects of other types, as opposed to a member function as in the factory method pattern. A client is configured with an instance of an abstract factory (a concrete factory that inherits and defines the abstract factory interface), and uses the interface of the abstract factory to create new objects by forwarding the work of creating an object to the factory. The interface of an abstract factory includes methods for creating different types of objects that are created by the client.
    a. Open “AbstractFileFactory.h”. In this header, create an interface for an AbstractFileFactory. The interface contains one method, createFile which takes a std::string argument representing the name of the file to create and returns a pointer to an AbstractFile (a pointer to the file that was created)
    b. Open “SimpleFileFactory.h” and “SimpleFileFactory.cpp”. Declare and define a concrete file factory (SimpleFileFactory) class that inherits the AbstractFileFactory interface and define it similarly to the current implementation of the createFile method of SimpleFileSystem (you should be able to copy/paste some code) and returns a pointer to the AbstractFile.

i. The factory creates a file only, it does not add it to the file system or even have access to the file system. So you will not need to check if the file already exists in the file system or add the file to the file system. This will be done by the client by calling addFile on the file system object it is configured with after the file is created.

As a group, discuss what you think might be advantages/disadvantages of the abstract factory pattern and summarize your discussion as the answer to this question. When would you want to create a new concrete file factory class? What code must be modified if a new file type is introduced that should be created by existing factories? Given two file system implementations that manage the same types of files, can the same factory be used to create files for both? Given two file system implementations that manage different types of files, can the same concrete factory implementation be used to create files for both?

4. Test your code to ensure it works properly. In main(), initialize a pointer to an AbstractFileSystem to point to a dynamically allocated SimpleFileSystem object. Configure a pointer to an AbstractFileFactory to point to a dynamically allocated SimpleFileFactory object. Create a few files via the factory object and add the files to the file system. Try opening, writing, and reading the files. As the answer to this question, record the test you ran and the output produced by those tests.

5. A file system is no longer responsible for creating files. Remove createFile from the AbstractFileSystem interface and remove its declaration/definition in SimpleFileSystem as well. At this point, does SimpleFileSystem depend on any concrete file types or does it depend on the AbstractFile interface only?

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.*