Studio 17: CSE 332S File System Assignment Part 2

Key Topic: OOP Design

Last studio, you created an AbstractFile class, which declares the interface all file types share. You then created a single concrete class that defined the AbstractFile interface (TextFile). In this studio you will create a new concrete file type (ImageFile). You will then start to implement a file system, which manages creating, destroying, opening, and closing files.

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 16, complete Studio 16 instead.** Use Zoom to create a meeting and invite your group members as in studio 16 (or if in class, have your TA place you and your team in a breakout room). 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 "*Studio17.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. Now that you have a common interface shared between all file types, it is easy to extend your program to include new concrete file types. The second file type you will create is an ImageFile. Open "ImageFile.h" and "ImageFile.cpp" and in them, declare and define a class called "ImageFile". ImageFile should inherit publicly from your AbstractFile class. ImageFile should have the following member variables:
   a. A std::string, which stores the name of the file

b. A std::vector<char>, which stores the contents of the file
c. A char, which stores the size of the image.

Declare and define a constructor for the ImageFile class that takes a single parameter, a std::string, and uses it to initialize the name member variable of your class. Also initialize the size member variable to 0.

Define the interface inherited by your AbstractFile class as follows:

a. getSize returns the size of the vector<char> member variable (not the member variable storing the size of the image)
b. getName returns the std::string member variable
c. write: We will be displaying files in the terminal, so we are limited in what exactly an "image" is (we can't print color, etc.). For this lab, an image will be an n x n board of pixels. Each pixel will either be an 'X' or a ' ' (space) when the image file is read and displayed. When writing an image file, the file will be encoded as a vector<char> where the first n x n chars in the vector are either a 'X' or a ' ' (space). The last char in the vector will be the size (n) of the image. For example, the input argument may be a vector<char> containing:

   { 'X', ' ', 'X', ' ', 'X', ' ', 'X',' ', 'X', '3' }

   NOTE: the size is passed in as the character '3' not the number 3. We will adjust for this now.

   With this input, you should first look at the **last** char in the vector and use it to set the size member variable. Remember the size is passed in as the character '3', which actually has the integer value of 51. Take a look at the ASCII table here to see how other characters are encoded. To obtain the integer value 3 (which should be stored in the size member variable) from the character '3', you can just subtract the integer value of the char '0' which is 48. This will work to convert any character that is a digit ('0'-'9') to its integer value. Then copy the previous contents of the vector into the vector<char> member variable that stores the contents of the file. If any issues occur, such as a size mismatch or a pixel being something other than a 'X' or a ' ', you should clear the contents of the file, zero out the size member variable, and return a unique non-zero value representing the error that occurred. Otherwise, return 0 indicating the write was successful.

d. append: an image file should not support append. Define this method to simply return a unique non-zero value representing the append operation is not supported by this file type
e. read: reading a file should be much like printing the gameboard from lab 2. The size member variable is the width and height of the image. An `(X,Y)` coordinate

pair can be converted into an index in the contents vector using the formula below:

```
Y * size + X
```

Remember to print the image from the top row down. A helper function that takes a coordinate pair and returns an index may be helpful. Should this function be private, public, or protected? (think about abstraction)

Test the functionality of your ImageFile class. Describe the tests you ran as the answer to this question.

3. Now that you have some file types and can create file objects, we need something to handle managing multiple files. This will be the file system. The file system object will manage creating empty files, adding files to the file system, opening files, closing files, and deleting files. Thinking ahead, we would like our software to remain easy to extend in the future. Today you will implement a very simple file system, however in a later studio you will be adding an implementation of a more complex file system that supports storing files hierarchically in a directory structure. We would like for the client that interacts with the file system to work equally well with either implementation. To support this, you should create an interface for a file system and program your client code to use the interface rather than a concrete file system implementation. Do that now, open the header file called "AbstractFileSystem.h". In this header file, create an interface called "AbstractFileSystem". The AbstractFileSystem interface should have the following member methods:
    a. addFile - this method takes 2 arguments. The first is a std::string providing the filename, the second is a pointer to an AbstractFile. This method returns an int.
    b. createFile - this method takes a single argument which is a std::string providing the name of the file and returns an int.
    c. deleteFile - this method takes a single argument which is a std::string providing the name of the file to delete and returns an int.
    d. openFile - this method takes a single argument which is a std::string providing the name of the file to open and returns a pointer to an AbstractFile
    e. closeFile - this method takes a single argument which is a pointer to an AbstractFile and returns an int.

    As the answer to this question, briefly discuss as a group the benefits of "programming a client to use an interface rather than a concrete class" and record your biggest takeaway. Think about creating software that is easy to extend in the future.

4. You now have an interface that describes what methods can be called on an "AbstractFileSystem". The last step of this studio is to define your first concrete file system class. This will be a very simple file system that does not support directories (or folders). Open "SimpleFileSystem.h" and "SimpleFileSystem.cpp". In them declare and

define a class called “SimpleFileSystem” that inherits publicly from AbstractFileSystem. SimpleFileSystem should have the following member variables:

a. A std::map that maps a std::string to a pointer to an AbstractFile. This map will contain all files in the file system.
b. A std::set that contains pointers to AbstractFiles. This set will contain all files that are currently open (openFiles).

The compiler defined default constructor will work fine for us now, so there is no need to declare or define a constructor for your SimpleFileSystem class. Define the interface inherited by the “AbstractFileSystem” class as follows:

c. addFile: first, check the map of files to ensure the file does not already exist (if it does, return an appropriate non-zero value), then ensure the AbstractFile pointer passed to the function is not a nullptr (return a non-zero value if it is). Now insert a std::pair containing the name of the file and the pointer to the file into the map member variable of the class.
d. createFile: First, ensure a file with the same name does not already exist. If it does, return a non zero value indicating the file already exists. Otherwise, you should use the file extension to determine which type of file to create. For example, if the file name passed to addFile is “test.txt” then you would create a TextFile. If it were “test.img” you would create an ImageFile. To do this, create a string that contains the file extension only(txt or img). If the string contains “txt”, dynamically allocate a new TextFile and add it to the map of files. If the extension is “img”, dynamically allocate an ImageFile and add it to the map of files. Return 0 to indicate success in either case.
e. openFile: check to see if the filename passed as the argument to this function exists by searching the map of files. If it does exist, check to ensure it is not already open. If the file exists and is not open, add the file to the set of open files and return a pointer to the file. Otherwise, return nullptr.
f. closeFile: search the set of open files for the AbstractFile pointer passed to this function. If the file is currently open, remove it from the set of open files and return 0 for success. Otherwise, return a non zero value indicating the file was not open.
g. deleteFile: First, check that the file exists. If it does, ensure it is not currently open (the file system should not allow a file to be deleted if that file is open and in use elsewhere). If the file is open, return a non zero value indicating this. If the file does not exist, return a non zero value indicating this. Otherwise, remove the file from the map of files, call delete on a pointer to the file to avoid memory leaks, and return 0 for success. **Hint:** make sure to declare a virtual destructor in the AbstractFile class. You can declare the destructor with “= default” to ask the compiler to define it for you. This ensures that when delete is called on a pointer to an AbstractFile, the destructor that actually executes will be that of the concrete file type pointed to by the pointer (dynamic binding).

Create a SimpleFileSystem object in main, create a few files and add them to the file system via the addFile method. Create a few files via the createFile method as well. Try opening, writing, reading, and deleting files to ensure things are working properly.

As the answer to this question, first discuss with your group where you would need to add code in the SimpleFileSystem class to be able to create an object of a new concrete file type (a MusicFile for example). Record your answer in your readme. Take a closer look at the SimpleFileSystem class. Note that it is entirely coded to use the AbstractFile interface, with the exception of the createFile method. The createFile method depends on understanding how to construct objects of each concrete file type. Next time, you will alleviate this dependency by creating a separate object to handle file creation, making the file system completely independent of the concrete type of each file it contains.

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 previous part of this assignment (the studio16 document).*