Name: _____ Date:_____

Lab 3 – Factory Method Pattern

Objectives

Part 1 – Setting up the Project

Part 2 – Implementing the Factory Method Pattern

Part 3 – Making the Factory Decision External

Reflection

Background/Scenario

In this lab you will work with classes that implement logging. Currently, the client of the loggers creates the logging classes directly. You will change the client code so that it uses factory methods to create the loggers.

After the following steps are completed, the client code stills hard codes the type of factory to create. The last part of the lab will be to create a configuration file that tells the client which factory to use.

Required Resources

- Visual Studio 2017

Part 1: Setting up the Project

Step 1: Download the FactoryMethodLabStarterCode project.

Open the FactoryMethodLabStarterCode project and familiarize yourself with the code.

Essentially, this is a client console application which consumes logging services from a Logger DLL.

Step 2: Look around at the classes and understand what they are doing.

The Logger DLL contains a ConsoleLogger class that has two constructors; one to create a default logger and one which takes a LogLevel to specify the level at which messages should be logged.

The library also contains a FileLogger class. It has constructors which take a file name as a construction parameter as well as the optional LogLevel

In this lab you will refactor the code in the library to use the Factory Method pattern.

Part 2: Creating the Factory

Step 1: Change the client so that it will use the new factory.

Locate where the ConsoleLogger is used (program.cs).

Change this from using “new ConsoleLogger” to using a factory. The new code will look something like:

```
ConsoleLoggerFactory factory = new ConsoleLoggerFactory();
ConsoleLogger logger1 = factory.CreateLogger();
```

Obviously this code will not compile yet.

Step 2: Create the Factory in the DLL

Now that the client is broken, let's fix it. The fix is to add the ConsoleLoggerFactory to the DLL that the client now uses.

1) Define a ConsoleLoggerFactory class and implement the CreateLogger method in the class. Remember that the ConsoleLoggerFactory class must be public if it needs to be accessed outside the DLL.
2) The CreateLogger method will simply create a new ConsoleLogger and return it to the caller.
3) Build and test the code, it should now compile and still work as before.
4) Now add a second creation method to the ConsoleLoggerFactory. This method will return a ConsoleLogger that takes LogLevel as a parameter.
5) Update the client to use this new factory method to create logger2. Build and run the code, it should work exactly the same as before.
6) At the moment, the client can still do new ConsoleLogger() which is exactly what we are trying to avoid. To fix this make the ConsoleLogger's constructors internal. (Internal means the method cannot be accessed outside the DLL.)

Step 3: Implement FileLogger

Perform the same steps as above for the FileLogger. Build and test the code, it should work as before.

Step 4: Unify the Factories

We still have not completed the Factory Method pattern . At the moment you have two unrelated factories. Remember that the factory method pattern provides a factory interface that creates a single type, and the types and factories have a one to one relationship.

To fix this code you need to provide a common base class for the factories and a common base class for the loggers.

1) Create a new public interface called ILogger. This will be the base for your ConsoleLogger and your FileLogger
2) This class should contain definitions of the two Log methods used in both ConsoleLogger and FileLogger.
3) Derive your ConsoleLogger class from ILogger.
4) Change the client so that it declares uses of ILogger rather than ConsoleLogger
5) Refactor FileLogger the same way.
6) Change the client to use ILogger declarations instead of FileLogger
7) Run the application. It should work as before.

Step 5: Create the common base class

Now it is time to finish up the Factory Method pattern. You need to change the factories to use a common base class.

1) Add a public interface called ILoggerFactory and have the two existing factories derive from it.
2) In the interface add two methods

```
ILogger CreateLogger();
```

```
ILogger CreateLogger(LogLevel level)
```

3) Refactor the CreateLogger methods of ConsoleLoggerFactory and FileLoggerFactory to return “ILogger” types.

4) Change the client to create instances of the ILoggerFactory. For example:

```
ILoggerFactory factory = new ConsoleLoggerFactory();
ILogger logger1 =  factory.CreateLogger();
```

Step 6: Implement FileLoggerFactory

Do this for the FileLoggerFactory as well. Build and run the code and it should still work.

Part 3: Making the Factory Decision External

During the lecture code examples, we have implemented a factory that uses reflection and obtains the type information for all the classes dynamically at runtime. Then in a second demo, we used the input from the user as a console parameter. In this lab, we will be using the applications configuration file as a location to get the factory information from.

Now for the fun part: using the app.config file.

Currently, the client still must know the type of the factory to create loggers. This means that to change the logger for the client you would need to recompile the client. A better solution is to store that “type” information in the application’s configuration file. As a final step, you will edit program.cs and have the client read the configuration data that it needs to create the factory.

Add a reference to the System.Configuration library and a “using System.Configuration” statement to the Client project. You will need this to read the configuration file.

The app.config of the file of the Client project needs an <appSettings> section between the configuration tags. Next add a key set to 'ConsoleFactory'. The value of this entry should contain both the namespace.name of the factory class that implements the factory and the name of the DLL that contains said factory class. Customarily, these are stored as a comma separated string. Add this config information between the configuration tags. Since you probably haven’t done this before, here is the code.

```
<appSettings>
       <add key="ConsoleFactory" value="Logger.ConsoleLoggerFactory,
Logger"/>
 </appSettings>
```

Step 1: Create the Factory Loader

Add a method to the Client project called LoadFactory. This method takes a single string parameter (the key to read from appSettings) and returns an ILoggerFactory

This method should look like this:

```
private static ILoggerFactory LoadFactory(string factoryName)
       {
string factoryTypeAsString = ConfigurationManager.AppSettings[factoryName];
              Type factoryType = Type.GetType(factoryTypeAsString);
              return  (ILoggerFactory)Activator.CreateInstance(factoryType);
       }
```

Step 2: Modify your Main() to use this method.

```
ILoggerFactory factory = LoadFactory("ConsoleFactory");
      ILogger logger1 =  factory.CreateLogger();
```

Step 3: Do the same for FIleLogger.

Perform the same steps so the FileLogger can be accessed via the config file and modify main to use the LoadFactory() to access a FileLogger

The client is now coded entirely in terms of 'interfaces'.

Reflection

1. We have seen the Factory Pattern be implemented using Reflection, Console Input, and now the app.confile file. What are some other places you can think to set to have the factory choose which classes to implement?

   _____

   _____

   _____

   _____

2. Referring to the previous question, which method did you like the best? Why?

   _____

   _____

   _____

   _____

3. Look through some of the various methods in the .NET Framework. What class is an example that instantiates itself via a factory and why do you think they did it that way instead of the normal way of instantiating class?

   _____

   _____

   _____

   _____

   _____