Factory Patterns

Estimated time for completion: 60 minutes

Overview:

Goals:

- Utilize the Factory Method pattern
- Utilize the Abstract Factory pattern
- Refactor some smelly code to use the abstract factory pattern

Lab Notes:

Part 1: Create a Factory with that uses the Factory Method pattern

In this part of the lab you will be working with classes that implement logging. Currently the client of the logger creates the logging classes directly; you will change that so that the client uses factory methods to create the logger.

At this point the client will still need to know the type of factory to create, so the last step of the lab will be to create a configuration file that tells the client which factory to use

1. Open the solution and familiarize yourself with the project.
2. There is a client console application which consumes logging services from a Logger class library.
3. The Logger class library contains a ConsoleLogger class. This 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.
4. The library also contains a FileLogger class. This has constructors which take a file name as a construction parameter as well as the optional LogLevel
5. In this lab you will refactor the code in the library to use the Factory Method pattern.
6. The first thing you need to do is to change the client so that it will use the new factory.
    a. Find where the ConsoleLogger is used (this is in the Client project's Program.cs file).
    b. You need to change this from using

```
new ConsoleLogger()
```

    c. To using a factory. The new code will look something like this

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

    d. Change the code. Obviously this code will not compile yet.
7. Now that you have broken the client, you need to fix the code. The fix is to create the ConsoleLoggerFactory that the client now wants to use.
    a. Define a ConsoleLoggerFactory class and implement the CreateLogger method on the class.
    b. The CreateLogger method will simply create a new ConsoleLogger and return it to the caller.
    c. Build and test the code, it should now compile and still work as before.
    d. Now add a new creation method to the factory. This method will replace the ConsoleLogger constructor that takes a parameter. Add a creation method to the factory that has the same signature as the constructor you are replacing.
    e. Update the client to use this new factory method. Build and run the code, it should work exactly the same as before.
    f. 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 ConsoleLogger():this(LogLevel.INFO)
{}

internal ConsoleLogger(LogLevel level)
{ ... }
```

    g. If your client is still using new ConsoleLogger() that code will now fail to compile.
8. You now need to perform the same steps as above for the FileLogger
    a. Add a FileLoggerFactory
    b. Move the constructors from FileLogger to FileLoggerFactory
    c. Make the FileLogger constructors internal
    d. Change the client to use FileLoggerFactory
    e. Build and test the code, it should work as before
9. You still have not implemented the Factory Method pattern (yet). At the moment you have two unrelated factories. Remember that the factory method pattern provides an abstract factory class that creates a single type, and that the types and factories have a one to one relationship.
10. To implement that here you need to provide a common base class for the factory and a common base class for the loggers.
    a. Create a new abstract base class called Logger. This will be the vase for your ConsoleLogger and your FileLogger
    b. This class should contain definitions of the four Log methods used in both ConsoleLogger and FileLogger, make these methods abstract.

```
public abstract class Logger
{
    public abstract void Log(string text);
    public abstract void Log(LogLevel level, string text);
```



```
    public abstract void Log(Exception e);
    public abstract void Log(LogLevel level, Exception e);
}
```

c. Change your ConsoleLogger class so that it derives from Logger, change the Log methods to specify that they override the base class methods.

```
override public void Log(string text)
{ ... }

override public void Log(LogLevel level, string text)
{ ... }

override public void Log(Exception e)
{ ... }

override public void Log(LogLevel level, Exception e)
{ ... }
```

d. Change the client so that it declares uses of Logger rather than ConsoleLogger.
e. Refactor FileLogger so that it too derives from Logger
f. Change the client to use Logger declarations instead of FileLogger.
g. You now need to change the factories to use a common base class.
h. Add an abstract class called LoggerFactory and make the two existing factories derive from it.
i. In the abstract class add two new abstract methods

```
public abstract Logger CreateLogger();
public abstract Logger CreateLogger(LogLevel level);
```

j. Override these two methods in both the ConsoleLoggerFactory and FileLoggerFactory classes.
k. Change the client to reference the LoggerFactory (like this: LoggerFactory factory = new ConsoleLoggerFactory();) making sure you do this for all the factory definitions.

```
LoggerFactory factory = new ConsoleLoggerFactory();
Logger logger1 = factory.CreateLogger();
...

Logger logger2 = factory.CreateLogger(LogLevel.ERROR);
...
LoggerFactory fileFactory = new FileLoggerFactory();
Logger flogger1 = fileFactory.CreateLogger();
```

l. Build and run the code and it should still work.

11. You are almost there, however there is still an issue. The FileLoggerFactory has two extra creation methods that take a filename. There are no methods that correspond to these on the ConsoleLoggerFactory. You need to add these methods to the Factory class

so that the client can create the FileLogger instance when specifying the log file name, the ConsoleLoggerFactory overrides of these methods will simply ignore the parameters.

a. Add two new creation method definitions to the LoggerFactory class. These definitions will be based on the definitions in the FileLoggerFactory class.
b. Change the implementation of FileLoggerFactory to override these methods
c. Add implementatations of these methods to ConsoleLoggerFactory

```
public override Logger CreateLogger(string logFileName)
{
    return new ConsoleLogger();
}

public override Logger CreateLogger(string logFileName, LogLevel level)
{
    return new ConsoleLogger(logLevel);
}
```

d. Change the client to exercise these methods. To do this, use a factory method that takes a filename as a parameter.

12. The client still has to know the type of the factory to use 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 applications configuration file. As a final (optional) step you will add a configuration file and have the client read the configuration data from this file that it needs to create the factory.
    a. Add an app.config file to your client project
    b. This file needs an <code><appSettings></code> section with the key set to 'Factory'. The value of this entry should contain both the name of the DLL that contains the factory and the name of the class within that DLL that implements the factory. Customarily these are stored as a comma separated string. Add this value now.

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

c. Add a reference to the System.Configuration namespace to the Client project; you will need this to read the configuration file.
d. Add a static method to the client called LoadFactory>, this method takes a single string parameter (the key to use to read from appSettings) and returns a LoggerFactory.
e. This method should
    - Read the appsetings value from the configuration file
    - Separate the DLL name from the type name (Split is your friend here)
    - Load the assembly using Assembly.Load
    - Create the instance by using asm.CreateInstance where asm is a reference to the loaded assembly
    - Return this created instance to the caller

```
private static LoggerFactory LoadFactory(string factoryName)
{
    string factoryTypeDetails =
   ConfigurationManager.AppSettings[factoryName];
    string[] typeDetails;
    typeDetails = factoryTypeDetails.Split(new char[] { ',' });

    Assembly asm = Assembly.Load(typeDetails[0]);
    return (LoggerFactory)asm.CreateInstance(typeDetails[1]);
}
```

f. Once the code has been written, replace the call to new ConsoleFileFactory with a call to LoadFactory.
g. Run the code. It should work exactly the same, except now you have moved the factory definition into the configuration file.

13. The client should now be coded entirely in terms of 'interfaces'.