Template and Strategy Patterns

Estimated time for completion: 60 minutes

Overview:

This lab will consists of three parts the first two will be used to build a simple template and strategy solution. The third part involves you refactoring some smelly code to make use of the strategy pattern.

Goals:

- Utilize the Template and Strategy pattern
- Refactor some smelly code which has overuse of switch/case to use the strategy pattern

Lab Notes:

Part 1: Utilising the Template pattern

1. Open the solution (Valuation.sln) and familiarize yourself with the project. The project is relatively simple it consists of a simple bank account class and some code to produce a projected return on an initial investment of 5000. Compile and run it..
1. Now the bank has discovered that its competitors offer a variety of savings accounts some with tax free incentives and some with higher rates of interest the more you save. The bank would also like to offer similar products, the bank would like to offer:
    a. Classic account paying a standard 6% interest but having tax collected at source at a rate of 25%
    b. A Tax Free account paying a standard 6% interest but no tax collected
    c. A variable interest account paying 5% for balances up to 1000, 6% for balances up to 6000 and 8% for balances >= 6000. In all cases tax is collected at source at a rate of 25%
2. Your task is to implement these various bank account types, such that the bank can show customers forecasts for all these two new account types too, allowing the customer to select the appropriate account for them.
3. Each bank account is similar in that it takes deposits and has a balance; the thing that makes them different is how interest is calculated. However the structure of formulae for calculating the Yearly Interest is the same balance + ((balance * interestRate) * (1 - taxRate)), thus this could be encapsulated in a template method, with each part of the formulae that varies encapsulated as a different method.
4. Refactor the BankAccount class to be an abstract class containing a template method PayYearlyInterest, introduce abstract properties for TaxRate and InterestRate and use them in the PayYearlyInterest method. Create a ClassicAccount that derives from BankAccount and implement the abstract properties with the values from the original formulae.

5. Once you are happy the original functionality is working again, add two additional classes to represent the two new bank account types, making them derive from BankAccount. Add them as additional parameters to the PrintAccountForecast call. You should now have three different types of bank account all giving different returns on an initial 5k investment. What you have achieved though is that you have separated what varies from what stays the same and also closed the BankAccount type definition whilst keeping it open for extension.

Part 2: Strategy Pattern

For this part of the lab you will look at extending an application that provides a single stock market report to provide multiple reports. Currently the application simply looks for high swings, and simply outputs the raw data to the console. You have now been asked to produce another report that outputs exceptionally high volume days. You could be a cut and paste pirates and modify the existing report, but you know that smells bad. Your goal here is to refactor the code to separate what varies from what stays the same. In this case that's the filter for deciding which trading days are present in the report. To separate these areas of code you will use the strategy pattern.

1. Load the StockReport solution and familiarize yourself with the solution. Run the code to see the generated reports.
2. Ok now you are comfortable with the code consider the first goal: to separate the code that iterates over each trading day from the code that determines if the day should go into the report. You do this so that you can refactor the basic report iteration code into a method, and then supply this method at runtime with the appropriate behavior for filtering the days. This can be achieved using the strategy pattern, the behavior you need to supply is a piece of logic that simply examines a supplied trading day and returns true or false depending if it should be included in the report or not. Some pseudo code

```
1. Foreach tradingDay in tradingDays
2.     If ( FilterStrategy( tradingDay ) = true ) Then
3.         Output Trading Day
4.     End  If
5. End
```

3. Now create a new type to represent this type of strategy, remember you can use a delegate or an abstract class or interface. In this case one of them seems to stand out. Your strategy definition will have a method that looks like bool ReportOnTradingDay(TradingDay day)
4. Now create a concrete version of the strategy that uses the filtering logic currently supplied in the report. If you chose the delegate route that means writing a method that matches the delegate signature, if you went the abstract class approach create a new class derived from your strategy and implement the abstract method.
5. Refactor the report method to take an instance of a strategy, now make the foreach loop use the supplied strategy to filter the trading days. Modify the Main method to create an instance of the strategy and supply it to the refactored report method. Compile it and run

it, you should now be back to the same level of functionality you started with, but refactored to allow you to easily extend.

6. A new report you have been asked to produce is one that shows days with high volume. Add a new strategy that can locate days where more than 20 million shares have been traded. Note that this should not result in any change to the report method.
7. If you did go down the delegate route (almost certainly the right one), you did not have to declare a delegate type, you could have used the built in generic delegate type called Predicate. Predicate is a method that takes a single input and returns either true or false.

```
namespace System
{
    // Summary:
    //     Represents the method that defines a set of criteria and
   determines whether
    //     the specified object meets those criteria.
    //
    // Returns:
    //     true if obj meets the criteria defined within the method
   represented by this
    //     delegate; otherwise, false.
    public delegate bool Predicate<T>(T obj);
}
The additional advantage of using a delegate is that you can make use of
   anonymous methods, so in the case of detecting a high volume, you could
   have placed the code inline.

ReportTradingDays(tradingDays, delegate(TradingDay day) { return day.Volume >
   20000000; });
```

8. Delegates work well when a strategy is composed of just a single method.

Part 3: Smelly Code...

In this part your task is to refactor a piece of code that is currently littered with similar switch blocks to different scenarios. Having lots of similar conditional if/else or switch blocks is often a sign that refactoring to the strategy pattern would make the code simpler and allow it easily be extended since it is clear what functionality the core algorithm requires. The piece of code that is going to be refactored is a console application that has been written to allow the conversion of numbers from one base to another. The user first picks the source and destination base and then enters a number in terms of the source base. On pressing return the number is reprinted in requested destination base, checks are made whilst the user is entering the number, to ensure that each digit is valid for that base in question.

1. Load the BaseConversion solution and familiarize yourself with the code, run the code, try out a simple conversion of Hex to Dec, and type in FF
2. Looking through the code you should be able to see similar switch blocks used to decide which behavior to use at various points in the algorithm. Your first task is to identify the pieces of behavior that vary and construct a abstract class that has methods that represent

this functionality. Hint, the things that vary are determining if a character is a valid digit, parsing a string to an integer and converting an integer to a string for the desired base.

3. Once you have defined your abstract class to represent the strategy, produce an implementation for each of the numeric bases supported. All the necessary logic should be taken from the various case statement entries.
4. Now you need to refactor the core algorithm to be written in terms of the base strategy type. Change the ConvertFromToBase method not to accept two Base's put accept two strategies, one for source base and one for destination base. Rewrite ConvertFromBase and GetNumbericValue not to use switch/case but to use the supplied strategies. Change the call in Main to ConvertFromToBase to pass simply pass null for both parameters and check everything compiles.
5. So now you should have code which contains no switch/case blocks, any conditional behaviour is handled by the supplied strategy. Thus you have separated what stays the same from what varies. All you need to do now is pass instances of the appropriate strategies into the ConvertFromToBase method call in Main. You could do this either by creating a creational method that has a similar switch/case statement you have been replacing or alternatively create a dictionary that maps between the Base enumeration value and an object to implement the appropriate strategy.

```
Dictionary<Base,NumberBaseStrategy> numberBaseStrategies = new
   Dictionary<Base,NumberBaseStrategy>();
// Repeat for all strategies
numberBaseStrategies[Base.Hex] = new HexBaseStrategy();
....
while (true)
{
  ConvertFromToBase(numberBaseStrategies[sourceBase],numberBaseStrategies[
   destBase]);
}
```

6. You should be able to see that by separating the various pieces that vary from the parts that do not, not only is the code clearer to read, but when wishing to extend it to support an additional base it is clear what you need to do, and does not require extensive changes to existing working code.