TEMPLATE METHOD & STRATEGY: Inheritance vs. Delegation

"The best strategy in life is diligence."

-- Chinese Proverb

Way, way back in the early 90s -- back in the early days of OO -- we were all quite taken with the notion of inheritance. The implications of the relationship were profound. With inheritance we could *program by differenc*e! That is, given some class that did something almost useful to us, we could create a subclass and change only the bits we didn't like. We could reuse code simply by inheriting it! We could establish whole taxonomies of software structures; each level of which reused code from the levels above. It was a brave new world.

Like most brave new worlds, this one turned out to be a bit too starry-eyed. By 1995 it was clear that inheritance was very easy to over-use; and that over-use of inheritance was very costly. Gamma, Helm, Johnson, and Vlissides went so far as to stress: "*Favor object composition over class inheritance.*"[1] So we cut back on our use of inheritance, often replacing it with composition or delegation.

This chapter is the story of two patterns that epitomize the difference between inheritance and delegation. TEMPLATE METHOD and STRATEGY solve similar problems, and can often be used interchangeably. However, TEMPLATE METHOD uses inheritance to solve the problem, whereas STRATEGY uses delegation.

1. [GOF95], p20

Intent of Template Method and Strategy

TEMPLATE METHOD and STRATEGY both solve the problem of separating a generic algorithm from a detailed context. We see the need for this very frequently in software design. We have an algorithm that is generically applicable. In order to conform to the Dependency Inversion Principle (DIP) we want to make sure that the generic algorithm does not depend upon the detailed implementation. Rather we want the generic algorithm and the detailed implementation to depend upon abstractions.

TEMPLATE METHOD

Consider all the programs you have written. Many probably have this fundamental main-loop structure.

```
Initialize();
while (!done()) // main loop
{
  Idle();       // do something useful.
}
Cleanup();
```

First we initialize the application. Then we enter the main loop. In the main loop we do whatever the program needs to do. We might process GUI events, or perhaps process database records. Finally, once we are done, we exit the main loop and clean up before we exit.

This structure is so common that we can capture it in a class named `Application`. Then we can reuse that class for every new program we want to write. Think of it! We never have to write that loop again![2]

For example, consider Listing 15-1. Here we see all the elements of the standard program. The `InputStreamReader` and `BufferedReader` are initialized. There is a main loop that reads Fahrenheit readings from the `BufferedReader` and prints out Celsius conversions. At the end, an exit message is printed.

Listing 15-1
ftoc raw

```
import java.io.*;
public class ftocraw
{
  public static void main(String[] args) throws Exception
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    boolean done = false;
    while (!done)
    {
      String fahrString = br.readLine();
      if (fahrString == null || fahrString.length() == 0)
```

2. I've also got this bridge I'd like to sell you.

Listing 15-1 (Continued)
ftoc raw

```
          done = true;
        else
        {
          double fahr = Double.parseDouble(fahrString);
          double celcius = 5.0/9.0*(fahr-32);
          System.out.println("F=" + fahr + ", C=" + celcius);
        }
      }
      System.out.println("ftoc exit");
    }
  }
```

This program has all the elements of the main-loop structure above. It does a little initialization, does its work in a main-loop, and then cleans up and exits.

We can separate this fundamental structure from the `ftoc` program by employing the TEMPLATE METHOD pattern. This pattern places all the generic code into an implemented method of an abstract base class. The implemented method captures the generic algorithm, but defers all details to abstract methods of the base class.

So, for example, we can capture the main loop structure in an abstract base class called `Application`. See Listing 15-2

Listing 15-2
Application.java

```
public abstract class Application
{
  private boolean isDone = false;

  protected abstract void init();
  protected abstract void idle();
  protected abstract void cleanup();

  protected void setDone()
  {isDone = true;}

  protected boolean done()
  {return isDone;}

  public void run()
  {
    init();
    while (!done())
      idle();
    cleanup();
  }
}
```

This class describes a generic main loop application. We can see the main loop in the implemented `run` function. We can also see that all the work is being deferred to the abstract methods `init`, `idle`, and `cleanup`. The `init` method takes care of any initialization we need done. The `idle` method does the main work of the program and will be called repeatedly until `setDone` is called. The `cleanup` method does whatever needs to be done before we exit.

We can rewrite the `ftoc` class by inheriting from `Application` and just filling in the abstract methods. Listing 15-3 show what this looks like.

Listing 15-3
`ftocTemplateMethod.java`

```
import java.io.*;
public class ftocTemplateMethod extends Application
{
  private InputStreamReader isr;
  private BufferedReader br;

  public static void main(String[] args) throws Exception
  {
    (new ftocTemplateMethod()).run();
  }

  protected void init()
  {
    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);
  }

  protected void idle()
  {
    String fahrString = readLineAndReturnNullIfError();
    if (fahrString == null || fahrString.length() == 0)
      setDone();
    else
    {
      double fahr = Double.parseDouble(fahrString);
      double celcius = 5.0/9.0*(fahr-32);
      System.out.println("F=" + fahr + ", C=" + celcius);
    }
  }

  protected void cleanup()
  {
    System.out.println("ftoc exit");
  }

  private String readLineAndReturnNullIfError()
  {
    String s;
    try
    {
      s = br.readLine();
    }
    catch(IOException e)
    {
      s = null;
    }
    return s;
  }
}
```

Dealing with the exception made the code get a little longer, but it's easy to see how the old `ftoc` application has been fit into the TEMPLATE METHOD pattern.

Pattern Abuse. By this time you should be thinking *"Is he serious? Does he really expect me to use this `Application` class for all new apps? It hasn't bought me anything, and it's overcomplicated the problem."*

Er...,Yeah.. :^(

I chose the example because it was simple, and provided a good platform for showing the mechanics of TEMPLATE METHOD. On the other hand, I don't really recommend building `ftoc` like this.

This is a good example of pattern abuse. Using TEMPLATE METHOD for this particular application is ridiculous. It complicates the program and makes its bigger. Encapsulating the main loop of every application in the universe sounded wonderful when we started; but the practical application is fruitless in this case.

Design patterns are wonderful things. They can help you with many design problems. But the fact that they exist does not mean that they should always be used. In this case, while TEMPLATE METHOD was applicable to the problem, its use was not advisable. The cost of the pattern was higher than the benefit it yielded.

So lets look at a slightly more useful example.

Bubble Sort[3]

See Listing 15-4.

Listing 15-4

BubbleSorter.java

```
public class BubbleSorter
{
  static int operations = 0;
  public static int sort(int [] array)
  {
    operations = 0;
    if (array.length <= 1)
      return operations;

    for (int nextToLast = array.length-2;
         nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
        compareAndSwap(array, index);

    return operations;
  }
```

3. Like Application, Bubble Sort is easy to understand, and so makes a useful teaching tool. However, no one in their right mind would ever actually use a Bubble Sort if they had any significant amount of sorting to do. There are *much* better algorithms.

Listing 15-4 (Continued)
BubbleSorter.java

```
  private static void swap(int[] array, int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  private static void compareAndSwap(int[] array, int index)
  {
    if (array[index] > array[index+1])
      swap(array, index);
    operations++;
  }
}
```

The `BubbleSorter` class knows how to sort an array of integers using the bubble sort algorithm. The `sort` method of `BubbleSorter` contains the algorithm that knows how to do a bubble sort. The two ancillary methods: `swap` and `compareAndSwap` deal with the details of integers and arrays, and handle the mechanics that the `sort` algorithm requires.

Using the TEMPLATE METHOD pattern we can separate the bubble sort algorithm out into an abstract base class named `BubbleSorter`. `BubbleSorter` contains an implementation of the `sort` function that calls an abstract method named `outOfOrder` and another called `swap`. The `outOfOrder` method compares two adjacent elements in the array and returns `true` if the elements are out of order. The `swap` method swaps two adjacent cells in the array.

The `sort` method does not know about the array, nor does it care what kind of objects are stored in the array. It just calls `outOfOrder` for various indices into the array, and determines whether those indices should be swapped or not. See Listing 15-5.

Listing 15-5
BubbleSorter.java

```
public abstract class BubbleSorter
{
  private int operations = 0;
  protected int length = 0;

  protected int doSort()
  {
    operations = 0;
    if (length <= 1)
      return operations;

    for (int nextToLast = length-2;
         nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (outOfOrder(index))
          swap(index);
        operations++;
```

Listing 15-5 (Continued)
BubbleSorter.java

```
      }

    return operations;
  }

  protected abstract void swap(int index);
  protected abstract boolean outOfOrder(int index);
}
```

Given `BubbleSorter` we can now create simple derivatives that can sort any different kind of object. For example, we could create `IntBubbleSorter` which sorts arrays of integers, and `DoubleBubbleSorter` which sorts arrays of doubles. See Figure 15-1, Listing 15-6, and Listing 15-7.

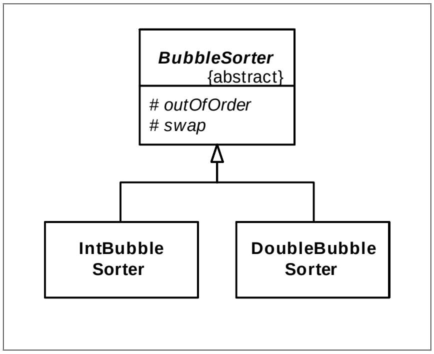

Figure 15-1
Bubble Sorter Structure

Listing 15-6
IntBubbleSorter.java

```
public class IntBubbleSorter extends BubbleSorter
{
  private int[] array = null;
  public int sort(int [] theArray)
  {
    array = theArray;
    length = array.length;
    return doSort();
  }

  protected void swap(int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  protected boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
```

Listing 15-6 (Continued)

IntBubbleSorter.java

```
  }
}
```

Listing 15-7

DoubleBubbleSorter.java

```
public class DoubleBubbleSorter extends BubbleSorter
{
  private double[] array = null;
  public int sort(double [] theArray)
  {
    array = theArray;
    length = array.length;
    return doSort();
  }

  protected void swap(int index)
  {
    double temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  protected boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```

The TEMPLATE METHOD pattern shows one of the classic forms of reuse in object oriented programming. Generic algorithms are placed in the base class and inherited into different detailed contexts. But this technique is not without its costs. Inheritance is a very strong relationship. Derivatives are inextricably bound to their base classes.

For example, the `outOfOrder` and `swap` functions of `IntBubbleSorter` are exactly what are needed for other kinds of sort algorithms. And yet, there is no way to reuse `outOfOrder` and `swap` in those other sort algorithms. By inheriting `BubbleSorter` we have doomed `IntBubbleSorter` to forever be bound to `BubbleSorter`. The STRATEGY pattern provides another option.

STRATEGY

The STRATEGY pattern solves the problem of inverting the dependencies of the generic algorithm and the detailed implementation in a very different way. Consider, once again, the pattern abusing `Application` problem.

Rather than placing the generic application algorithm into an abstract base class, we place it into a *concrete* class named `ApplicationRunner`. We define the abstract methods that the generic algorithm must call within an interface named `Application`. We derive `ftocStrategy` from this interface, and pass it into the `ApplicationRunner`.

`ApplicationRunner` then delegates to this interface. See Figure 15-2, and Listing 15-8 through Listing 15-10.

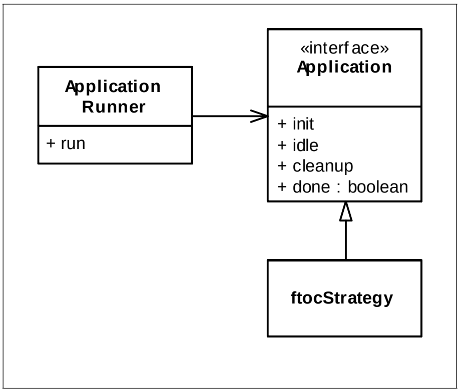

Figure 15-2
Strategy structure of the Application algorithm.

Listing 15-8
`ApplicationRunner.java`

```
public class ApplicationRunner
{
  private Application itsApplication = null;

  public ApplicationRunner(Application app)
  {
    itsApplication = app;
  }
  public void run()
  {
    itsApplication.init();
    while (!itsApplication.done())
      itsApplication.idle();
    itsApplication.cleanup();
  }
}
```

Listing 15-9
`Application.java`

```
public interface Application
{
  public void init();
  public void idle();
  public void cleanup();
  public boolean done();
}
```

Listing 15-10
`ftocStrategy.java`

```
import java.io.*;
public class ftocStrategy implements Application
```

Listing 15-10 (Continued)

ftocStrategy.java

```
{
  private InputStreamReader isr;
  private BufferedReader br;
  private boolean isDone = false;

  public static void main(String[] args) throws Exception
  {
    (new ApplicationRunner(new ftocStrategy())).run();
  }

  public void init()
  {
    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);
  }

  public void idle()
  {
    String fahrString = readLineAndReturnNullIfError();
    if (fahrString == null || fahrString.length() == 0)
      isDone = true;
    else
    {
      double fahr = Double.parseDouble(fahrString);
      double celcius = 5.0/9.0*(fahr-32);
      System.out.println("F=" + fahr + ", C=" + celcius);
    }
  }

  public void cleanup()
  {
    System.out.println("ftoc exit");
  }

  public boolean done()
  {
    return isDone;
  }

  private String readLineAndReturnNullIfError()
  {
    String s;
    try
    {
      s = br.readLine();
    }
    catch(IOException e)
    {
      s = null;
    }
    return s;
  }
}
```

It should be clear that this structure has both benefits and costs over the TEMPLATE METHOD structure. STRATEGY involves more total classes, and more indirection than TEMPLATE METHOD. The delegation pointer within `ApplicationRunner` incurs a slightly higher cost in terms of run-time and data space than inheritance would. On the other hand, if we had many different applications to run, we could reuse the `ApplicationRunner` *instance* and pass in many different implementations of `Application`; thereby reducing the code space overhead.

None of these costs and benefits are overriding. In most cases none of them matters in the slightest. In the typical case, the most worrisome is the extra class needed by the STRATEGY pattern. However, there is more to consider.

Sorting again.

Consider an implementation of the bubble sort that uses the STRATEGY pattern. See Listing 15-11 through Listing 15-13.

Listing 15-11

`BubbleSorter.java`

```
public class BubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandle itsSortHandle = null;

  public BubbleSorter(SortHandle handle)
  {
    itsSortHandle = handle;
  }

  public int sort(Object array)
  {
    itsSortHandle.setArray(array);
    length = itsSortHandle.length();
    operations = 0;
    if (length <= 1)
      return operations;

    for (int nextToLast = length-2;
         nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandle.outOfOrder(index))
          itsSortHandle.swap(index);
        operations++;
      }

    return operations;
  }
}
```

Listing 15-12
SortHandle.java

```
public interface SortHandle
{
  public void swap(int index);
  public boolean outOfOrder(int index);
  public int length();
  public void setArray(Object array);
}
```

Listing 15-13
IntSortHandle.java

```
public class IntSortHandle implements SortHandle
{
  private int[] array = null;

  public void swap(int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  public void setArray(Object array)
  {
    this.array = (int[])array;
  }

  public int length()
  {
    return array.length;
  }

  public boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```

Notice that the `IntSortHandle` class knows nothing whatever of the `BubbleSorter`. It has no dependency whatever upon the bubble sort implementation. This is not the case with the TEMPLATE METHOD pattern. Look back at Listing 15-6 and you can see that the `IntBubbleSorter` depended directly on `BubbleSorter`, the class that contains the bubble sort algorithm.

The TEMPLATE METHOD approach partially violates DIP. The implementation of the `swap` and `outOfOrder` methods depends directly upon the bubble sort algorithm. The STRATEGY approach contains no such dependency. Thus, we can use the `IntSortHandle` with `Sorter` implementations other than `BubbleSorter`.

For example, we can create a variation of the bubble sort that terminates early if a pass through the array finds it in order. (See Listing 15-14.). `QuickBubbleSorter` can also use `IntSortHandle`, or any other class derived from `SortHandle`.

Listing 15-14
QuickBubbleSorter.java

```
public class QuickBubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandle itsSortHandle = null;

  public QuickBubbleSorter(SortHandle handle)
  {
    itsSortHandle = handle;
  }

  public int sort(Object array)
  {
    itsSortHandle.setArray(array);
    length = itsSortHandle.length();
    operations = 0;
    if (length <= 1)
      return operations;

    boolean thisPassInOrder = false;
    for (int nextToLast = length-2;
         nextToLast >= 0 && !thisPassInOrder; nextToLast--)
    {
      thisPassInOrder = true; //potenially.
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandle.outOfOrder(index))
        {
          itsSortHandle.swap(index);
          thisPassInOrder = false;
        }
        operations++;
      }
    }

    return operations;
  }
}
```

Thus, the STRATEGY pattern provides one extra benefit over the TEMPLATE METHOD pattern. Whereas the TEMPLATE METHOD pattern allows a generic algorithm to manipulate many possible detailed implementations, by fully conforming to the DIP the STRATEGY pattern additionally allows each detailed implementation to be manipulated by many different generic algorithms.

Bibliography

[GOF95]: Design Patterns, Gamma, et. al., Addison Wesley, 1995

[PLOPD3]: Pattern Languages of Program Design 3, Robert C. Martin, et. al., Addison Wesley, 1998.