Tecniche di Progettazione: Design Patterns

GoF: Template method

Coffee

```
public class Coffee {
   void prepare Recipe() {
      boilWater();
      brewCoffeeGrinds();
      pourInCup();
      addSugarAndMilk();
}
public void boilWater() {
   System.out.println(“Boiling water”);
}
public void brewCoffeeGrinds {
   System.out.println(“Dripping Coffee through filter”);
}
public  void pourInCup() {
   System.out.println(“Pouring in cup”);
}
public void addSugarAndMilk() {
   System.out.println(“adding Sugar and Milk”);
}
```

Tea

```
public class Tea{
   void prepare Recipe() {
      boilWater();
      seepTeaBag();
      pourInCup();
      addLemon();
}
public void boilWater() {
   System.out.println(“Boiling water”);
}
public void seepTeaBag() {
   System.out.println(“Steeping the tea”);
}
public  void pourInCup() {
   System.out.println(“Pouring in cup”);
}
public void addLemon() {
   System.out.println(“adding Lemon”);
}
```

Clearly there is code duplication. Do you think this is a contrived example?

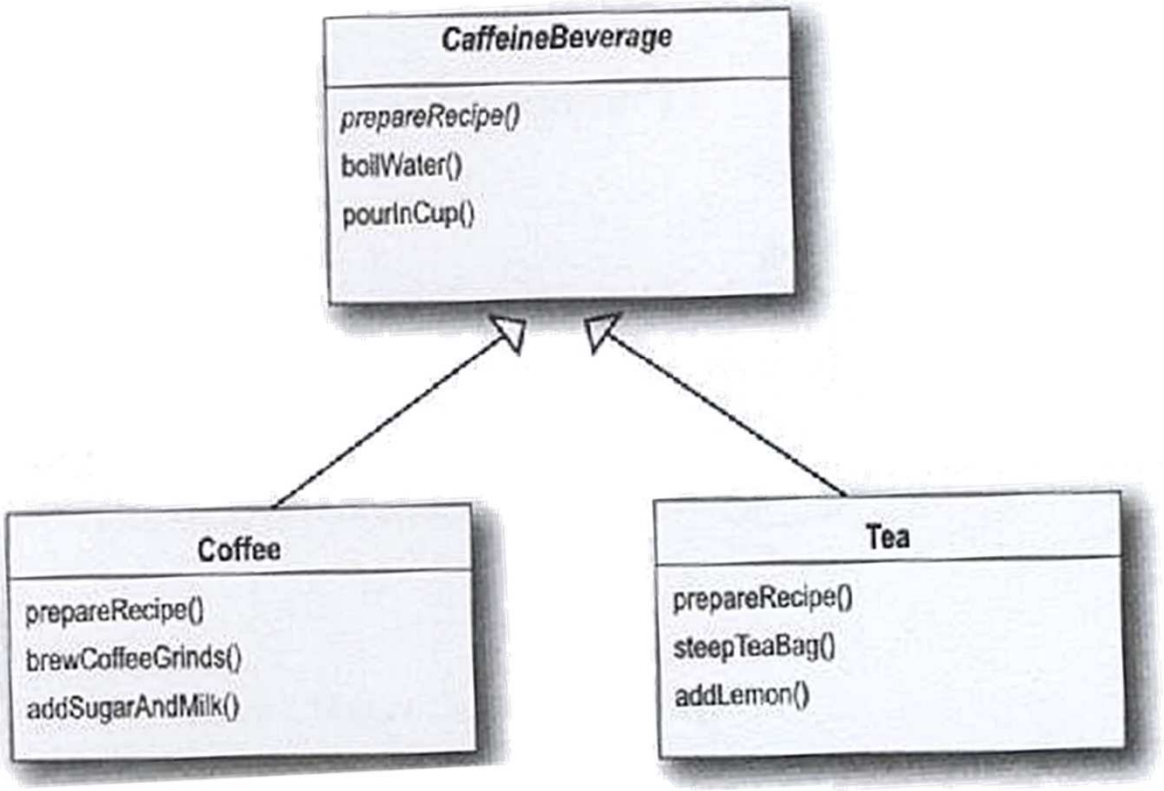
Wouldn't you really say:

```
public void boilWater() {
   System.out.println("Boiling water for coffee");
}

public void boilWater() {
   System.out.println("Boiling water for tea");
}
```

As well as:

```
public  void pourInCup() {
   System.out.println("Pouring coffee in cup");

public  void pourInCup() {
   System.out.println("Pouring tea in cup");
```

Let's abstract Coffee and Tea



Both recipes follow the same algorithm

1. Boil some water.
2. Use the hot water to extract the coffee or tea.
3. Pour the resulting beverage into a cup.
4. Add the appropriate condiments to the beverage.

Compare the two prepareRecipe() methods

```
void prepareRecipe() {
   boilWater();
   brewCoffeeGrinds();
   pourInCup();
   addSugarAndMilk();
}
```

```
void prepareRecipe() {
   boilWater();
   SeepTeaBag();
   pourInCup();
   addLemon();
}
```

These can be rewritten as:

```
void prepareRecipe() {
   boilWater();
   brew();
   pourInCup();
   addCondiments();
}
```

Is this better?

class CaffeineBeverage

If we implement the new prepare recipe, we now have:

```
public abstract class CaffeineBeverage {

final void prepareRecipe() {
   boilWater();
   brew();
   pourInCup();
   addCondiments();
}

abstract void brew();
abstract void addCondiments();

void boilWater() {
   System.out.println(“Boiling water”);
}

void pourInCup() {
   System.our.println(“Pouring in cup”);
}
```

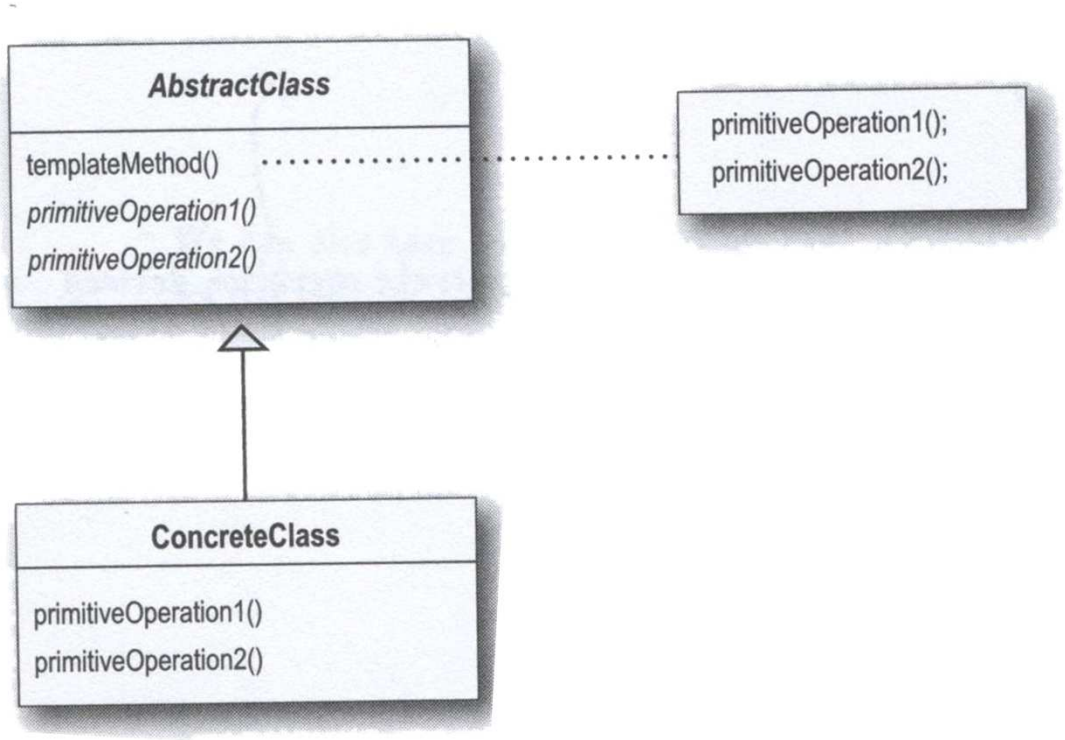
Tea & Coffee

```
public class Tea extends CaffeineBeverage {
   public void brew() {
      System.out.println(“Steeping the tea”);
}
   public void addCondiments() {
      System.out.println(“Adding Lemon”);
}
```

```
public class Coffee extends CaffeineBeverage {
   public void brew() {
      System.out.println(“Dripping Coffee through filter”);
}
   public void addCondiments() {
      System.out.println(“Adding Sugar and Milk”);
}
```

Template Method Pattern

- The Template method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.



Template Method Pattern

```
abstract class AbstractClass {
   final void templateMethod() {
      primitiveOperation1();
      primitiveOperation2();
      concreteOpertation();
}

   abstract void primitiveOperation1();
   abstract void primitiveOperation2();
   void concreteOperation(); {
      //implementaiton here
   }
}
```

Applicability

- Use the Template Method pattern:
  - To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - To localize common behavior among subclasses and place it in a common class (in this case, a superclass) to avoid code duplication. This is a classic example of "code refactoring."
  - To control how subclasses extend superclass operations. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.
- *The Template Method is a fundamental technique for code reuse.*

Implementation Issues

- Operations which must be overridden by a subclass should be made abstract
- If the template method itself should not be overridden by a subclass, it should be made final
- To allow a subclass to insert code at a specific spot in the operation of the algorithm, insert "hook" operations into the template method. These hook operations may do nothing by default.
- Try to minimize the number of operations that a subclass must override, otherwise using the template method becomes tedious for the developer
- In a template method, the parent class calls the operations of a subclass and not the other way around.
  - This is an inverted control structure that's sometimes referred to as "the Hollywood principle," as in, "Don't call us, we'll call you".

Hook

```
abstract class AbstractClass {
   final void templateMethod() {
      primitiveOperation1();
      primitiveOperation2();
      hook();
      concreteOpertation();
  }
   abstract void primitiveOperation1();
   abstract void primitiveOperation2();
   void concreteOperation() {
      //implementaiton here
   }
   public void hook() {}    // Do nothing hook method.
}
```

The hook method does nothing, but provides an interface so that subclasses may override them, although this is not required.

Placing a Hook

```
public abstract class CaffeineBeverage {
      final void prepareRecipe() {
         boilWater();
         brew();
         pourInCup();
         if (customerWantsCondiments()) {addCondiments();}
      }
      abstract void brew();
      abstract void addCondiments();
      void boilWater() {
         System.out.println(“Boiling water”);
      }

      void pourInCup() {
         System.our.println(“Pouring in cup”);
      }

       public boolean customerWantsCondiments() {
         return true;
      }
}
```

Implementing the Hook

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {
    public void brew() {
        System.out.println(“Dripping Coffee through filter”);
    }
    public void addCondiments() {
          System.out.println(“Adding Sugar and Milk”);
    }
    public boolean customerWantsCondiments() {
        String answer = getUserInput();
        if ( answer.toLowerCase().startsWith(“y”))  return true;
            else return false;
    }
    private String GetUserInput() {
        String answer = “no”;
        System.out.print(“Wanna milk and sugar with your coffee (y/n?”);
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try { answer = in.readLine(); }
            catch (IOException ioe) {
                System.err.println(“IO error trying to read your answer”);
            }
        return answer;
    }
}
```

Sorting with the Template Method

```
public static void sort(Object[] a) {
   Object aux[] = (Object[])a.clone();
   mergeSort(aux, a, 0, a.length, 0);
}

private static void mergeSort(Object[] src, Object[] dest,
     int low, int high, int off) {
     int length = high - low;

     // Exit if already sorted
     // Recursively sort halves of dest into src
     // Merge sorted halves (now in src) into dest
     for (int i=destLow, p=low, q=mid; i < destHigh; i++) {
          if (q >= high || p < mid
          && ((Comparable) src[p]).compareTo(src[q]) <= 0)
                dest[i] = src[p++];
           else
                dest[i] = src[q++];
           }
        }
```

Comparable Ducks (by weight)

```
public class Duck implements Comparable {
      String name;
      int weight;

      public Duck(String name, int weight) {
         this.name = name;
         this.weight = weight;
      }
      public String toString() {
         return name + “weighs “ + weight;
      }

      public int compareTo(Object object) {
         Duck otherDuck = (Duck)object;
         if (this.weight < otherDuck.weight) {
                  return -1;
         } else if (this.weight > otherDuck.weight) {
                  return 1;
      } else { return She is a witch, may we burn her? }
      }
}
```

Sorting Ducks

```
public class DuckSortTestDrive {
   public static void main (String[] args) {
      Duck[] ducks = {
         new Duck(“Daffy”, 8),
         new Duck(“Dewey”, 2),
         new Duck(“Howard”, 7)};
         System.outprintln(“Before sorting:”);
         display(ducks);
         Arrays.sort(ducks);
         System.out.println(“\nAfter sorting:”);
         display(ducks);
      }

   public static void display(duck[] ducks) {
      for (int i = 0; i < ducks.length; i++) {
         System.out.println(ducks[i]);
     }
   }
}
```

Template method & Java generics

```
abstract class GenerifiedTemplate<T> {
  public void templateMethod() { f(); g(); }
  public abstract void f();
  public abstract void g();
}
class Subtype extends GenerifiedTemplate<Subtype> {
  public void f() { System.out.println("f()"); }
  public void g() { System.out.println("g()"); } }

public class GenericTemplateMethod {
  public static void main(String[] args) {
       new Subtype().templateMethod(); }
} /* Output: f() g() */
```

I can't see the advantage wrt simply overriding an abstract class.

Template method & Java generics cnt'd

But suppose you add these methods to the base class:

```
abstract T foo();
abstract T bar();
```

The subclass then overrides it:

```
class Subclass extends GenerifiedTemplate<Subclass> {
  Subclass foo() { ... }
  Subclass bar() { ... }
  }
```

It's a way of specifying that the subclass should use covariance for the return values of these methods, and both methods should use the same subtype as the return value

However, nothing says that the subclass has to return itself. You could also do this:

```
class Subclass2 extends GenerifiedTemplate<AnotherSubclass> {
AnotherSubclass foo() { ... }
AnotherSubclass bar() { ... }
}
```

Without the generic type, there are no constraints on how the subclass uses covariance, if at all.