Tecniche di Progettazione: Design Patterns

GoF: Template method

Hint

- The underlying is idea not so different from Factory method
- Similarity: a common algorithm with specialization of the algorithm steps
- Difference: the specialized steps are «doing» methods and not factory methods

Coffee

```
public class Coffee {
   void prepareRecipe() {
         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 prepareRecipe() {
         boilWater();
         seepTeaBag();
         pourInCup();
         addLemon();
}
public void boilWater() {
   System.out.println("Boiling water");
}
public void brewCoffeeGrinds {
   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.

The only reason to do that is if you REALLY need to say:

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

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

As well as:

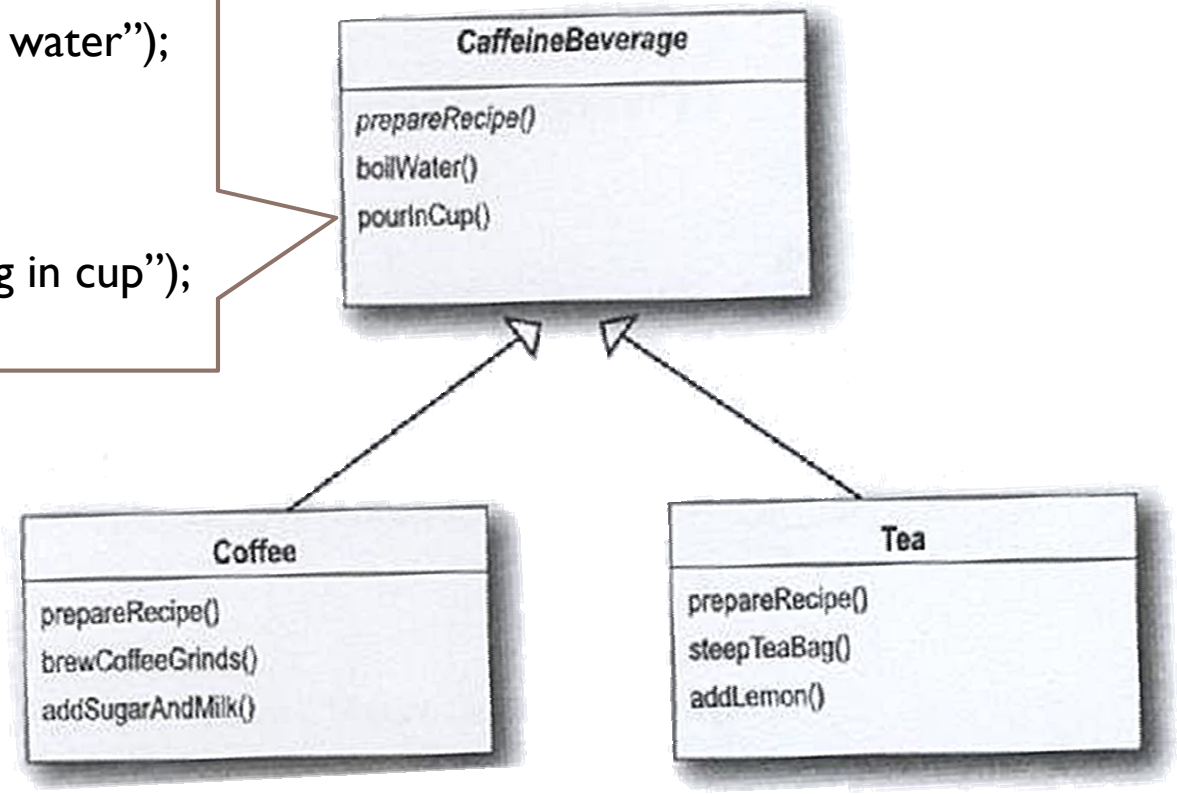
```
public  void pourInCup() {
   System.out.println(“Pouring coffee in cup”);

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

Let's simplify *boiling* and *pouring* and abstract Coffee and Tea

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

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



Moreover: (!)
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

```
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”);
      }
```

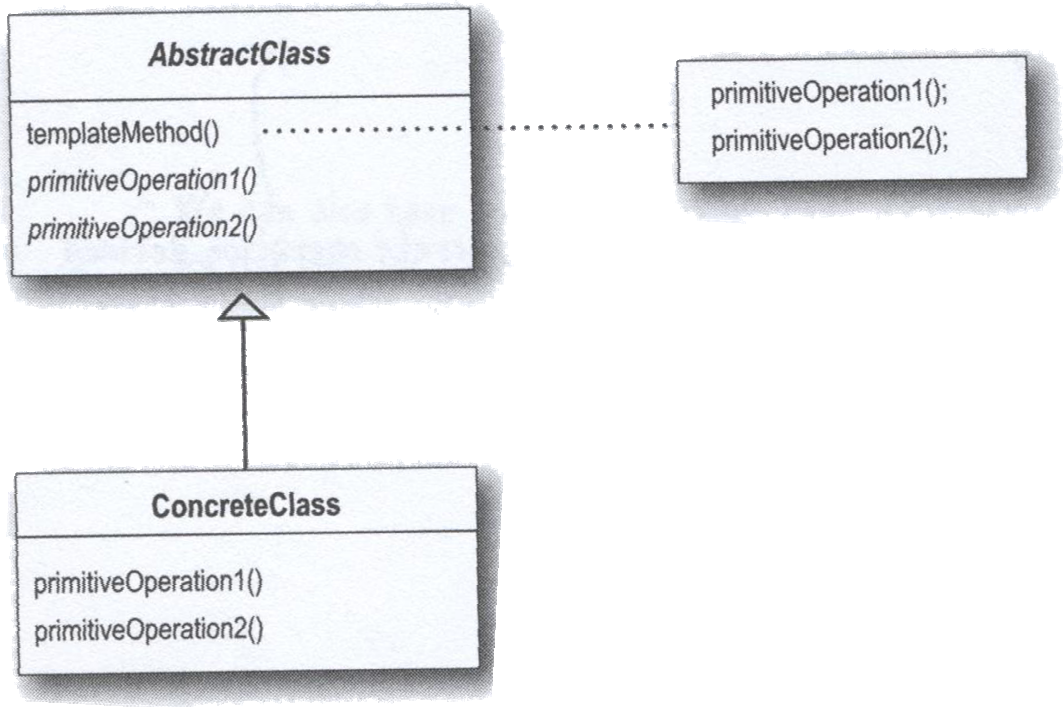
Tea & Coffee

```
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

- *The Template Method is a fundamental technique for code reuse.*
- 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 (we'll see "hook" methods in a while)

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
- 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".
- 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.

A **hook** is a method that is declared in the abstract class, but only given an empty or default implementation.

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

Another hook

```
public abstract class CaffeineBeverageWithHook {
     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;
    }
}
```

Example: 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++];
        }
     }
```

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() { ... }
  }
```

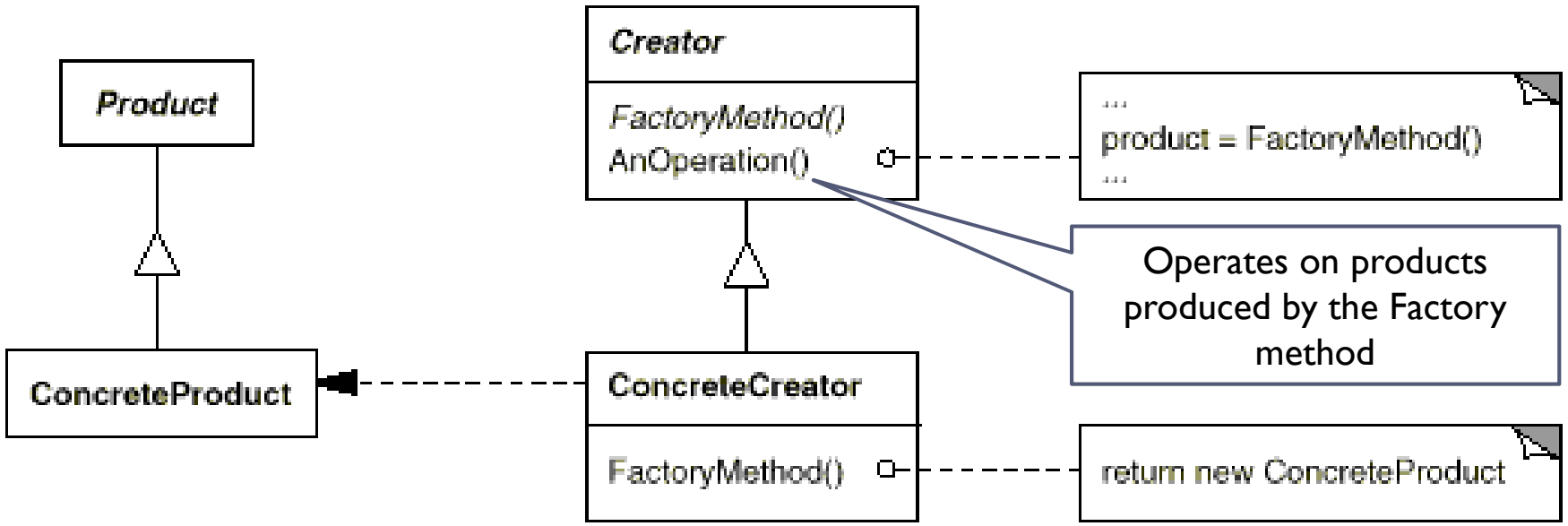
It's a way of specifying that the subclass should use covariance for the return values of these methods, and both methods should use the same subtype as the return value

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.

VS Factory Method Pattern



In the official definition:
Factory method lets the subclasses ***decide*** which class to instantiate
Decide: --not because the classes themselves decide at runtime
-- but because the creator is written without knowlwdge of the actual products that will be created, which is decided by the choice of the subclass that is usd

Recall the Pizza Factory Method

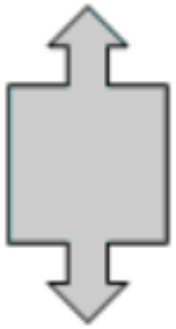
```
public abstract class PizzaStore {
        protected abstract createPizza(String type);
         public Pizza orderPizza(String type) {
              Pizza pizza = createPizza(type);
              pizza.prepare();
              pizza.bake();
              pizza.cut();
              pizza.box();
              return pizza;
              }
}
```

```
public class NYPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
            if (type.equals("cheese")) {
                return new NYCheesePizza();
            } else if (type.equals("greek")) {
                return new NYGreekPizza();
            } else if (type.equals("pepperoni")) {
                return new NYPepperoniPizza();
            }
            return null;
    }
}
```

Factory Method is a specialization of Template Method.

Suming up

Template Method



define skeleton, defer implementation

1) Evaluate an algorithm with multiple steps, and many possible implementations for one or more of those steps.

2) Where implementation is reusable, define it in a base class.

3) Where multiple implementations are possible: define "place holder" method signatures in the base class, and allocate each possible implementation to its own derived class.

Check list

1. Examine the algorithm, and decide which steps are standard and which steps are peculiar to each of the current classes.
2. Define a new abstract base class to host the "don't call us, we'll call you" framework.
3. Move the shell of the algorithm (now called the "template method") and the definition of all standard steps to the new base class.
4. Define an abstract (Java) or pure virtual (C++) method or a "hook" method in the base class for each step that requires many different implementations.
5. Invoke the abstract/hook method(s) from the template method.
6. Each of the existing classes declares an "is-a" relationship to the new abstract base class.
7. Remove from the existing classes all the implementation details that have been moved to the base class.
8. The only details that will remain in the existing classes will be the implementation details peculiar to each derived class.

Homework

- Use the pattern to
  - Refine a sorting algorithm for sorting in a different (ascending vs descending) order. (Any sorting algorithm at your choice)
  - Write your favorite recipe with measures in the metric system and in the anglosaxon one

- (optional) Build a simple exercise using generics