Template Method Pattern

Template Method Pattern

- Purpose
  - Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.
- Use When
  - A single abstract implementation of an algorithm is needed.
  - Common behavior among subclasses should be localized to a common class.
  - Parent classes should be able to uniformly invoke behavior in their subclasses.
  - Most or all subclasses need to implement the behavior.

Class Coffee

```
public class Coffee {
    void prepareRecipe() {
         boilWater();
         brewCoffeeGrinds();
         pourInCup();
         addSugarAndMilk();
    }
    public void boilWater() {
         Sysetm.out.println("Boiling water");
    }
    public void brewCoffeeGrinds() {
         Sysetm.out.println("Dripping Coffee through filter");
    }
    public void pourInCup() {
         System.out.println("Pouring into cup");
    }
    public void addSugarAndMilk() {
         System.out.println("Adding Sugar and Milk");
    }
}
```

Class Tea

```
public class Tea {
    void prepareRecipe() {
          boilWater();
          steepTeaBag();
          pourInCup();
          addLemon();
    }
    public void boilWater() {
        Sysetm.out.println("Boiling water");
    }
    public void steepTeaBag() {
          Sysetm.out.println("Steeping the tea");
    }
    public void pourInCup() {
          System.out.println("Pouring into cup");
    }
    public void addLemon() {
          System.out.println("Adding Lemon");
    }
}
```

Problems with the original design

- Code is duplicated across the classes – code changes would have to be made in more than one place.
- Adding a new beverage would result in further duplication.
- Knowledge of the algorithm and implementation is distributed over classes.

Abstracting prepareRecipe()

```
public abstract class CaffeineBeverage {
   final void prepareRecipe() {
     boilWater();
     brew();
     pourInCup();
     addCondiments();
   }
   abstract void brew();
   abstract void addCondiments();

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

```
          Coffee                                    Tea
void prepareRecipe() {               void prepareRecipe() {
   boilWater();                         boilWater();
   brewCoffeeGrinds();       <---->     steepTeaBag();
   pourInCup();                         pourInCup();
   addSugarAndMilk();        <---->     addLemon();
}                                    }
```

Rewriting Coffee and Tea

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

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

More General Approach

- Changes
    - Both subclasses inherit a **general algorithm**.
    - Some methods in the algorithm are **concrete**, i.e. methods that perform the **same actions for all subclasses**.
    - Other methods in the algorithm are **abstract**, i.e. methods that perform **class-specific actions**.
- Advantages
    - A **single class protects and controls the algorithm**, namely, CaffeineBeverage.
    - The **superclass facilitates reuse** of methods.
    - **Code changes** will occur in **only one place**.
    - Other beverages can be **easily added**.

Template Method

```
public abstract class CaffeineBeverage {
    void final prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    abstract void brew();
    abstract void addCondiments();

    void boilWater() {
         // implementation
    }

    void pourInCup () {
         // implementation
    }

}
```

Template Method

Template Method Pattern

- *prepareRecipe*() implements the **template method pattern**.
  - serves as a template for an algorithm, namely that for making a caffeinated beverage.
  - In the template, each step is represented by a method.
  - Some methods are implemented in the superclass.
  - Other method must be implemented by the subclass and are declared abstract.
- The template pattern defines the steps of an algorithm and allows the subclasses to implement one or more of the steps.

Template Method Pattern

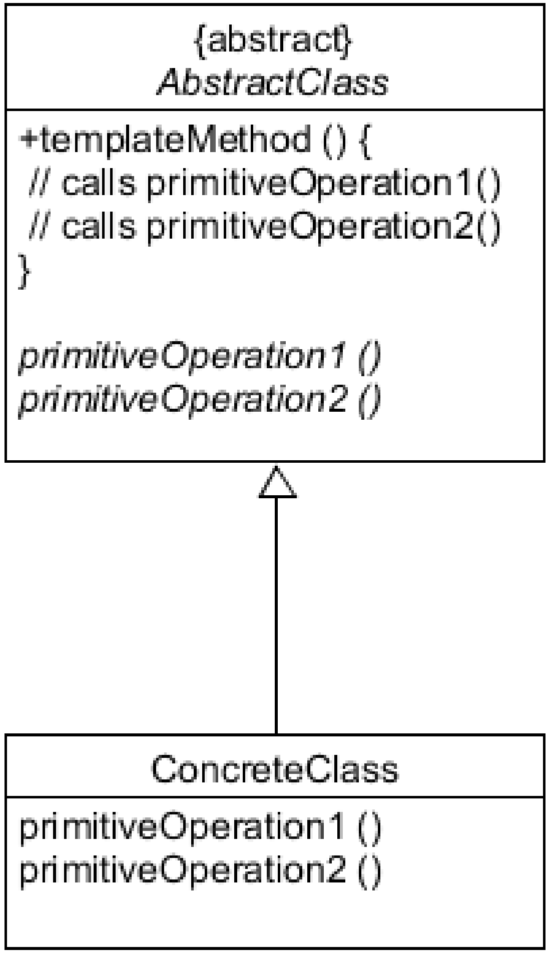


Figure from [HF]

Template Method Pattern

- **Encapsulates** an algorithm by **creating a template** for it.
- Defines the **skeleton of an algorithm** as a **set of steps**.
- Some methods of the algorithm **have to be implemented by the subclasses** – these are abstract methods in the super class.
- The **subclasses** can **redefine certain steps** of the algorithm **without changing the algorithm's structure**.
- Some steps of the algorithm are **concrete methods** defined in the super class.

Hook Method

- A hook is a method that is declared in the abstract class, but only given an empty or default implementation.
  - Gives the subclasses the ability to "*hook into*" the algorithm at various points, if they wish; they can ignore the hook as well.

```
public abstract class CaffeineBeverage {
    final void prepareRecipe() {
          boilWater();
          brew();
          pourInCup();
          if (customerWantsCondiments()) addCondiments();
    }
    abstract void brew();
    abstract void addCondiments();
    public void boilWater() {
           Sysetm.out.println("Boiling water");
    }
    public void pourInCup() {
           System.out.println("Pouring into cup");
    }
    boolean customerWantsCondiments() {
           return true;
    }
}
```

Using the hook in the derived class

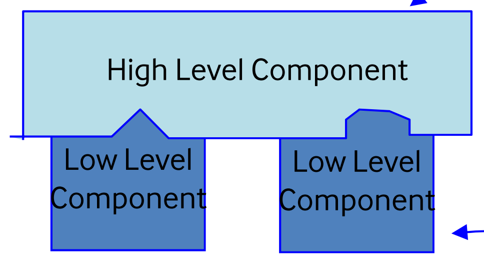
```
public class Coffee extends CaffeineBeverage {
    public void brew() {
         Sysetm.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
         System.out.println("Adding Sugar and Milk");
    }
    public boolean customerWantsCondiments() {
         String answer = getUserInput();
         if (answer.toLowerCase().startsWith("y"_))
             return true;
         else
             return false;
    }
}
```

Examples of Hooks in the Java API

- JFrame hooks
  - paint()
- Applet hooks
  - init()
  - repaint()
  - start()
  - stop()
  - destroy()
  - paint()

Design Principle: Hollywood Principle

- The Hollywood Principle: Don't call us, we'll call you!
  - It prevents "Dependency rot"
  - Dependency rot: high-level components depend on low-level components, and vice versa.
- With the Hollywood principle
  - We allow low level components to hook themselves into a system
  - But high level components determine when they are needed and how.
  - High level components give the low-level components a "don't call us, we'll call you" treatment.



But high-level components control when and how.

Low-level components can participate in the computation.

A low-level component never calls a high-level component directly.

Figure from [HF]

Related Patterns

- Template Method uses inheritance to vary part of an algorithm.
- Strategy uses delegation to vary the entire algorithm.

- Factory Method is a specialization of Template Method

Summary

- Hollywood Principle
  - Don’t call us, we’ll call you
- Template Method Pattern
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
  - Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.