Template Method Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outline

After today's lecture you will be able to:

- Understand the use of the Template Design Pattern
- Use and implement the Template Pattern
- Describe the dangers of Code Duplication
- Describe how hook methods works
- Describe and use the Hollywood Principle

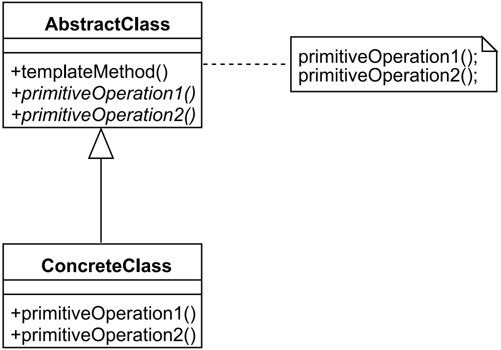
Inspiration

“… the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.” – Edsger W. Dijkstra

Template Method: Definition

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses **redefine** certain steps of an algorithm without changing the algorithm's **structure**
- Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps
  - Makes the algorithm abstract: Each step of the algorithm is represented by a method
  - Encapsulates the details of most steps:
    - Steps (methods) handled by subclasses are declared abstract
    - Shared steps (concrete methods) are placed in the same class that has the template method, allowing for code re-use among the various subclasses.

Template Method: Structure



Very simple pattern...
...but also very powerful
Used typically in application frameworks, e.g. Cocoa and .NET
`primitiveOperation1()` and `primitiveOperation2()` are sometimes referred to as **hook methods** as they allow subclasses to hook their behavior into the service provided by AbstractClass

Example: Tea and Coffee

- We will use the Starbuzz example. This example shows the training guide for baristas and, in particular, the recipes for making coffee and tea.

- Coffee
  - Boil water
  - Brew coffee in boiling water
  - Pour coffee into cup
  - Add sugar and milk

- Tea
  - Boil water
  - Steep tea in boiling water
  - Pour tea in cup
  - Add lemon

Coffee Implementation

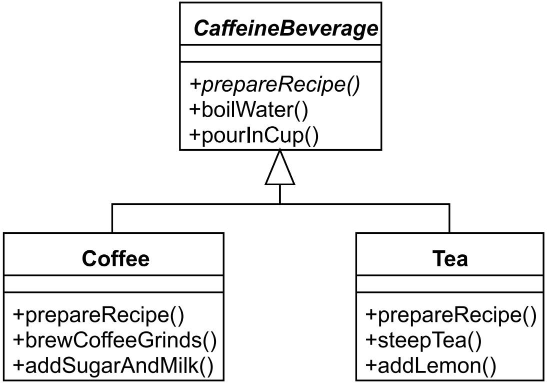
```
public class Coffee {

    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 into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Tea Implementation

```java
public class Tea {

    void perpareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

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

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

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

Code Duplication!

- We have code duplication occurring in these two classes
  - `boilWater()` and `pourInCup()` are exactly the same
- Lets get rid of the duplication



Similar Algorithms

- The structure of the algorithms in `prepareRecipe()` is similar for `Tea` and `Coffee`
  - We can improve our code further by making the code in `prepareRecipe()` more abstract
    - `brewCoffeeGrinds()` and `SteepTea()` -> `brew()`
    - `addSugarAndMilk()` and `addLemon()` -> `addCondiments()`
- Excellent, now all we need to do is specify this structure in `CaffeineBeverage.prepareRecipe()` and make it such that subclasses can't change the structure
- How do we do that?
- Answer: By convention OR by using the keyword "final" in languages that support it

CaffeineBeverage Implementation

```
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.out.println("Pouring into cup");
    }
}
```

- Note: use of final keyword for `prepareRecipe()`
- `brew()` and `addCondiments()` are abstract and must be supplied by subclasses
- `boilWater()` and `pourInCup()` are specified and shared across all subclasses

Coffee and Tea Implementations

```
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");
    }
}

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

Nice and Simple!

What Have We Done?

- Took two separate classes with separate but similar algorithms
- Noticed duplication and eliminated it by introducing a superclass
- Made steps of algorithm more abstract and specified its structure in the superclass
  - Thereby eliminating another “implicit” duplication between the two classes
- Revised subclasses to implement the abstract (unspecified) portions of the algorithm… in a way that made sense for them.

Template Method (TM) vs. No TM

- **No Template Method**
- Coffee and Tea each have own copy of algorithm
- Code is duplicated across both classes
- A change in the algorithm would result in a change in both classes
- Not easy to add new caffeine `CaffeineBeverage`
- Knowledge of algorithm distributed over multiple classes

- **Template Method**
- `CaffeineBeverage` has the algorithm and protects it
- `CaffeineBeverage` shares common code with all subclasses
- A change in the algorithm likely impacts only `CaffeineBeverage`
- New caffeine beverages can easily be plugged in
- `CaffeineBeverage` centralizes knowledge of the algorithm; subclasses plug-in missing pieces

Hook Methods

- Previously I called the abstract methods that appear in a template method “hook” methods
  - Hook methods is an overloaded term. Specifically, hook methods may also refer to a concrete method that appears in the `AbstractClass` that has an empty method body (or a mostly empty method body, see example on next slide), i.e.:

```
public void hook() {}
```

  - `Subclasses` are free to override them but don’t have to since they provide a method body, albeit an empty one
    - In contrast, a subclass is forced to implement abstract methods that appear in `AbstractClass`
- Hook methods, thus, should represent optional parts of the algorithm

Adding a Hook to CaffeineBeverage

```
public abstract class CaffeineBeverageWithHook {

    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

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

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

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

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

prepareRecipe() altered to have a hook method:

- `customerWantsCondiments()`

This method provides a mostly empty method body that subclasses can override

To make the distinction between hook and non-hook methods more clear, you can add the “final” keyword to all concrete methods that you don’t want subclasses to touch

Adding a hook to coffee

```java
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();
        return answer.toLowerCase().startsWith("y");
    }

    private String getUserInput() {
        String answer = null;
        System.out.print("Would you like mil and sugar with your cofee (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 == null ? "no" : answer;
    }
}
```

New Design Principle: Hollywood Principle

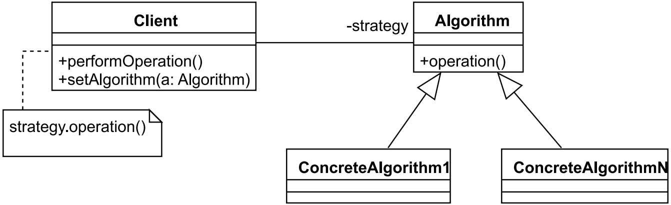
- Don’t call us, we’ll call you
- Or, in OO terms, high-level components call low-level components, not the other way around
  - In the context of the template method pattern, the template method lives in a high-level class and invokes methods that live in its subclasses
- this principle is similar to the dependency inversion principle we previously discussed.
  - Template method encourages clients to interact with the abstract class that defines template methods as much as possible; this discourages the client form depending on the template method subclasses.

Template Methods in the Wild

- Template Method is used a lot since it's a great design tool for creating frameworks
  - the framework specifies how something should be done with a template method
  - that method invokes abstract and hook methods that allow client-specific subclasses to "hook into" the framework and take advantage of/influence its services
- Examples in the Java API
  - Sorting using `compareTo()` method
  - Frames in Swing
  - Applets
- Demonstration

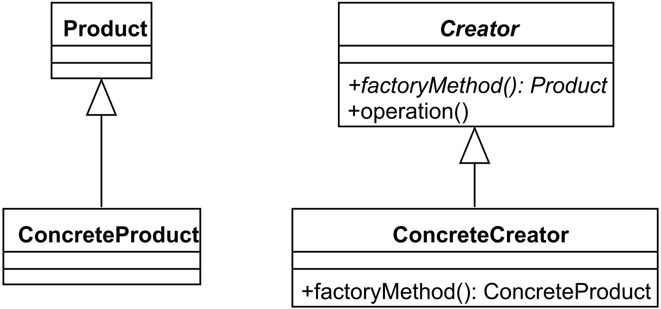
Template Method vs. Strategy

- Both Template Method and Strategy deal with the encapsulation of algorithms
  - Template Method focuses encapsulation on the steps of the algorithm
  - Strategy focuses on encapsulating entire algorithms
  - You can use both patterns at the same time if you want
- Strategy Structure



Template Method vs. Strategy

- Template Method encapsulates the details of algorithms using inheritance
  - Factory Method can now be seen as a specialization of the Template Method pattern


Product
ConcreteProduct
Creator
+factoryMethod(): Product
+operation()
ConcreteCreator
+factoryMethod(): ConcreteProduct

- In contrast, Strategy does a similar thing but uses composition/delegation

Template Method vs. Strategy

- Because it uses inheritance, Template Method offers code reuse benefits not typically seen with the Strategy pattern
- On the other hand, Strategy provides run-time flexibility because of its use of composition/delegation
  - You can switch to an entirely different algorithm when using Strategy, something that you can't do when using Template Method

Are there any questions?