Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Template Method: Intent

- 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.

Template Method: Applicability

- The Template Method pattern should be used
  - to implement the invariant parts of an algorithm once and leave it upto subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
  - to control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

Template Method

- The template method pattern defines an algorithm's steps generally, deferring the implementation of some steps to subclasses.
- The template method is best used when you can generalize between two classes into a new superclass.
- Think of it like another technique to use when you notice you have two separate classes with very similar functionality and order of operations.
- You can choose to use a template method, so that changes to these algorithms only need to be applied in one place instead of two.

Template Method (contd.)

- The template method would be within the superclass, and would therefore be inherited by the subclasses.
- Differences in algorithms would be done through calls to abstract methods whose implementations are provided by the subclass.
- After using generalization, objects can be more effectively reused.
- Inheritance allows functionality to be shared between classes, and enables clearer and more self-explanatory code.

Template Method: Example

- Let's check it out with an example: Assume you have a coffee shop where you serve both coffee and tea. You have a training manual for preparing coffee and tea like below:

- Coffee Recipe:
  1. Boil some water
  2. Brew coffee in boiling water
  3. Pour coffee in cup
  4. Add sugar and milk

- Tea Recipe:
  1. Boil some water
  2. Steep tea in boiling water
  3. Pour tea in cup
  4. Add lemon

Template Method: Example (contd.)

```
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");
    }
}
```

Template Method: Example (contd.)

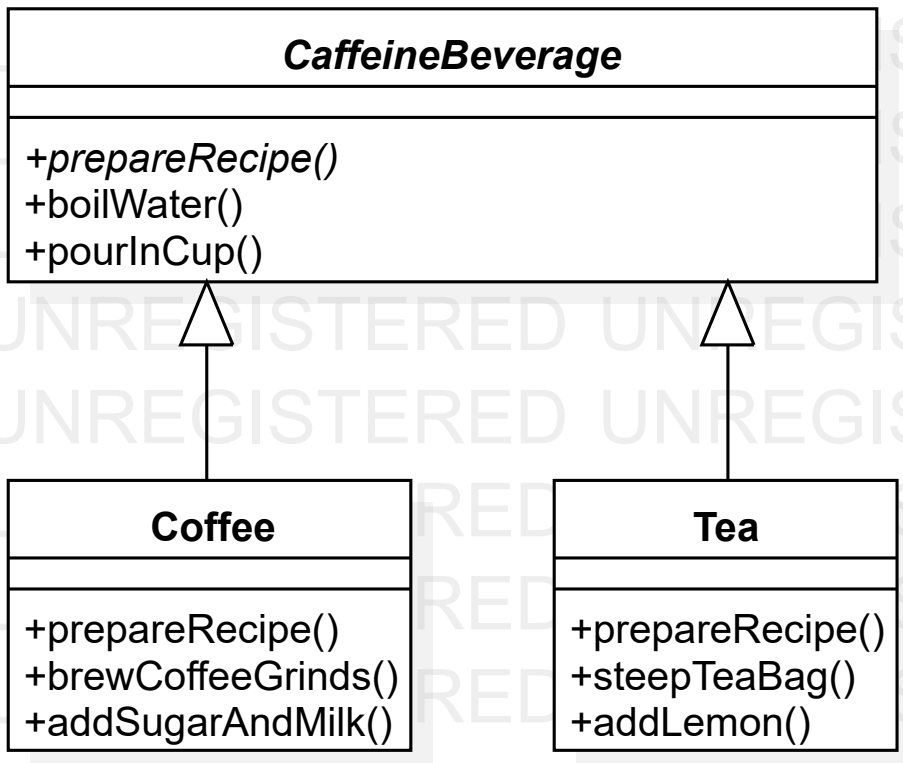
```
public class Tea {
    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }
    public void boilWater() {
        System.out.println("Boiling water");
    }
    public void steepTeaBag() {
        System.out.println("Steeping the tea");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

Template Method: Example (contd.)

- You've seen that the `Coffee` and `Tea` classes have a fair bit of code duplication.

- Take another look at the `Coffee` and `Tea` classes and draw a class diagram showing how you'd redesign the classes to remove redundancy

Template Method: Example (contd.)



Template Method: Example (contd.)

- Did we do a good job on the redesign?
- Hmmmm, take another look. Are we overlooking some other commonality?
- What are other ways that `Coffee` and `Tea` are similar?
- So what else do `Coffee` and `Tea` have in common?

Template Method: Example (contd.)

- See both the recipes again, they follow the same algorithm:
  - Step-1 and step-4 are already abstracted into the base class
  - Step-2 and step-3 are not abstracted, but are the same, they just apply to different beverages.
- So, can we find a way to abstract `prepareRecipe()` too? Yes, let's find out...

Template Method: Example (contd.)

- The first problem we have is that Coffee uses `brewCoffeeGrinds()` and `addSugarAndMilk()` methods while Tea uses `steepTeaBag()` and `addLemon()` methods.
- Let's think through this: steeping and brewing aren't so different; they're pretty analogous. So let's make a new method name, say, `brew()`, and we'll use the same name whether we're brewing coffee or steeping tea.
- Likewise, adding sugar and milk is pretty much the same as adding a lemon: both are adding condiments to the beverage. Let's also make up a new method name, `addCondiments()`, to handle this.
- So, our new `prepareRecipe()` method will look like this:

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

Template Method: Example (contd.)

- Now we have a new `prepareRecipe()` method, but we need to fit it into the code. To do this we are going to start with the `CaffeineBeverage` superclass:

```
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");
    }
}
```

Template Method: Example (contd.)

- Finally we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments:

```
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");
    }
}
```

Meet the Template Method

- We've basically just implemented the Template Method Pattern. What's that? Let's look at the structure of the `CaffeineBeverage` class; it contains the actual "template method:"

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

Let's make a cup of tea...

Template Method: Known Uses

- Template methods are so fundamental that they can be found in almost every abstract class.

Template Method: Related Patterns

- Factory methods are often called by template methods.
- Strategy: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.