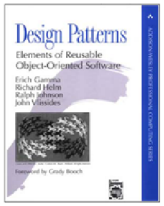


Design Pattern Definitions from the GoF Book

The Template Method Pattern

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

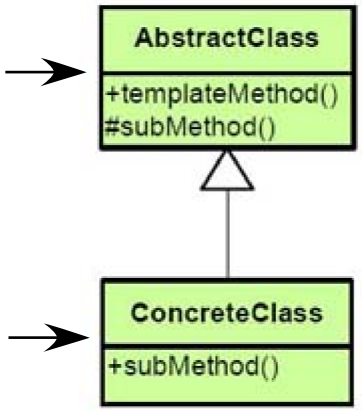
- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Template Method
Quick Overview

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.

The class that defines the interface for the algorithm possibly along with the common functionality.

The class that defines the specific steps for the algorithm inheriting the common functionality.



Design Patterns: The Template Method

Starbuzz Coffee Recipe

```
class Coffee
{
    public:
        void prepareRecipe()
        {
            boilWater();
            brewCoffeeGrinds();
            pourInCup();
            addSugarAndMilk();
        }
        void boilWater();
        void brewCoffeeGrinds();
        void pourInCup();
        void addSugarAndMilk();
}
```

Starbuzz Tea Recipe

```
class Tea
{
    public:
        void prepareRecipe()
        {
            boilWater();
            steepTeaBag();
            pourInCup();
            addSugarAndLemon();
        }
        void boilWater();
        void steepTeaBag();
        void pourInCup();
        void addSugarAndLemon();
}
```

Do you notice the similarities in these two Beverage “Algorithms”?

Design Patterns: The Template Method

Starbuzz Beverage Recipe

```
class CaffeineBeverage
{
    public:
        void prepareRecipe()
        {
            boilWater();
            brew();
            pourInCup();
            addCondiments();
        }
        void boilWater();
        virtual void brew();
        void pourInCup();
        virtual void addCondiments();
}
```

Starbuzz Coffee Recipe

```
class Coffee:CaffeineBeverage
{
    public:
        void brew()
        {
            cout << "Dripping coffee through filter";
        }
        void addCondiments();
        {
            cout << "Adding sugar and milk";
        }
}
```

Starbuzz tea Recipe

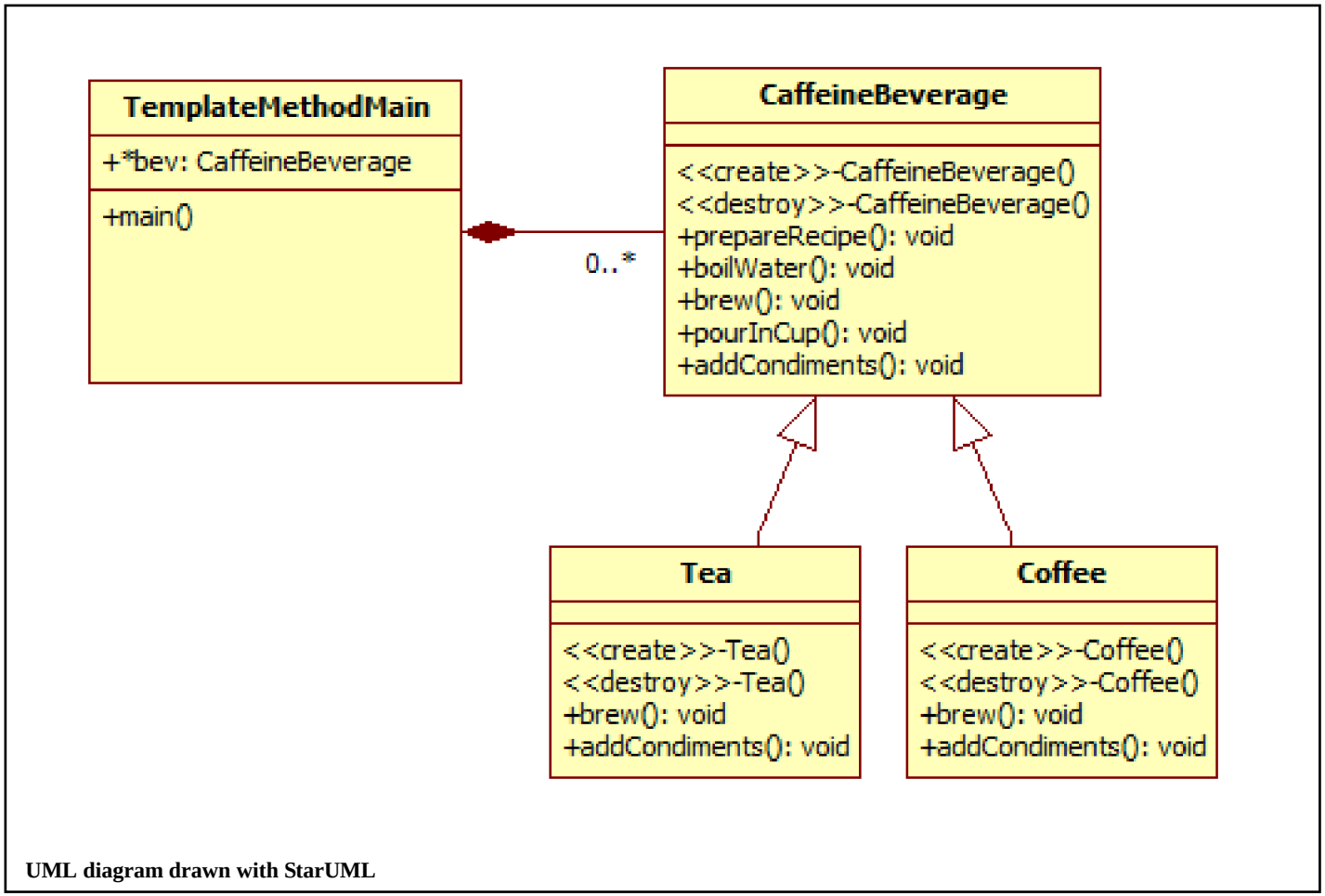
```
class Tea:CaffeineBeverage
{
    public:
        void brew()
        {
            cout << "Steeping the tea";
        }
        void addCondiments();
        {
            cout << "Adding sugar and lemon";
        }
}
```

Now we have an Algorithm that is the same for both beveages.

But we have deferred certain steps of the algorithm to sub-classes.

Design Patterns: The Template Method

Code Sample



UML diagram drawn with StarUML

TemplateMethodMain

Creates an instance of Tea and Coffee using a pointer to a CaffeineBeverage
Calls each of the preparation algorithm functions in each.
Parent class handles all common steps.
Sub-classes handle all specific steps.

Let's look at the code and run the demonstration.