Template Method

★ Structural Patterns
  » strategy
  » adapter
  » façade
★ Behavioral Patterns
  » observer
  » decorator
  » command
  » template method
★ Creational Patterns
  » factory method
  » abstract factory
  » singleton
Existing Problem with Code

Duplicated code is difficult to change, maintain, or extend
Example

```java
Class A
void do() {
    do_method_A();
    do_method_B();
    do_method_C();
    do_method_D();
}
void method_A() {
}
void method_B() {
}
void method_C() {
}
void method_D() {
}
```

```java
Class B
void do() {
    do_method_A();
    do_method_1();
    do_method_C();
    do_method_2();
}
void method_A() {
}
void method_1() {
}
void method_C() {
}
void method_2() {
}
```
Remove Redundancy

**Superclass**
- `abstract do()`
- `method_A()`
- `method_C()`

**Class A**
```java
do() {
    A();
    B();
    C();
    D();
}
method_B()
method_DC()
```

**Class B**
```java
do() {
    A();
    1();
    C();
    2();
}
method_1()
method_2()
```
Remove Redundancy

Superclass:
- abstract do()
- method_A()
- method_C()

Class A:
- do() {
  A();
  B();
  C();
  D();
}
- method_B()
- method_D()

Class B:
- do() {
  A();
  1();
  C();
  2();
}
- method_1()
- method_2()

Abstract methods are marked with "abstract method" and common methods are marked with "common".
Remove Redundancy

Class A

```java
do() {
    A();
    B();
    C();
    D();
}
method_B() {
    op1();
    op2();
    op3();
}
```

Class B

```java
do() {
    A();
    1();
    C();
    2();
}
method_1() {
    op1();
    opA();
    op3();
}
```

(code duplication)

(same)
Define Template Method

Abstract Class

```java
final do() {
    A();
    1B();
    C();
    2D();
}
abstract method_1B()
abstract method_2D()
method_AC()
method_C()
```

Class A
- method_1B()
- method_2D()

Class B
- method_1B()
- method_2D()

This is called a template method and its final to avoid subclasses overriding it.

Must be implemented by concrete subclasses.

Primitive method

Concrete method

Override
Template Method Definition

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses
- Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- Note the method itself is typically declared final so that it cannot be modified by subclasses
- This pattern may be the most common pattern used!
abstract class AbstractClass {
    /* A template method : */
    final void TemplateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
        doSomething();
    }
}
Add a Hook (from HFDP)

```java
abstract class AbstractClass {
    /* A template method : */
    final void TemplateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
        doSomething();
    }
    void hook() {
    }
}
```
Hooks

- Concept is used often in APIs – Java API paint method for drawing is a hook
- Applets have hooks (repaint, start, stop, destroy, etc.)
- Junit TestCase (setUp, tearDown)
- Hook is an empty method or provides some default behavior in super class. Designed to be overridden if deemed necessary in subclasses.
Design Principle: Depend on Abstraction

Abstract class

controls algorithm

final do()
abstract method_1B()
abstract method_2D()
method_A()
method_C()

clients depend on abstraction not on class A or B

subclasses don’t call abstract without being called first

Class A

method_1B()
method_2D()

just implementation

Class B

method_1B()
method_2D()
OO Design Principle

The Hollywood Principle: Don't call us, we'll call you

- A low-level component never calls a high-level component directly
- Low-level components can participate in a computation, but the high-level components control when and how
- Low-level components are sub-classes that provide implementations for abstract behaviors defined in super-class (high level)
Example from Wikipedia

/**
 * An abstract class that is
 * common to several games in
 * which players play against
 * the others, but only one is
 * playing at a given time.
 */

abstract class Game {

    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    /* A template method : */
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
Example from Wikipedia

//Now we can extend this class in order
//to implement actual games:

class Monopoly extends Game {

    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        // Initialize money
    }
    void makePlay(int player) {
        // Process one turn of player
    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    void printWinner() {
        // Display who won
    }

    /* Specific declarations for the Monopoly game. */

    // ...}
Example from Wikipedia

class Chess extends Game {

    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        // Put the pieces on the board
    }

    void makePlay(int player) {
        // Process a turn for the player
    }

    boolean endOfGame() {
        // Return true if in Checkmate or
        // Stalemate has been reached
    }

    void printWinner() {
        // Display the winning player
    }

    /* Specific declarations for the chess game. */

    // ...
}
Template Method Usage

- Let subclasses implement (through method overriding) behavior that can vary
- Avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in each of the subclasses.
- Control at what point(s) subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.
- There are many real world examples of this pattern in action, but not all follow the pattern 'by the book'
Template and Strategy

• Both encapsulate algorithms
• Template Method accomplishes its work via inheritance
• Strategy uses composition
• Which provides more flexibility?
• Side note: the Factory Method we know is a specialization of the Template Method :-)
