Introduction to Design Pattern

Lutfun Nahar Lota

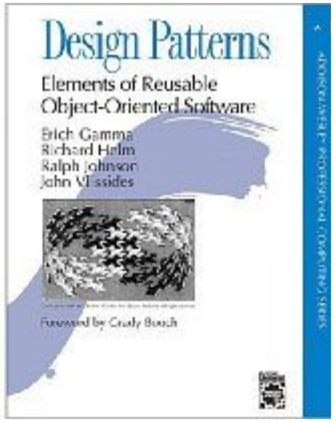
Lecturer, CSE

The Beginning of Patterns

"Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

“Gang of Four” (GoF) Book

- Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994
- Written by this "gang of four"
  - Dr. Erich Gamma, then Software Engineer, Taligent, Inc.
  - Dr. Richard Helm, then Senior Technology Consultant, DMR Group
  - Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department
  - Dr. John Vlissides, then a researcher at IBM
    - Thomas J. Watson Research Center
    - See John's WikiWiki tribute page http://c2.com/cgi/wiki?JohnVlissides



Other Discovered Patterns

The book **Data Access Patterns** by Clifton Nock introduces 4 decoupling patterns, 5 resource patterns, 5 I/O patterns, 7 cache patterns, and 4 concurrency patterns.

Types of Design Patterns

- Three categories of Design Pattern
  - ***Creational patterns*** deal with the process of object creation and class instantiation
  - ***Structural patterns***, deal with structure of classes and objects to form larger structures and provide new functionality.
  - ***Behavioral patterns***, which deal with dynamic interaction among classes and objects

GoF Patterns

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

Why Study Patterns?

- Reusable in multiple projects.
- Provide the solutions that help to define the system architecture.
- Capture the software engineering experiences.
- Provide transparency to the design of an application.
- Well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
- Design patterns don't guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

Other advantages

- Most design patterns make software more modifiable, less brittle
  - we are using time tested solutions
- Using design patterns makes software systems easier to change—more maintainable
- Helps increase the understanding of basic object-oriented design principles
  - encapsulation, inheritance, interfaces, polymorphism

Style for Describing Patterns

- We will use this structure:
  - *Pattern name*
  - *Recurring problem:* what problem the pattern addresses
  - *Solution:* the general approach of the pattern
  - *UML for the pattern*
    - *Participants:* a description as a class diagram
  - *Use Example(s):* examples of this pattern, in Java

Template Pattern

Football
initialize(){} StartPlay() {} endPlay(){}

Cricket
initialize(){} StartPlay() {} endPlay(){}

Template Pattern

```
Football

initialize(){}

StartPlay()  {}

endPlay(){}
```

```
Cricket

initialize(){}

StartPlay()  {}

endPlay(){}
```

Derive the
Parent Class

```
Game

initialize(){}

StartPlay()  {}

endPlay(){}
```

Template Pattern


```
Football
---
initialize(){}

StartPlay()  {}

endPlay(){}
```

```
Cricket
---
initialize(){}

StartPlay()  {}

endPlay(){}
```

Make it Abstract

The method should be declared 'final'

```
Game
---
initialize();

StartPlay()  ;

endPlay();
```

Template Pattern

```
Football f = new Football();
 f.initialize(){}
 f.StartPlay() {}
 f.endPlay()   {}
```

```
Cricket c = new Cricket();
 c.initialize(){}
 c.StartPlay() {}
 c.endPlay()   {}
```

Template Pattern

```
Football f = new Football();
```

```
Play(){

initialize();
StartPlay();
endPlay() ;

}
initialize(){}
StartPlay(){}
endPlay() {}
```

```
Football f = new Football();
f.play();
```

```
Cricket c = new Cricket();
```

```
Play(){

initialize();
StartPlay();
endPlay()  ;

}
initialize(){}
StartPlay(){}
endPlay() {}
```

```
Cricket c = new Cricket();
c.play();
```

Template Pattern

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template Pattern

Problem

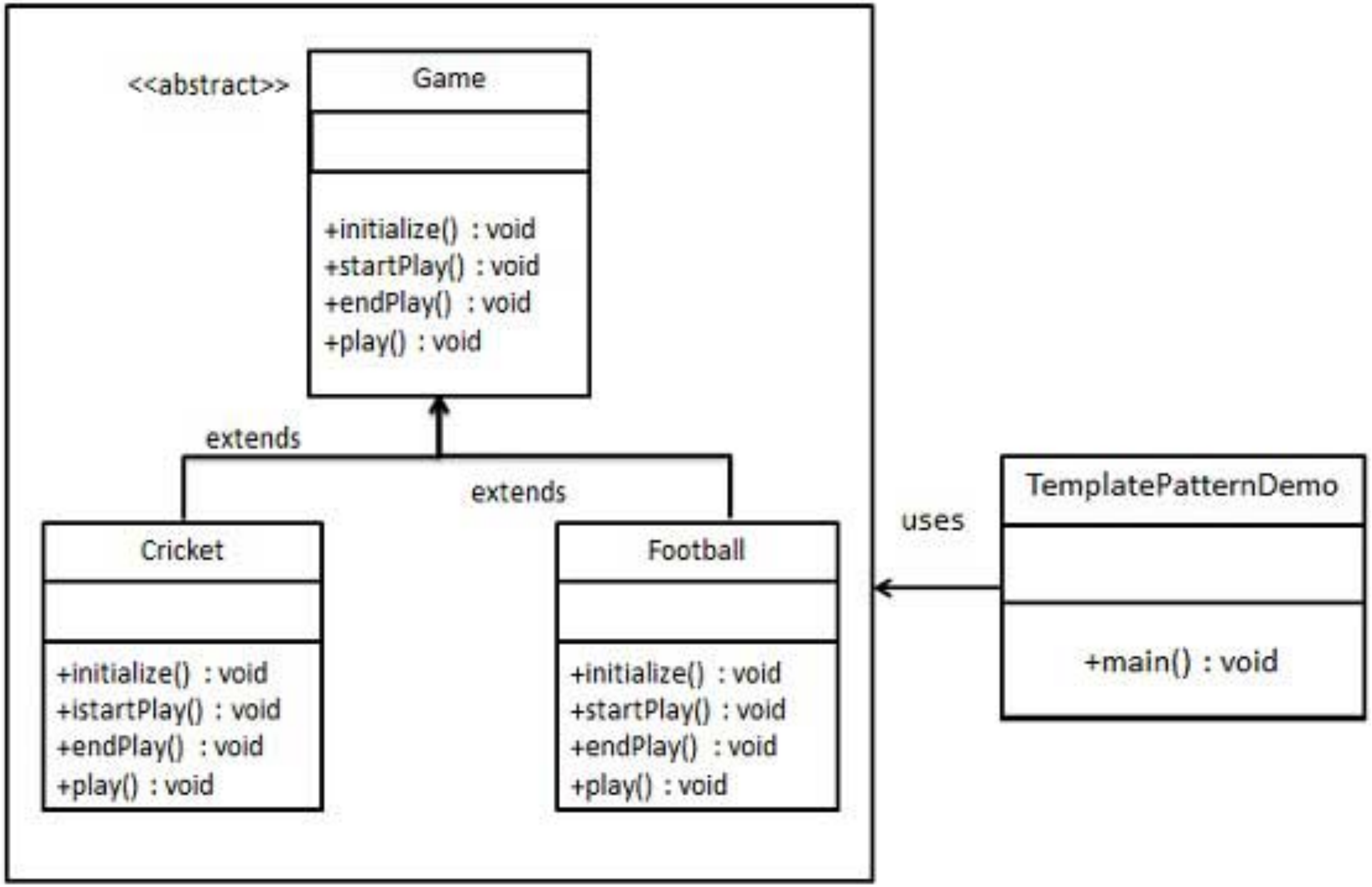
Two different components have significant similarities, but demonstrate no reuse of common interface or implementation.

Template Pattern

Solution

The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single *template method.*

Class Diagram for Template Pattern



Strategy Pattern

- a class behavior or its algorithm can be changed at run time.

- *Enforces: Open Closed Principle*

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

Strategy Pattern

Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

Strategy Pattern

Problem

encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: **no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.**

Strategy Pattern
Problem

```
Class Arithmetic OpAddSub{
      If (add){
       res = a+b;
      Return res;
      }
      If (sub){
       res = a-b;
      Return res;
      }
}
```

Strategy Pattern
Problem

```
Class Arithmetic OpAddSubMul{
        If (add){
         res = a+b;
        }
        If (sub){
         res = a-b;
        }
        If (mul){
         res = a*b;
        }

}
```

Strategy Pattern

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Strategy Pattern

```
OperationAdd aaaa = new OperationAdd()
aaaa.doOperation(10,5);

OperationSub bbbb = new OperationSub()
bbbb.doOperation(10,5);
```

Strategy Pattern

Create *Context* Class.

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Strategy Pattern

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

Strategy Pattern UML Diagram