Design Patterns

Definition and History:

The use of patterns is essentially the reuse of well-established good ideas. A "pattern" is a named, well-understood good solution to a common problem in context.

The idea comes originally from the architect Christopher Alexander (70s), who described and named common architectural problems and discussed solutions to them.

His actual definition was:

- Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
- As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
- As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

Christopher Alexander says, "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".

Four essential elements of a pattern::

1. The pattern **name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. The **problem** describes when to apply the pattern.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations
4. The **consequences** are the results and trade-offs of applying the pattern.

The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

- A design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.
- The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities.
- Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not it can be applied in view of other design constraints, and the consequences and trade-offs of its use.
- Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation.
- Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages.
- While their description is slanted toward OO design, it could be easily generalized.

Their book gives common software design patterns found in industry. They group the designs into three categories:

1. **creational design patterns** (techniques to instantiate groups of objects)
2. **structural design patterns** (organize classes and objects into larger structures)
3. **behavioral design patterns** (assign responsibilities to objects, how objects collaborate)

In software design, there are many commonly arising technical problems. Software design patterns are not classes or objects, but sets of classes and objects. Good software designers recognize them and understand how to solve them.

Elements of a Pattern:

GoF says, in general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.

2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well.

Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system"s flexibility, extensibility or portability. Listing these consequences explicitly helps you understand and evaluate them.

Describing Design Patterns

(a consistent format from GoF):

Name and Classification -- Must have a meaningful name which conveys the essence of the pattern succinctly. Classification categorizes it into either creational, structural, or behavioral design patterns.

Intent -- A statement of the problem which answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue does it address?

Also Known As -- Other well-known names for the pattern, if any.

Motivation -- A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability -- What are the situations in which the design pattern can be applied? What are example of poor design that the pattern can address? How can you recognize these situations?

Structure -- A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT). (UML, based on OMT, is now the standard notation used.)

Participants -- The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations -- How the participants collaborate to carry out their responsibilities.

Consequences -- How does the pattern support its objectives? What are the trade offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation -- What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code and Usage -- Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk (or Java).

Known Uses -- Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns -- What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

About Patterns:

A good pattern does the following:

- It solves a problem: Patterns capture solutions, not just abstract principles or strategies.
- It is a proven concept: Patterns capture solutions with a track

record, not theories or speculation.

- The solution isn't obvious: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles.

The best patterns generate a solution to a problem indirectly -- a necessary approach for the most difficult problems of design.

- It describes a relationship: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- The pattern has a significant human component ... All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object Oriented Software** , there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operators. This gives the program more flexibility in

	deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Singleton pattern

In software engineering, the **singleton pattern** is a design pattern that **restricts the instantiation of a class to one object**. This is useful when exactly one object is needed to coordinate actions across the system.

The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.

There are some who are critical of the singleton pattern and consider it to be an anti-pattern in that it is frequently used in scenarios where it is not beneficial, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application

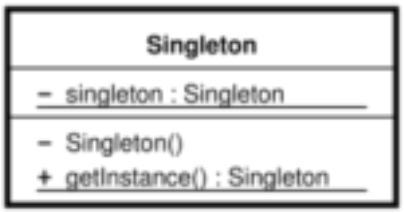
Common uses

· The Abstract Factory, Builder, and Prototype patterns can use Singletons in their implementation.

· Facade objects are often singletons because only one Facade object is required. · State objects are often singletons.

· Singletons are often preferred to global variables because:

- They do not pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables.
- They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.



Implementation of a singleton pattern must satisfy the single instance and global access principles. It requires a mechanism to access the singleton class member without creating a class object and a mechanism to persist the value of class members among class objects. The singleton pattern is implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made private. Note the **distinction** between a simple static instance of a class and a singleton: although a singleton can be implemented as a static instance, it can also be lazily constructed, requiring no memory or resources until needed.

- The singleton pattern must be carefully constructed in multi-threaded applications. If two threads are to execute the creation method at the same time when a singleton does not yet exist, they both must check for an instance of the singleton and then only one should create the new one. If the programming language has concurrent processing capabilities the method should be constructed to execute as a mutually exclusive operation. The classic solution to this problem is to use mutual exclusion on the class that indicates that the object is being instantiated.

```
public final class SingletonDemo {
    private static volatile SingletonDemo instance;
    private SingletonDemo() { }

    public static SingletonDemo getInstance() {
        if (instance == null ) {
            synchronized (SingletonDemo.class) {
                if (instance == null) {
                    instance = new SingletonDemo();
                }
            }
        }

        return instance;
    }
}
```

An alternate simpler and cleaner version may be used at the expense of potentially lower concurrency in a multithreaded environment:

```
public final class SingletonDemo {
    private static SingletonDemo instance = null;
    private SingletonDemo() { }

    public static synchronized SingletonDemo getInstance() {
        if (instance == null) {
            instance = new SingletonDemo();
        }

        return instance;
    }
}
```

Factory Method

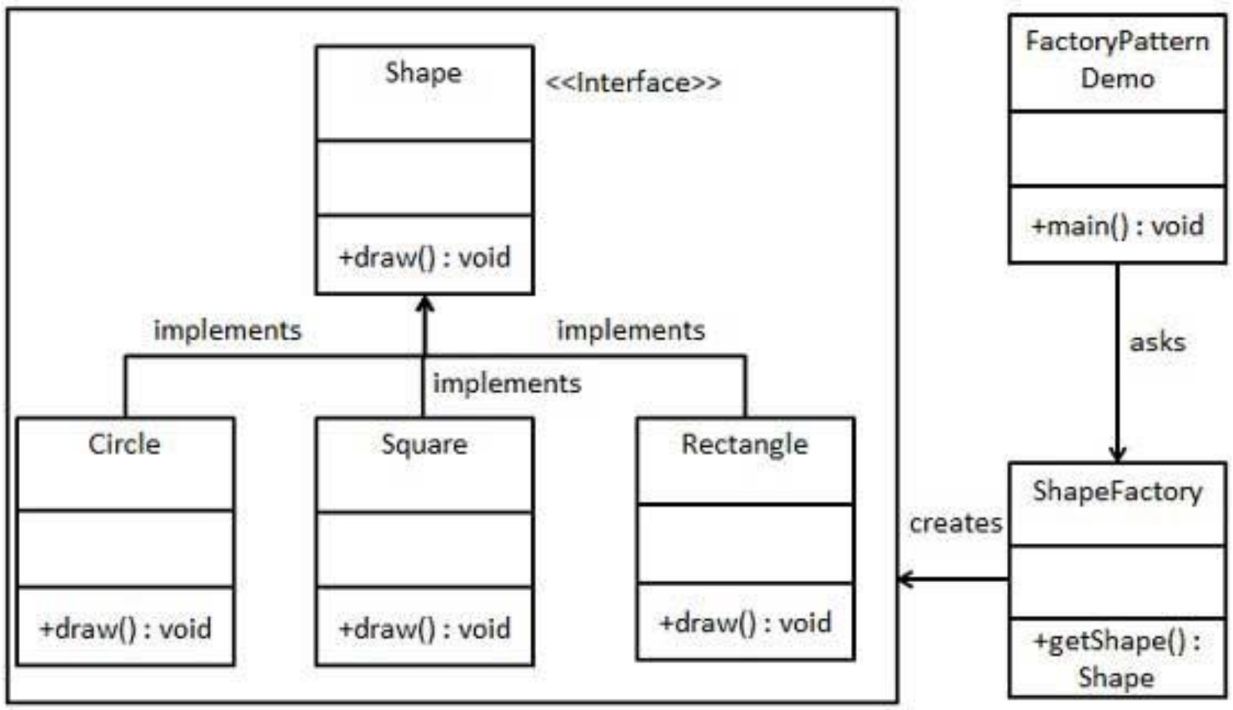
Factory pattern is one of most used design pattern in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



Step 1 Create an interface.

Shape.java

```
public interface Shape {
 void draw();
}
```

Step 2 Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

 @Override
 public void draw() {
 System.out.println("Inside Rectangle::draw()
method.");  }
 }
```

Square.java

```
public class Square implements Shape {

 @Override
 public void draw() {
 System.out.println("Inside Square::draw()
method.");  }
}
```

Circle.java

```
public class Circle implements Shape {

 @Override
 public void draw() {
 System.out.println("Inside Circle::draw() method.");
}
 }
```

Step 3 Create a Factory to generate objects of concrete class based on given information. *ShapeFactory.java*

```
public class ShapeFactory {

 //use getShape method to get object of type
shape   public Shape getShape(String
shapeType){
 if(shapeType == null){
 return null;
 }
 if(shapeType.equalsIgnoreCase("CIRCLE"))
```

```
        {  return new Circle();
 } else if(shapeType.equalsIgnoreCase("RECTANGLE"))
        {  return new Rectangle();
 } else if(shapeType.equalsIgnoreCase("SQUARE"))
        {  return new Square();
}
return null;
} }
```

Step 4 Use the Factory to get an object of concrete class by passing information such as type. *FactoryPatternDemo.java*

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

Step 5 Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Lazy Initialization

In computer programming, **lazy initialization** is the **tactic of delaying the creation of an object**, the calculation of a value, or some other expensive process until the first time it is needed.

This is typically accomplished by augmenting a variable's accessor method (or property definition) to check for a previously-created instance. If none exists a new instance is created, placed into the variable, and this new object is returned to the caller in a just-in-time fashion. In this manner object creation is deferred until first use which can, in some circumstances (e.g., sporadic object access), increase system responsiveness and speed startup by bypassing large scale object pre-allocation. (Note that this may have attendant counter-effects on overall performance, however, as the impact of object instantiation is then amortized across the startup/warm-up phase of the system.) In multithreaded code, access to lazy-initialized objects/state must be synchronized to guard against a race condition.

In a software design pattern view, lazy initialization is often used together with a factory method pattern. This combines three ideas:

· Using a factory method to get instances of a class (factory method pattern) · Store the instances in a map, so you get the *same* instance the next time you ask for an instance with *same* parameter (multiton pattern)

· Using lazy initialization to instantiate the object the first time it is requested (lazy initialization pattern)

Source Code:

```
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public class Program {

 /**
 * @param args
 */
 public static void main(String[] args) {
 Fruit.getFruitByTypeName(FruitType.banana);
 Fruit.showAll();
 Fruit.getFruitByTypeName(FruitType.apple);
 Fruit.showAll();
 Fruit.getFruitByTypeName(FruitType.banana);
 Fruit.showAll();
 }
 }

enum FruitType {

 none,
 apple,
 banana,
 }

class Fruit {

 private static Map<FruitType, Fruit> types = new
HashMap<>();
 /**
 * Using a private constructor to force the use of the factory method.
* @param type
 */
 private Fruit(FruitType type) {
 }

 /**
 * Lazy Factory method, gets the Fruit instance associated with a
certain
 * type. Instantiates new ones as needed.
```

```
 * @param type Any allowed fruit type, e.g. APPLE
 * @return The Fruit instance associated with that
type.  */
 public static Fruit getFruitByTypeName(FruitType
type) {  Fruit fruit;
 // This has concurrency issues. Here the read to types is  not
synchronized,
 // so types.put and types.containsKey might be called at  the same
time.
 // Don't be surprised if the data is corrupted.  if
(!types.containsKey(type)) {
 // Lazy initialisation
 fruit = new Fruit(type);
 types.put(type, fruit);
 } else {
 // OK, it's available currently
 fruit = types.get(type);
 }

 return fruit;
 }

 /**
 * Lazy Factory method, gets the Fruit instance associated with a
certain
 * type. Instantiates new ones as needed. Uses double-checked
locking   * pattern for using in highly concurrent environments.
 * @param type Any allowed fruit type, e.g. APPLE
 * @return The Fruit instance associated with that
type.  */
 public static Fruit
getFruitByTypeNameHighConcurrentVersion(FruitType  type) {
 if (!types.containsKey(type)) {
 synchronized (types) {
 // Check again, after having acquired the lock to make  sure

 // the instance was not created meanwhile by another  thread
 if (!types.containsKey(type)) {
 // Lazy initialisation
                        types.put(type, new Fruit(type));
 }
```

```
}
}

return types.get(type);
}

/**
* Displays all entered fruits.
*/
public static void showAll() {
if (types.size() > 0) {

System.out.println("Number of instances made = " +  types.size());

for (Entry<FruitType, Fruit> entry : types.entrySet()) {
String fruit = entry.getKey().toString();
fruit = Character.toUpperCase(fruit.charAt(0)) +  fruit.substring(1);
System.out.println(fruit);
}

System.out.println();
}
}
}
```

Output

```
Number of instances made = 1
Banana

Number of instances made = 2
Banana
Apple

Number of instances made = 2
Banana
Apple
```

Convention Over Configuration

Convention over configuration (also known as **coding by convention**) is a software design paradigm used by software frameworks that attempt to decrease the number of decisions that a developer using the framework is required to make without necessarily losing flexibility. The concept was introduced by David Heinemeier Hansson to describe the philosophy of the Ruby on Rails web framework, but is related to earlier ideas like the concept of "sensible defaults" and the principle of least astonishment in user interface design.

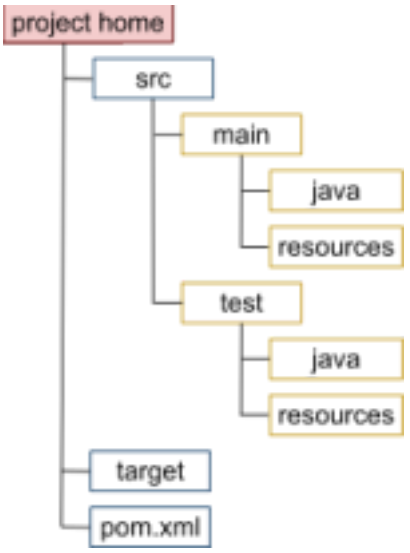
The phrase essentially means a developer only needs to specify unconventional aspects of the application. For example, if there is a class Sales in the model, the corresponding table in the database is called "sales" by default. It is only if one deviates from this convention, such as calling the table "product sales", that one needs to write code regarding these names.

When the convention implemented by the tool matches the desired behavior, it behaves as expected without having to write configuration files. Only when the desired behavior deviates from the implemented convention is explicit configuration required.

Ruby on Rails' use of the phrase is particularly focused on its default project file and directory structure, which prevent developers from having to write XML configuration files to specify which modules the framework should load, which was common in many earlier frameworks.

Disadvantages of the convention over configuration approach can occur due to conflicts with other software design principles, like the Zen of Python's "explicit is better than implicit." A software framework based on convention over configuration often involves a domain-specific language with a limited set of constructs or an inversion of control in which the developer can only affect behavior using a limited set of hooks, both of which can make implementing behaviors not easily expressed by the provided conventions more difficult than when using a software library that does not try to decrease the number of decisions developers have to make or require inversion of control.

Usage



The Maven software tool auto-generated this directory structure for a Java project.

Many modern frameworks use a *convention over configuration* approach.

The concept is older, however, dating back to the concept of a default, and can be spotted more recently in the roots of Java libraries. For example, the JavaBeans specification relies on it heavily. To quote the JavaBeans specification 1.01:

"As a general rule we don't want to invent an enormous java.beans.everything class that people have to inherit from. Instead

we'd like the JavaBeans runtimes to provide default behaviour for 'normal' objects, but to allow objects to override a given piece of default behaviour by inheriting from some specific java.beans.something interface."

Dependency Injection and Inversion of Control:

Dependency Injection:

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. A dependency is an object that can be used. An injection is the passing of dependency to a dependent object that would use it.There are three main common means for a client to accept dependency injection.

1. Setter and Getter based injection
2. Interface based injections
3. Constructor based injections

Inversion Of Control

Inversion of Control (IoC) is a design principle in which custom-written portions of **a computer program receive the flow of control from a generic framework**. It is used to increase modularity of the program and make it extensible. Inversion of control is the principle where the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the external sources (framework, services, other components) take control of it. It's like we plug something into something else.
(http://martinfowler.com/bliki/InversionOfControl.html)

The main goal of Inversion of control and Dependency Injection is to remove dependencies of an application. This makes the system more decoupled and maintainable.

Differences between Dependency Injection and Inversion of Control

- IoC is a generic term and implemented in several ways (events, delegates etc). DI is a subtype of IOC and is implemented by constructor injection, setter injection or method injection. DI is a specialized version of the IoC pattern or DI and Service is the way of implementing IoC.
- Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).
- Dependency Injection (DI) means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.

EXTRAS:

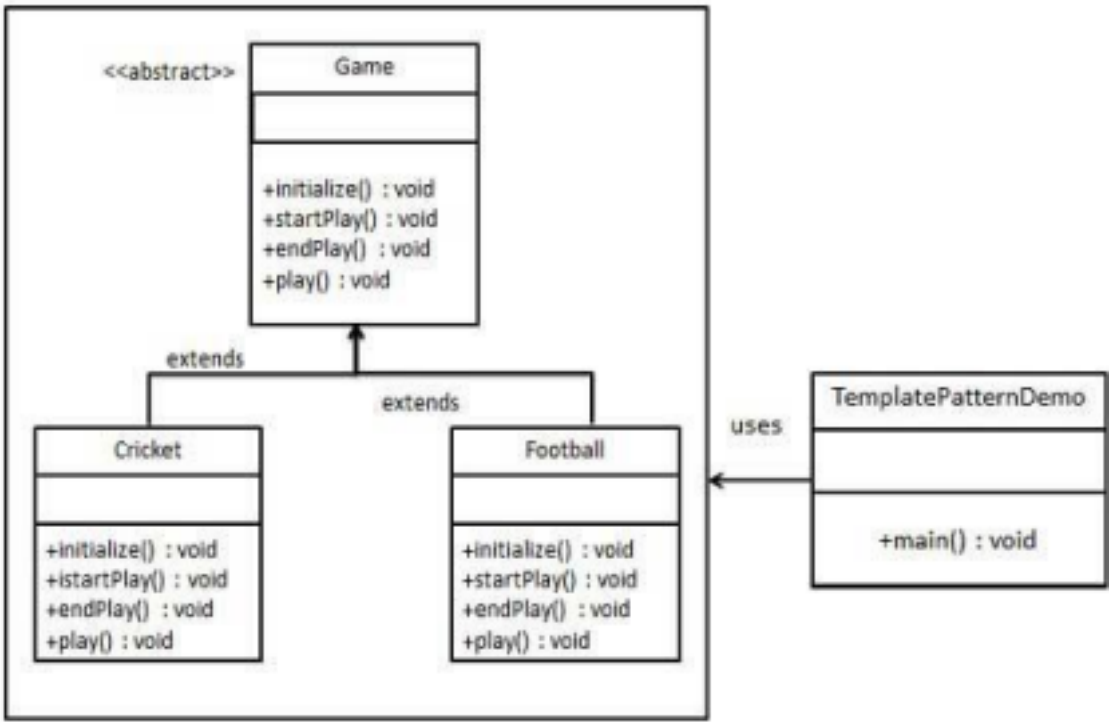
Template pattern

In Template pattern, an abstract class exposes a defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class. This pattern comes under the behavior pattern category.

Implementation

We are going to create a *Game* abstract class defining operations with a template method set to be final so that it cannot be overridden. *Cricket* and *Football* are concrete classes that extend *Game* and override its methods.

TemplatePatternDemo, our demo class, will use *Game* to demonstrate use of template pattern.



Step 1

Create an abstract class with a template method being final.

Game.java

```
public abstract class Game {
 abstract void initialize();
 abstract void startPlay();

 abstract void endPlay();

 //template method
 public final void play(){

 //initialize the game
 initialize();

 //start game
 startPlay();

 //end game
 endPlay();
 }
 }
```

Step 2

Create concrete classes extending the above class.

Cricket.java

```
public class Cricket extends Game {

 @Override
 void endPlay() {
 System.out.println("Cricket Game Finished!");
 }

 @Override
 void initialize() {
  System.out.println("Cricket Game Initialized! Start playing.");
}

 @Override
 void startPlay() {
   System.out.println("Cricket Game Started. Enjoy the
game!");  } }
```

Football.java

```
public class Football extends Game {

 @Override
 void endPlay() {
 System.out.println("Football Game Finished!");
 }

 @Override
 void initialize() {
 System.out.println("Football Game Initialized! Start playing.");
}

 @Override
 void startPlay() {

   System.out.println("Football Game Started. Enjoy the
game!");  }
 }
```

Step 3

Use the *Game*'s template method play() to demonstrate a defined way of playing game. *TemplatePatternDemo.java*

```
public class TemplatePatternDemo {
 public static void main(String[] args) {

 Game game = new Cricket();
 game.play();
 System.out.println();
 game = new Football();
 game.play();
 }
 }
```

Step 4

Verify the output.
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!
Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!

MVC Pattern

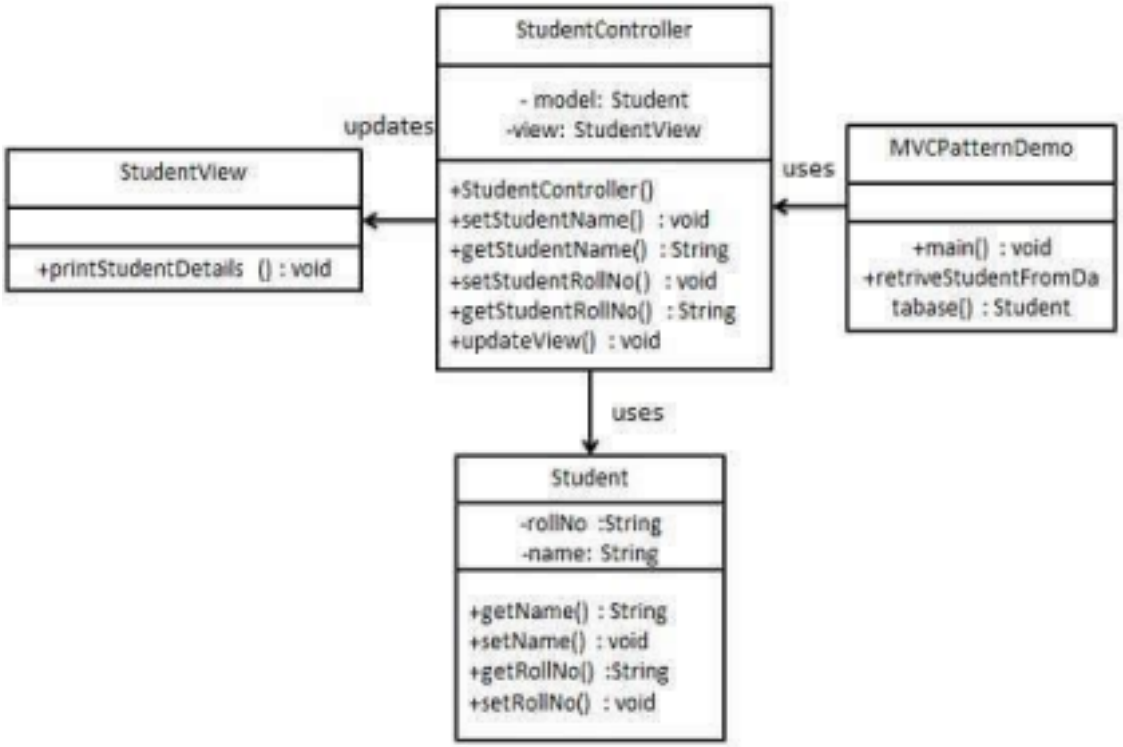
MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

- **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualization of the data that model contains. · **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Implementation

We are going to create a *Student* object acting as a model.*StudentView* will be a view class which can print student details on console and *StudentController* is the controller class responsible to store data in *Student* object and update view *StudentView* accordingly.

MVCPatternDemo, our demo class, will use *StudentController* to demonstrate use of MVC pattern.



Step 1

Create Model.

Student.java

```
public class Student {
 private String rollNo;
 private String name;

 public String getRollNo() {
 return rollNo;
 }

 public void setRollNo(String rollNo) {
 this.rollNo = rollNo;
 }

 public String getName() {
 return name;
 }

 public void setName(String name) {
```

```
 this.name = name;
 }

 }
```

Step 2

Create View.

StudentView.java

```
public class StudentView {
    public void printStudentDetails(String studentName, String
studentRollNo){  System.out.println("Student: ");
 System.out.println("Name: " + studentName);
 System.out.println("Roll No: " + studentRollNo);
 }
 }
```

Step 3

Create Controller.

StudentController.java

```
public class StudentController {
 private Student model;
 private StudentView view;

  public  StudentController(Student  model,  StudentView
view){  this.model = model;
 this.view = view;
 }

 public void setStudentName(String name){
```

```
 model.setName(name);
 }

 public String getStudentName(){
 return model.getName();
 }

 public void setStudentRollNo(String rollNo){
 model.setRollNo(rollNo);
 }

 public String getStudentRollNo(){
 return model.getRollNo();
 }

 public void updateView(){
                        view.printStudentDetails(model.getName(),
model.getRollNo());  }
}
```

Step 4

Use the *StudentController* methods to demonstrate MVC design pattern usage.

MVCPatternDemo.java

```
public class MVCPatternDemo {
 public static void main(String[] args) {

 //fetch student record based on his roll no from the database
Student model = retriveStudentFromDatabase();
```

```
 //Create a view : to write student details on console
StudentView view = new StudentView();

  StudentController controller = new StudentController(model,
view);  controller.updateView();
 //update model data
 controller.setStudentName("John");

 controller.updateView();
 }

 private static Student
retriveStudentFromDatabase(){  Student student =
new Student();
 student.setName("Robert");
 student.setRollNo("10");
 return student;
 }
}
```

Step 5

Verify the output.

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```