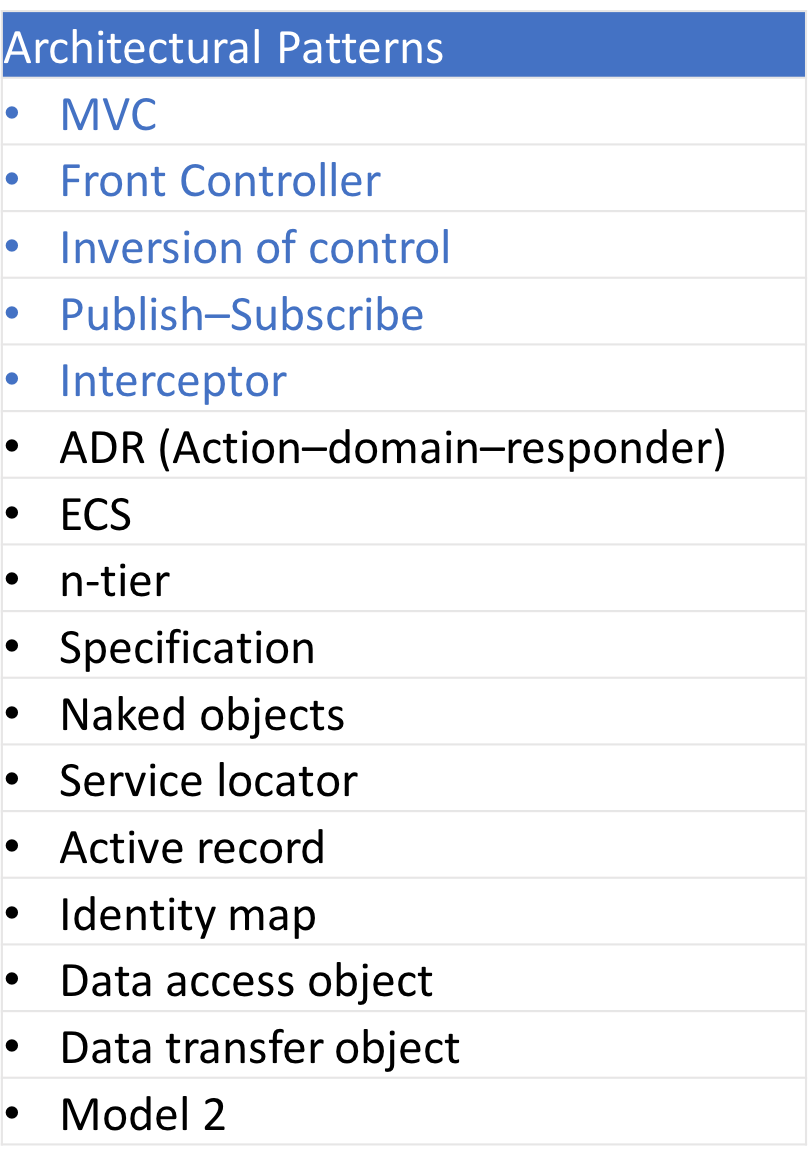
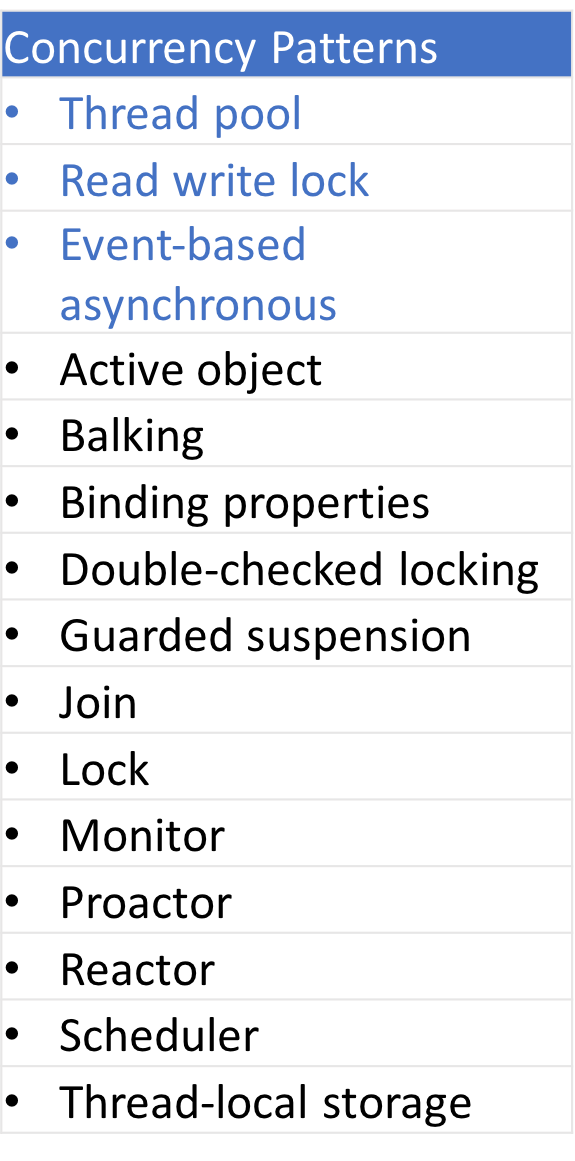
# Design Pattern

## What is Design Pattern? Why?

* General & Re-usable solutions — to common ‘software design’ problems
* It provides professional “Language” to talk in Software Design area
  + Developers can easily communicate each other (more professionally)
  + E.g. Developer can say, we should use “Factory Pattern” here
  + E.g. Developer can say, we should use “Singleton Pattern” here
* First Book in Software Industry: Gang of Four Design Patterns

## Design Pattern Classifications







## Singleton Design Pattern

Singleton pattern

* restricts the instantiation of a class
* and ensures that only one instance of the class exists in the Java Virtual Machine

Singleton Design Pattern (Various Implementations):

### 1. Eager initialization:

*// Java code to create singleton class by*

*// Eager Initialization*

public class GFG {

*// public instance initialized when loading the class*

public static GFG instance = new GFG();

private GFG() {

*// private constructor*

}

}

**Cons:** May lead to resource wastage. Because instance of class is created always, whether it is required or not.

### 2. Lazy initialization:

*//Java Code to create singleton class*

*// With Lazy initialization*

public class GFG {

*// private instance, so that it can be*

*// accessed by only by getInstance() method*

private static GFG instance;

private GFG() {

*// private constructor*

}

*// method to return instance of class*

public static GFG getInstance() {

if (instance == null) {

*// if instance is null, initialize*

instance = new GFG();

}

return instance;

}

}

**Cons:** NOT **-**Thread Safe (In multithreaded environment, it may break singleton property)

### 3. Thread Safe Singleton:

singleton property is maintained even in multithreaded environment

*// Java program to create Thread Safe*

*// Singleton class*

public class GFG {

*// private instance, so that it can be*

*// accessed by only by getInstance() method*

private static GFG instance;

private GFG() {

*// private constructor*

}

*// synchronized method to control simultaneous access*

synchronized public static GFG getInstance() {

if (instance == null) {

*// if instance is null, initialize*

instance = new GFG();

}

return instance;

}

}

**Cons:** getInstance() method is synchronized so it causes slow performance as multiple threads can’t access it simultaneously.

### 4. Lazy initialization with Double check locking:

In this method, getInstance is not synchronized but the block which creates instance is synchronized so that minimum number of threads have to wait and that’s only for first time.

*// Java code to explain double check locking*

public class GFG {

*// private instance, so that it can be*

*// accessed by only by getInstance() method*

private static GFG instance;

private GFG() {

*// private constructor*

}

public static GFG getInstance() {

if (instance == null) {

*// synchronized block to remove overhead*

synchronized (GFG.class) {

if (instance == null) {

*// if instance is null, initialize*

instance = new GFG();

}

}

}

return instance;

}

}

**Cons:**

1. First time, it can affect performance.

As cons. of double check locking method is bearable so it can be used for high performance multi-threaded applications.

### 5. Bill Pugh Singleton Implementation:

Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment.

*// Java code for Bill Pugh Singleton Implementaion*

public class GFG {

private GFG() {

*// private constructor*

}

*// Inner class to provide instance of class*

private static class BillPughSingleton {

private static final GFG INSTANCE = new GFG();

}

public static GFG getInstance() {

return BillPughSingleton.INSTANCE;

}

}

## Factory Method –Design Pattern

In Factory pattern,

* we create object without exposing the creation logic to the client
* and refer to newly created object using a common interface.



interface Shape {

void draw();

}

class Rectangle implements Shape {

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

class Square implements Shape {

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

class Circle implements Shape {

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

*// Create a Factory to generate object of concrete class based on given information.*

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;

}

}

*// Use the Factory to get object of concrete class by passing an information such as type.*

public class MyMainClass {

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"); *// Inside Circle::draw() method.*

*//call draw method of Circle*

shape1.draw();

*//get an object of Rectangle and call its draw method.*

Shape shape2 = shapeFactory.getShape("RECTANGLE"); *// Inside Rectangle::draw() method.*

*//call draw method of Rectangle*

shape2.draw();

*//get an object of Square and call its draw method.*

Shape shape3 = shapeFactory.getShape("SQUARE"); *// Inside Square::draw() method.*

*//call draw method of square*

shape3.draw();

}

}

## MVC

## Front Controller

## Inversion of control

## Publish–Subscribe

*//Event Driven - Javascript (publish subscribe) pattern*

var events = {

events: {},

on: function (eventName, fn) {

this.events[eventName] = this.events[eventName] || [];

this.events[eventName].push(fn);

},

off: function (eventName, fn) {

if (this.events[eventName]) {

for (var i = 0; i < this.events[eventName].length; i++) {

if (this.events[eventName][i] === fn) {

this.events[eventName].splice(i, 1);

break;

}

};

}

},

emit: function (eventName, data) {

if (this.events[eventName]) {

this.events[eventName].forEach(function (fn) {

fn(data);

});

}

}

};