DESIGN PATTERNS

i. Defn: Pattern

A generalized solution to a software development problem based on well proven experience that promotes good design practice, in a given context.
Design pattern is a description of a commonly recurring structure of communicating components that solves a general design problem within a particular context.

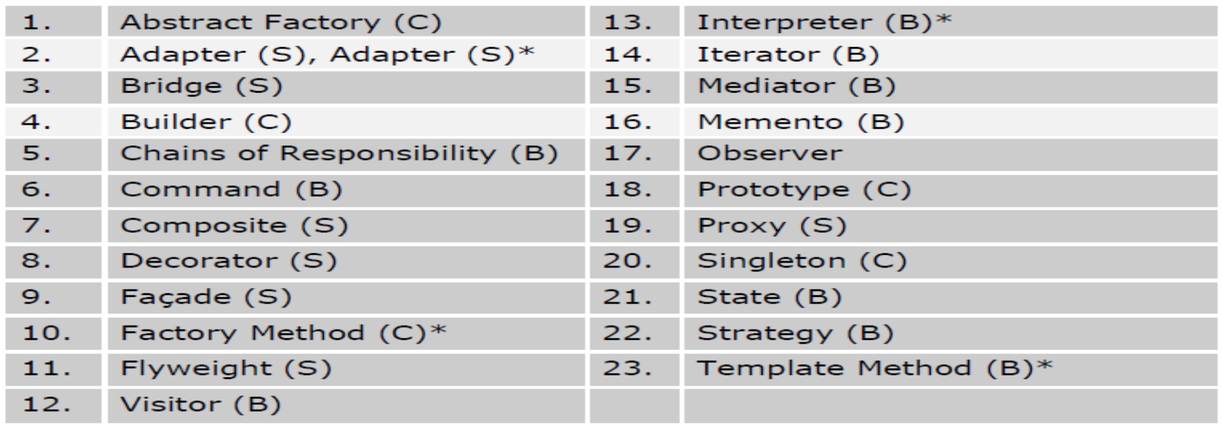
ii. Elements of Patterns

1) name: a handle which describes the design problem, its solution, and consequences in a word or two
2) problem: describes when to apply the pattern and explains the problem and its context
3) solution: elements that must make up the design, their relationships, responsibilities and collaborations.
4) consequences: associated trade-offs in using the pattern.

iii. Types of Design Patterns

1) Creational Patterns
a) instantiation of objects
b) decoupling the type of objects from the process of constructing that object
2) Structural and Architectural patterns
a) organization of a system
b) larger structures composed from smaller structures.
3) Behavioural Patterns
a) assigning responsibilities among a collection of objects.

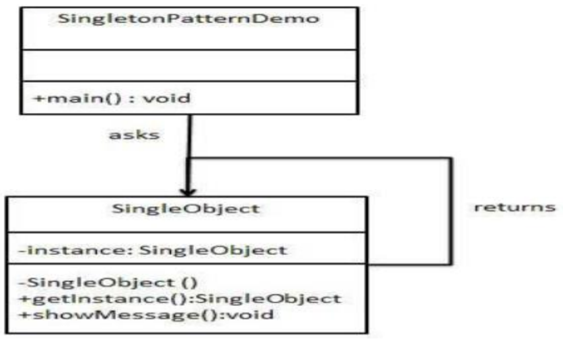
iv. Example of Design Patterns



1.	Abstract Factory (C)	13.	Interpreter (B)*
2.	Adapter (S), Adapter (S)*	14.	Iterator (B)
3.	Bridge (S)	15.	Mediator (B)
4.	Builder (C)	16.	Memento (B)
5.	Chains of Responsibility (B)	17.	Observer
6.	Command (B)	18.	Prototype (C)
7.	Composite (S)	19.	Proxy (S)
8.	Decorator (S)	20.	Singleton (C)
9.	Façade (S)	21.	State (B)
10.	Factory Method (C)*	22.	Strategy (B)
11.	Flyweight (S)	23.	Template Method (B)*
12.	Visitor (B)		

1) Name: **Singleton (creational)**
2) Problem:
a) when you want to be sure that a class can only ever have a single instance
b) when a single class is responsible for creating its own object and provides a way to access it directly without instantiation
3) Solution:

Class Diagram



SingleObject class provides a static method to get its static instance to outside world.*SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.

4) Consequences:
a) no public constructor
b) if someone wants to make instance must call getInstance

5) Realworld applicability:
The file system instantiated by the operating system is accessed by applications through singleton

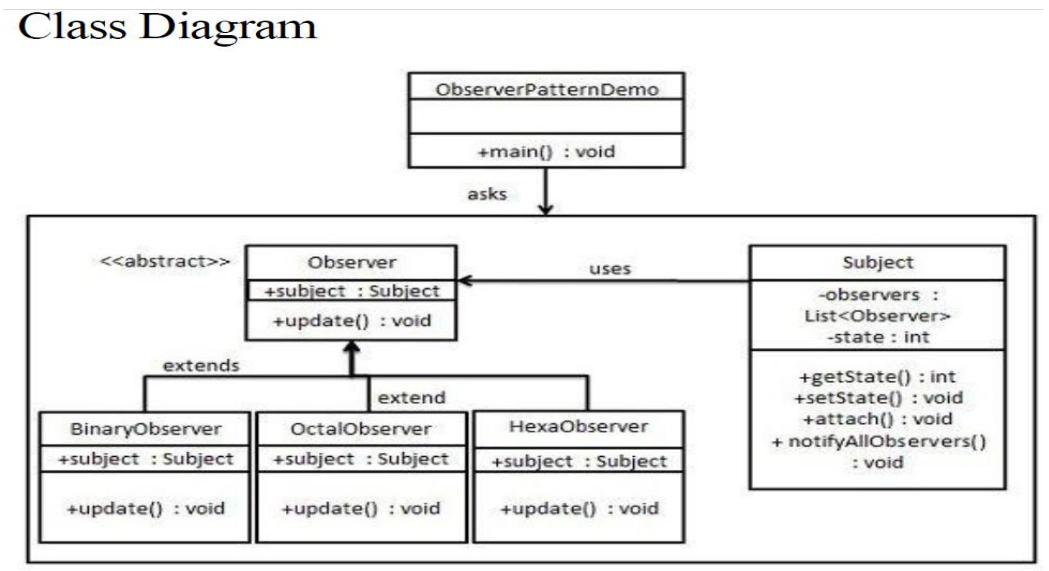
6) Implementation:
Step 1
Create a Singleton Class.
SingleObject.java
```
public class SingleObject {
        //create an object of SingleObject
        private static SingleObject instance = new SingleObject();

        //make the constructor private so that this class cannot be
        //instantiated
        private SingleObject(){}

        //Get the only object available
        public static SingleObject getInstance(){
                return instance;
        }
        public void showMessage(){
                System.out.println("Hello World!");
        }
}
```
Step 2
Get the only object from the singleton class.
SingletonPatternDemo.java
```
public class SingletonPatternDemo {
        public static void main(String[] args) {
                //illegal construct
                //Compile Time Error: The constructor SingleObject() is not visible
                //SingleObject object = new SingleObject();

                //Get the only object available
                SingleObject object = SingleObject.getInstance();
                //show the message
                object.showMessage();
        }
}
```

Step 3
Verify the output.
```
Hello World!
```

ALSO

```
Public class FileSystem
{
      private static FileSystem fs = null; //note static and private
      private FileSystem() //note private constructor, can't be called from outside
      {
      <appropriate initialisation>
      }
      public static FileSystem getInstance() //note static and public
      {
             if (fs == null) { //if instance doesn't already exist create it and store
             fs = new FileSystem();
             //do anything else that might need to be done here (initialisation)
             }
      return fs; //return stored instance
      }
}
```

1) Name: **Observer (behaviorial)**
2) Problem:
a) when an abstraction has two aspects, one dependent on the other
b) when a change to one object requires changing others, and you don't know how many objects need to be changed
c) when an object should be able to notify other object without making assumptions about who these objects are
3) Solution:



Observer pattern uses three actor classes Subject, Observer and Client. Subject an object having methods to attach and de-attach observers to a client object. We've created classes *Subject*, *Observer*abstract class and concrete classes extending the abstract class the *Observer*.

4) Consequences:
a) abstract coupling between Subject and Observer
b) support for broadcast communication
c) unexpected updates

5) Realworld applicability:
Messages needed to be sent to citizens each time a typhoon approaches

6) Implementation:

Step 1

Create Subject class.

Subject.java

```
import java.util.ArrayList;
import java.util.List;
public class Subject {
       private List<Observer> observers = new ArrayList<Observer>();
       private int state;

       public int getState() {
              return state;
       }
       public void setState(int state) {
              this.state = state;
              notifyAllObservers();
       }
       public void attach(Observer observer){
              observers.add(observer);
       }
       public void notifyAllObservers(){
              for (Observer observer : observers) {
              observer.update();
              }
       }
}
```

Step 2

Create Observer class.

Observer.java

```
public abstract class Observer {
       protected Subject subject;
       public abstract void update();
}
```

Step 3

Create concrete observer classes

BinaryObserver.java

```
public class BinaryObserver extends Observer{
       public BinaryObserver(Subject subject){
              this.subject = subject;
              this.subject.attach(this);
       }
       @Override
       public void update() {
       System.out.println("Binary String:"+
       Integer.toBinaryString(subject.getState()));
       }
}
```

OctalObserver.java

```
public class OctalObserver extends Observer{
       public OctalObserver(Subject subject){
              this.subject = subject;
              this.subject.attach(this);
       }
       @Override
       public void update() {
              System.out.println("Octal String:"+
              Integer.toOctalString(subject.getSta
              te()));
       }
}
```

HexaObserver.java

```
public class HexaObserver extends Observer{
	public HexaObserver(Subject subject){
		this.subject = subject;
		this.subject.attach(this);
	}
	@Override
	public void update() {
	System.out.println("Hex String:"+Integer.toHexString(subject.getState()
	).toUpperCase());
	}
}
```

Step 4
Use *Subject* and concrete observer objects.
ObserverPatternDemo.java

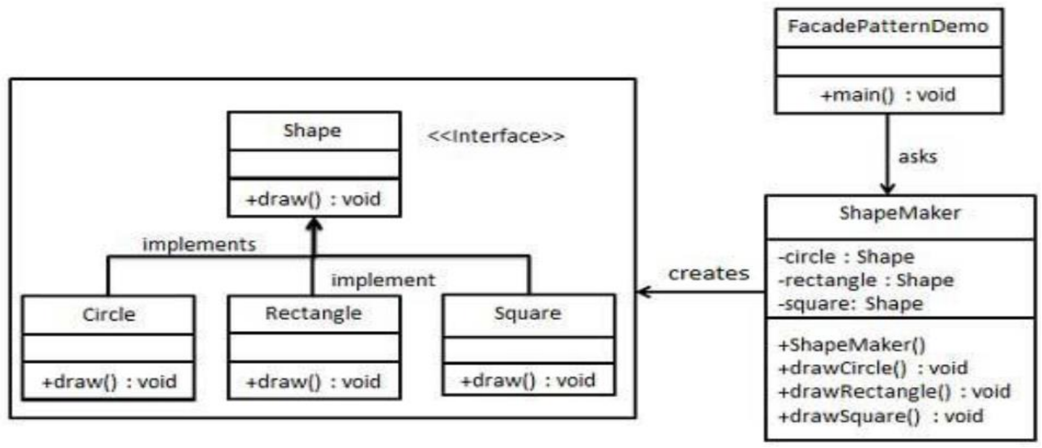
```
public class ObserverPatternDemo {
	public static void main(String[] args) {
		Subject subject = new Subject();
		new HexaObserver(subject);
		new OctalObserver(subject);
		new BinaryObserver(subject);
		System.out.println("First state change: 15");
		subject.setState(15);
		System.out.println("Second state change: 10");
		subject.setState(10);
	}
}
```

Step 5
Verify the output.

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

1) Name: **Facade (Structural)**
2) Problem:
a) when dependencies among classes is complex to the client
b) when a simplified interface is needed to reduce complexity in accessing the system
c) when simplified methods are required by the client to delegate calls to existing complex system's methods
3) Solution:

Class Diagram



A *Shape* interface and concrete classes implementing the *Shape* interface are defined. A facade class *ShapeMaker* is defined that uses the concrete classes and delegates user calls to these classes. *FacadePatternDemo*, our demo class will use *ShapeMaker* class to show the results.

4) Consequences:
a) reduces dependencies among classes by encapsulating subsystems with simple unified interfaces

5) Realworld applicability:
A need to provide several buttons on a web form which executes different actions

6) Implementation:
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("Rectangle::draw()");
       }
}
```
Square.java
```
public class Square implements Shape {
       @Override
       public void draw() {
              System.out.println("Square::draw()");
       }
}
```

Circle.java

```
public class Circle implements Shape {
       @Override
       public void draw() {
              System.out.println("Circle::draw()");
       }
}
```

<u>Step 3</u>

Create a facade class.

ShapeMaker.java

```
public class ShapeMaker {
       private Shape circle;
       private Shape rectangle;
       private Shape square;
       public ShapeMaker() {
              circle = new Circle();
              rectangle = new Rectangle();
              square = new Square();
       }
       public void drawCircle(){
              circle.draw();
       }
       public void drawRectangle(){
              rectangle.draw();
       }
       public void drawSquare(){
              square.draw();
       }
}
```

<u>Step 4</u>

Use the facade to draw various
types of shapes.

FacadePatternDemo.java

```
public class FacadePatternDemo {
       public static void main(String[] args) {
              ShapeMaker shapeMaker = new ShapeMaker();
              shapeMaker.drawCircle();
              shapeMaker.drawRectangle();
              shapeMaker.drawSquare();
       }
}
```

<u>Step 5</u>

Verify the output.

```
Circle::draw()
Rectangle::draw()
Square::draw()
```