Design Defects and Restructuring

Engr. Abdul-Rahman Mahmood

abdulrahman@nu.edu.pk

alphapeeler.sf.net/pubkeys/pkey.htm

pk.linkedin.com/in/armahmood

www.twitter.com/alphapeeler

www.facebook.com/alphapeeler

abdulmahmood-sss   alphasecure

armahmood786

http://alphapeeler.sf.net/me

alphapeeler#9321

reddit.com/user/alphapeeler

www.flickr.com/alphapeeler

http://alphapeeler.tumblr.com

armahmood786@jabber.org

alphapeeler@aim.com

mahmood_cubix   48660186

alphapeeler@icloud.com

pinterest.com/alphapeeler

www.youtube.com/user/AlphaPeeler

Observer pattern

What is Observer pattern?

Observer is one of the 23 Design Patterns which were selected by the GoF (Gang of Four).

|		Purpose		
		Creation	Structure	Behavior
Scope	Class	Factory Method	Interpreter Template	Interpreter Template
	Objects	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer ⬅ State Strategy Visitor

Observer Pattern

- Defines a "one-to-many" dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

- a.k.a Dependence mechanism / publish-subscribe / broadcast / change-update
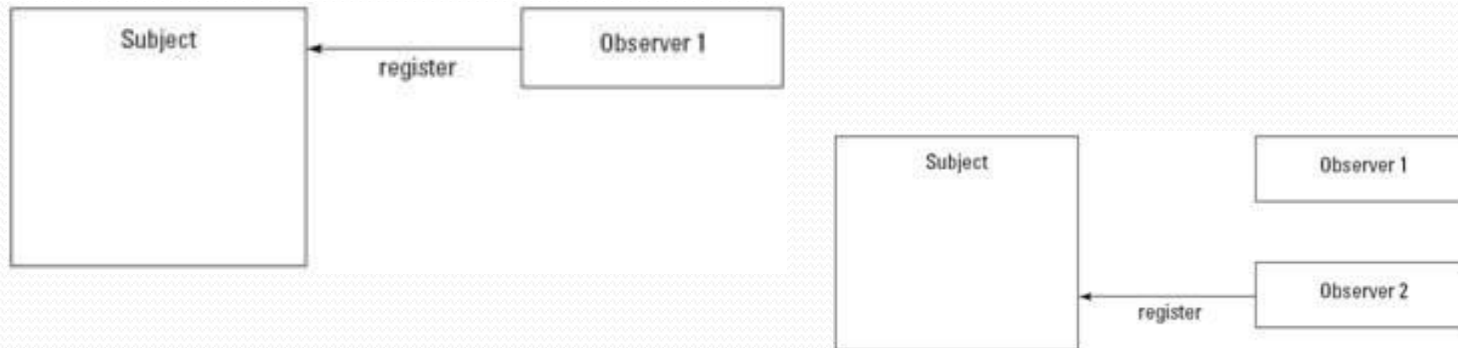
Subject & Observer

- Subject
  - the object which will frequently change its state and upon which other objects depend

- Observer
  - the object which depends on a subject and updates according to its subject's state.
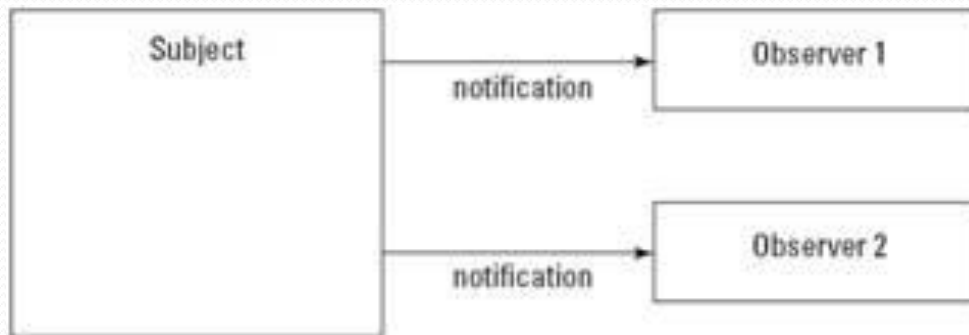
Observer Pattern - Example
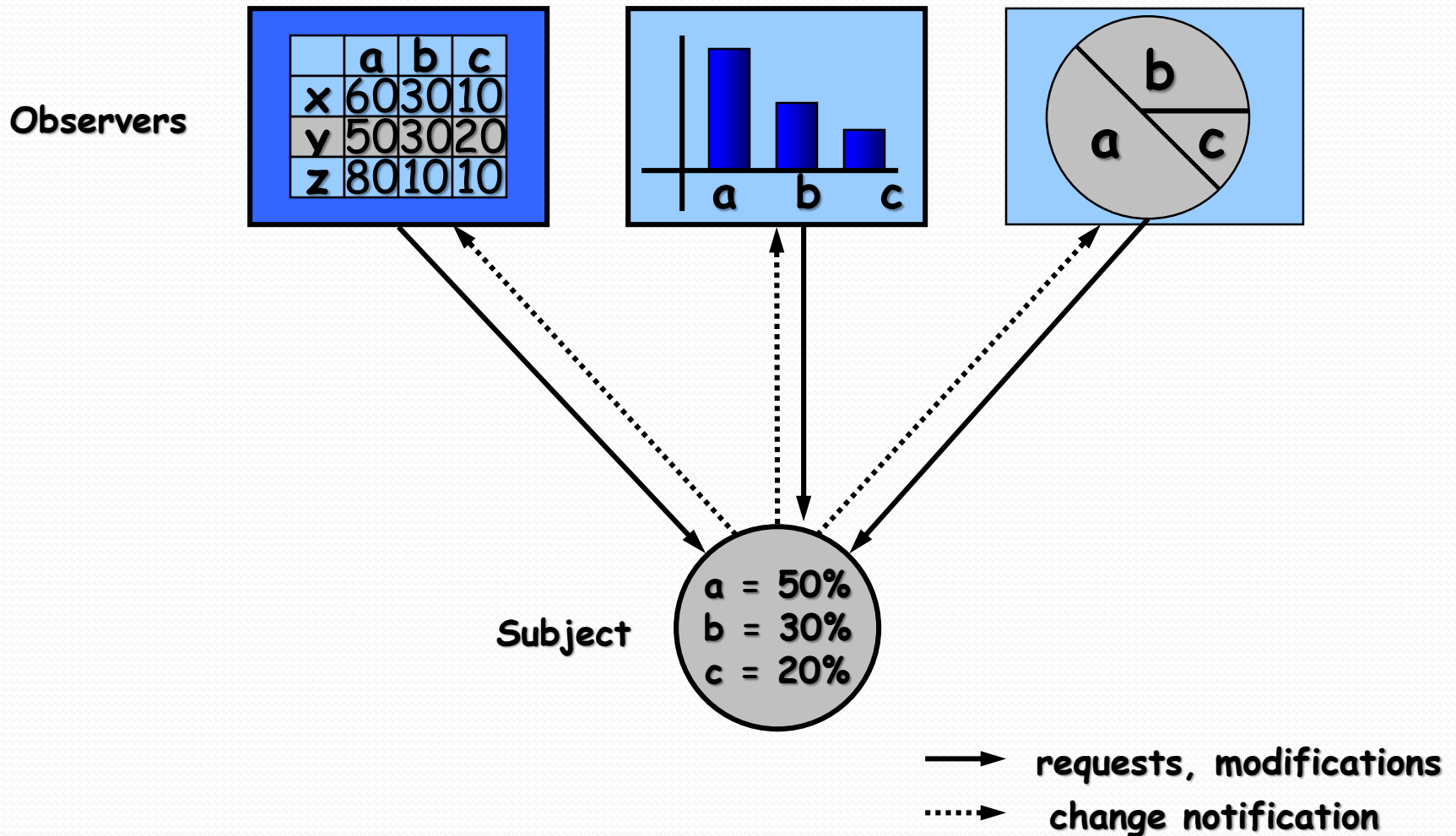
Observer Pattern - Working

A number of Observers "register" to receive notifications of changes to the Subject. Observers are not aware of the presence of each other.


When a certain event or "change" in Subject occurs, all Observers are "notified'.

Observer Pattern - Key Players

- Subject
  - has a list of observers
  - Interfaces for attaching/detaching an observer
- Observer
  - An updating interface for objects that gets notified of changes in a subject
- ConcreteSubject
  - Stores "state of interest" to observers
  - Sends notification when state changes
- ConcreteObserver
  - Implements updating interface

Observer Pattern - UML
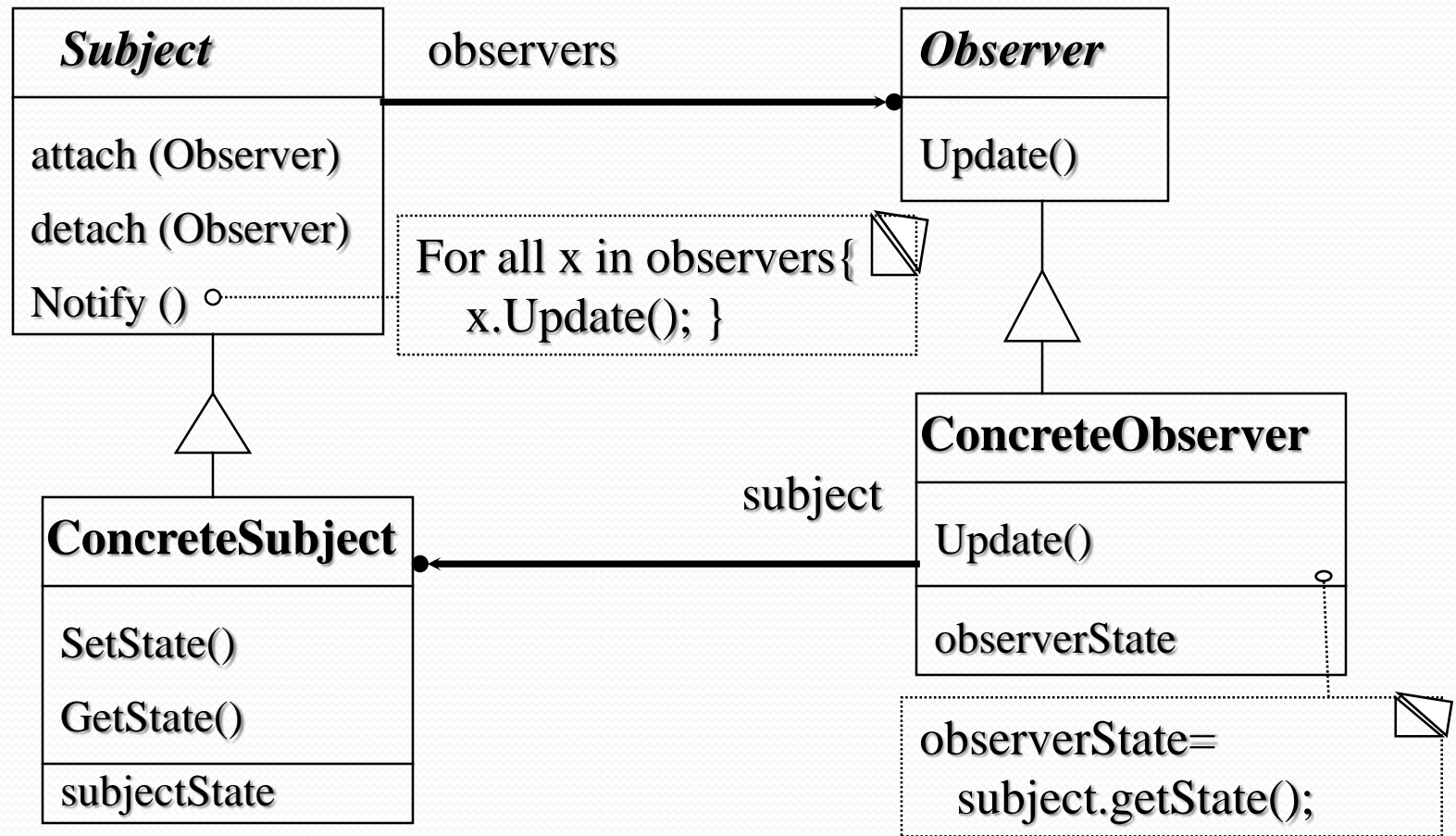
Observer Pattern - Collaborations
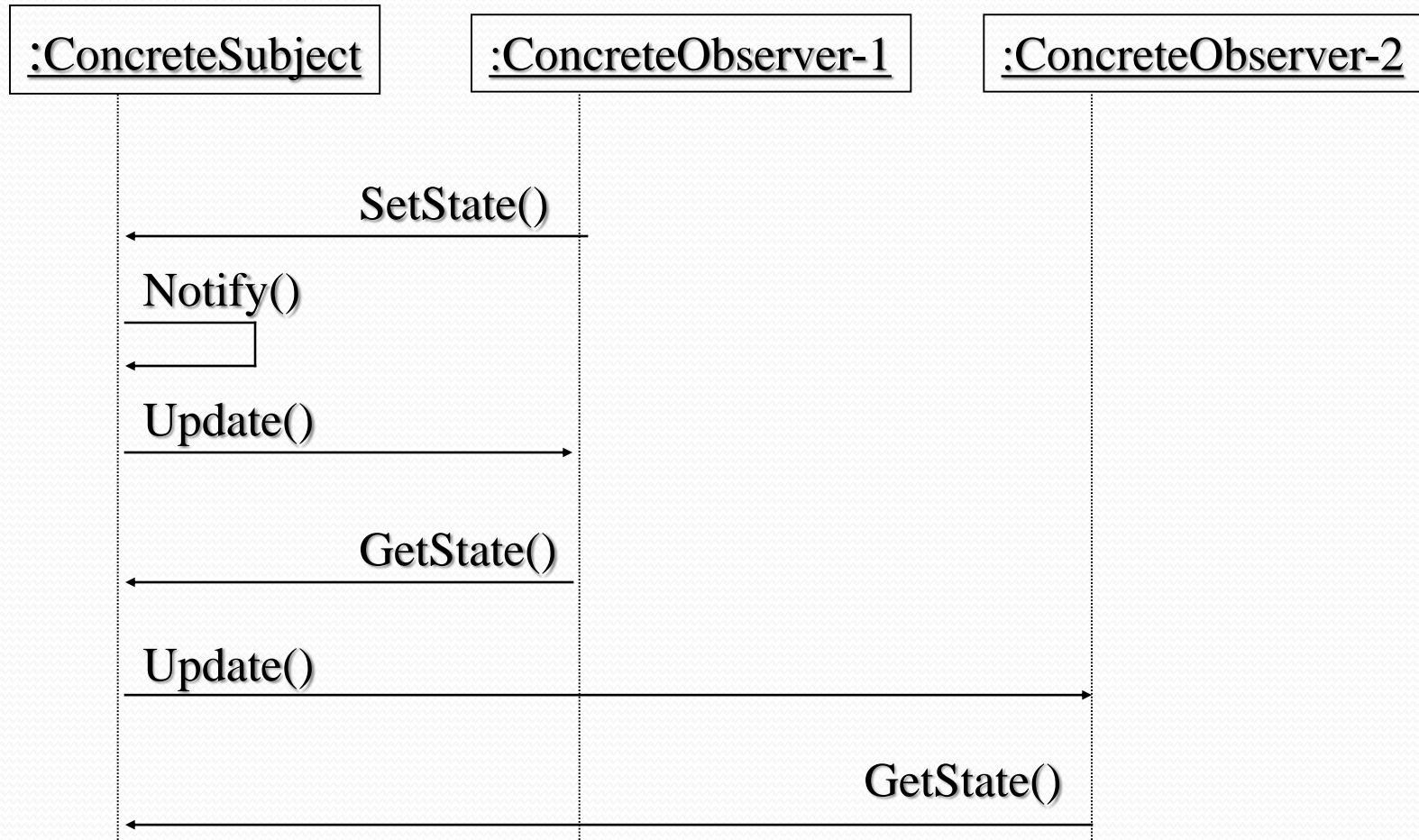
Consequences

- Loosely Coupled
  - Reuse subjects without reusing their observers, and vice versa
  - Add observers without modifying the subject or other observers

- Abstract coupling between subject and observer
  - Concrete class of none of the observers is known

- Support for broadcast communication
  - Subject doesn't need to know its receivers

- *ObserverPatternDemo*, our demo class, will use *Subject* and concrete class object to show observer pattern in action.
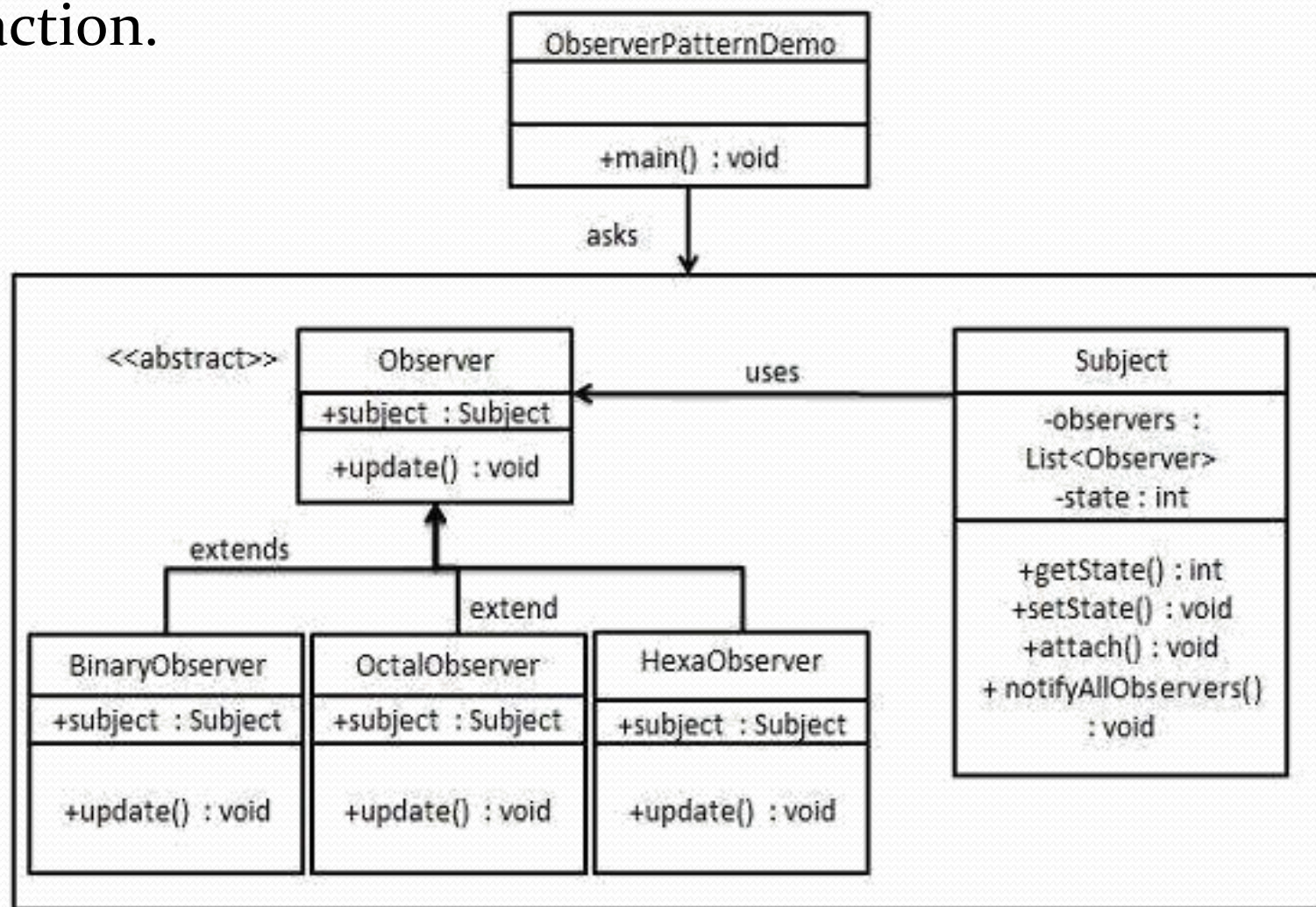
Step 1: Create *Subject.java*

```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.java*

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

Step 3 Create concrete observer classes

```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() ) );
    }
}
```

```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 3 Create concrete observer classes

```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.getState() ) );
  }
}
```

OctalObserver.java

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

```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