Centro Universitário UniBH
Ciência da Computação
Práticas de Programação
Professor: Lucas Schmidt

Aula Prática - 19 Padrões de Comportamento - Observer

Intenção

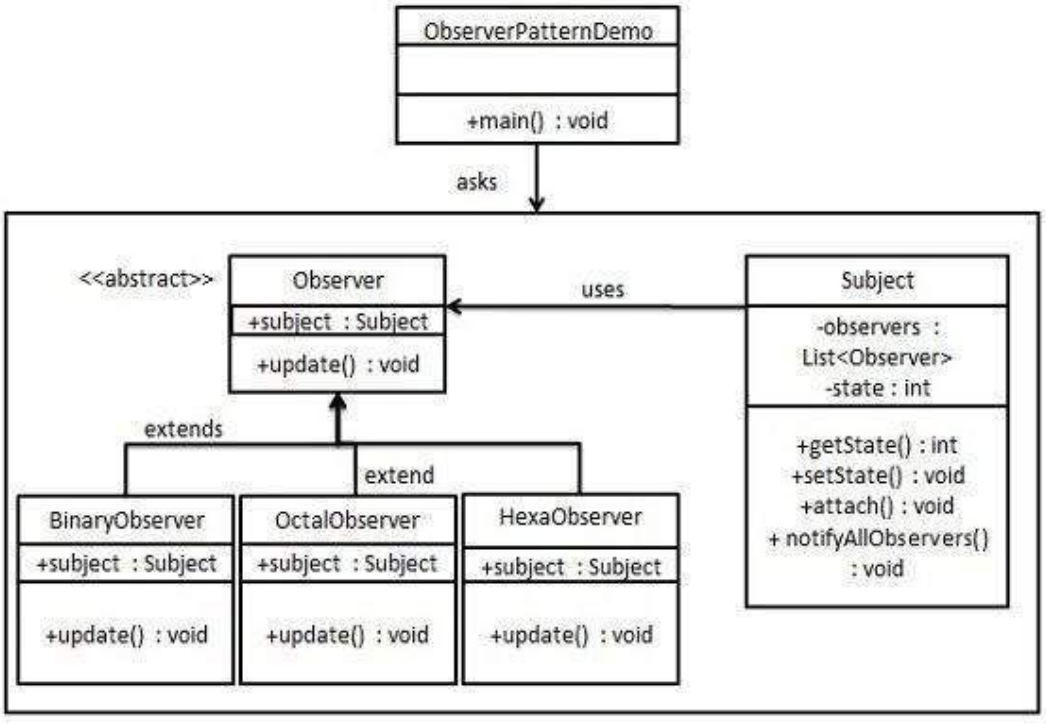
Definir uma dependência um-para-muitos entre objetos de forma que quando um objeto muda de estado, os outros são notificados e se atualizam. Também conhecido como: Dependents, Publish-Subscribe.

Usar este padrão quando…

- Uma abstração possui dois aspectos e é necessário separá-los em dois objetos para variá-los;
- Alterações num objeto requerem atualizações em vários outros objetos não-determinados;
- Um objeto precisa notificar sobre alterações em outros objetos que, a princípio, ele não conhece.

Vantagens e desvantagens

- Flexibilidade:
  - Observável e observadores podem ser quaisquer objetos;
  - Acoplamento fraco entre os objetos: não sabem a classe concreta uns dos outros;
  - É feito broadcast da notificação para todos, independente de quantos;
  - Observadores podem ser observáveis de outros, propagando em cascata.



Passo 1
Criar classe de Subject.

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();
      }
   }
}
```

Passo 2
Criar classe Observer.

Observer.java

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

Passo 3
Crie classes concretas de observadores

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

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() );
   }
}
```

Passo 4
Use o Subject e objetos observadores concretos.

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);
   }
}
```

Passo 5

Teste sua implementação!