NOMBRE			
AREA	Software	REGIMEN	Semestral

Tema: Patrones de diseño

Caso Patrón Observer

Este patrón de diseño permite reaccionar a ciertas clases llamadas observadores sobre un evento determinado.

Es usado en programación para monitorear el estado de un objeto en un programa. Está relacionado con el principio de invocación implícita. La motivación principal de este patrón es su utilización como un sistema de detección de eventos en tiempo de ejecución. Es una característica muy interesante en términos del desarrollo de aplicaciones en tiempo real.

Debe ser utilizado cuando:

Un objeto necesita notificar a otros objetos cuando cambia su estado. La idea es encapsular estos aspectos en objetos diferentes permite variarlos y reutilizarlos independientemente.

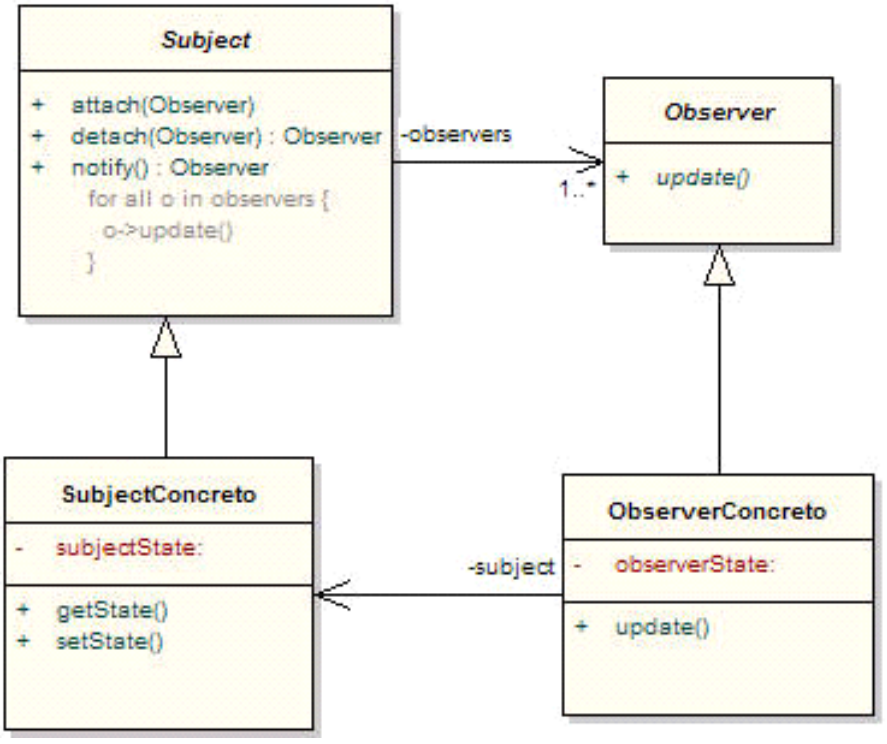
Cuando existe una relación de dependencia de uno a muchos que puede requerir que un objeto notifique a múltiples objetos que dependen de él cuando cambia su estado.

Este patrón tiene un uso muy concreto: varios objetos necesitan ser notificados de un evento y cada uno de ellos deciden cómo reaccionar cuando esta evento se produzca.

Un caso típico es la Bolsa de Comercio, donde se trabaja con las acciones de las empresas. Imaginemos que muchas empresas están monitoreando las acciones una empresa X. Posiblemente si estas acciones bajan, algunas personas estén interesadas en vender acciones, otras en comprar, otras quizás no hagan nada y la empresa X quizás tome alguna decisión por su cuenta. Todos reaccionan distinto ante el mismo evento. Esta es la idea de este patrón y son estos casos donde debe ser utilizado.

Diagramas UML

Vista estática: diagrama de clases

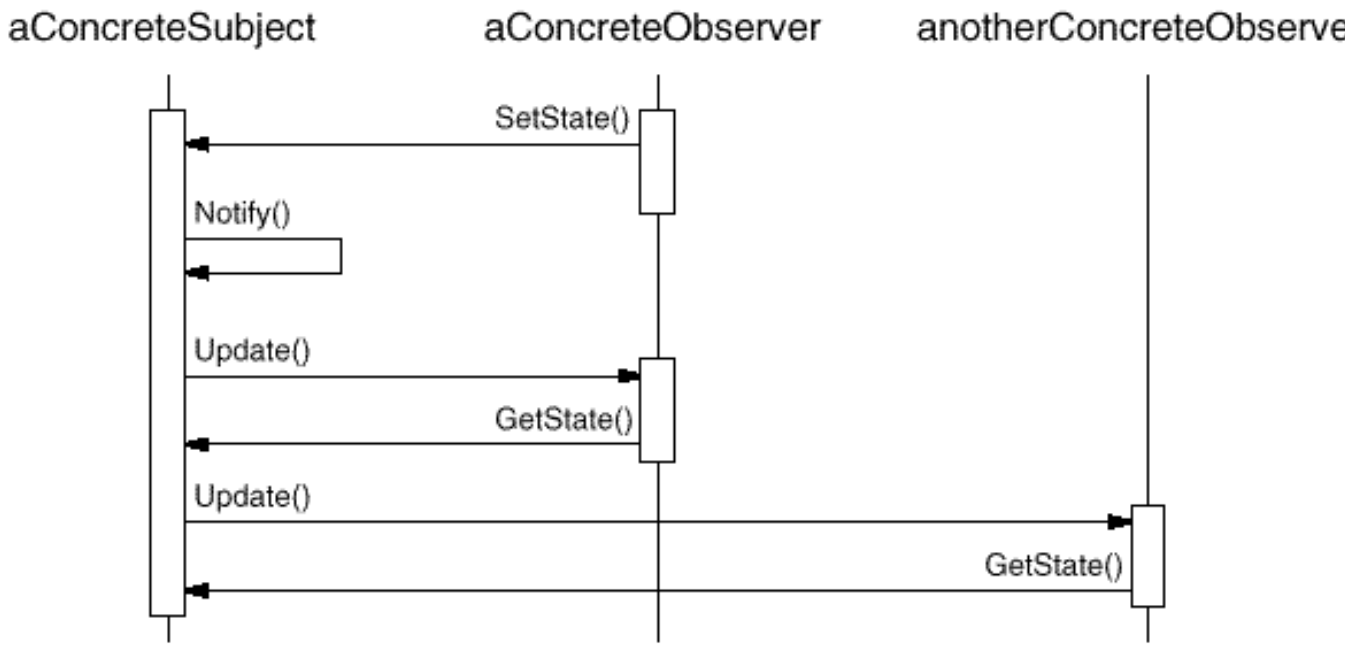


Subject: conoce a sus observadores y ofrece la posibilidad de añadir y eliminar observadores. Posee un método llamado attach () y otro detach () que sirven para agregar o remover observadores en tiempo de ejecución.

Observer: define la interfaz que sirve para notificar a los observadores los cambios realizados en el Subject.

SubjectConcreto: almacena el estado que es objeto de interés de los observadores y envía un mensaje a sus observadores cuando su estado cambia.
ObserverConcreto: mantiene una referencia a un SubjectConcreto. Almacena el estado del Subject que le resulta de interés. Implementa la interfaz de actualización de Observer para mantener la consistencia entre los dos estados.

Vista Dinámica: Diagrama de secuencia



Un observador inicializa *la petición de cambio de estado y espera la respuesta. El sujeto Concreto notifica el cambio de estado y esta acción desencadena la operación de actualización de los observadores. Quienes solicitan el valor del nuevo estado del Sujeto. Pueden existir variaciones en la implementación*

Caso práctico a desarrollar en clase

Vamos a suponer un ejemplo de una Biblioteca, donde cada vez que un lector devuelve un libro se ejecuta el método devuelveLibro (Libro libro) de la clase Biblioteca.Si el lector devolvió el libro dañado entonces la aplicación avisa a ciertas clases que están interesadas en conocer este evento:

```java
public class Libro {
    private String tiulo;
    private String estado;

    // Un libro seguramente tendrá más atributos
    // como autor, editorial, etc pero para nuestr
    // ejemplo no son necesarios.

    public String getTiulo() {
        return tiulo;
    }

    public void setTiulo(String tiulo) {
        this.tiulo = tiulo;
    }
```

Cada clase que quiera observar el cambio del estado en el libro deberá implementar la siguiente interface y darle lógica al método update:

```
public interface ILibroMalEstado {

    public void update();
}
```

```java
public class Administracion implements ILibroMalEstado {

    public void update() {
        System.out.println("Administracion: ");
        System.out.println("Envio una queja formal...");
    }

}
```

```java
public class Compras implements ILibroMalEstado {

    public void update() {
        System.out.println("Compras: ");
        System.out.println("Solicito nueva cotizacion...
    }
}
```

```java
public class Compras implements ILibroMalEstado {

    public void update() {
        System.out.println("Compras: ");
        System.out.println("Solicito nueva cotizacion.
    }
}
```

```java
public class Stock implements ILibroMalEstado {

public void update() {
        System.out.println("Stock: ");
        System.out.println("Le doy de baja...");
    }
}
```

Por otro lado está el Subject

```
public interface Subject {

    public void attach(ILibroMalEstado observad
    public void dettach(ILibroMalEstado observa
    public void notifyObservers();

}
```

que es implementado por AlarmaLibro

```
public class AlarmaLibro implements Subject {
    private static ArrayList<ILibroMalEstado> observador
        new ArrayList<ILibroMalEstado>();

    public void attach(ILibroMalEstado observador) {
        observadores.add(observador);
    }

    public void dettach(ILibroMalEstado observador) {
        observadores.remove(observador);
    }

    public void notifyObservers() {
        for (int i = 0; i < observadores.size(); i++) {
            observadores.get(i).update();
        }
    }

}
```

La Biblioteca es quien dispara el evento.

```
public class Biblioteca {

    public void devuelveLibro(Libro libro) {
        if (libro.getEstado().equals("MALO")) {
            AlarmaLibro a = new AlarmaLibro();
            a.notifyObservers();
        }
    }

}
```

El próximo paso es hacerlo funcionar creando el procedimiento main se crean el objeto que es sujeto de observación: AlarmaLibro y los objetos que intervienen como observadores: Administración, Stock y Compras.
Se genera el evento al crear un libro y setear su estado. En este caso se setea como "*MALO*". Inmediatamente se puede crear el objeto Biblioteca que genera el procedimiento *devuelveLibro*.

```
    public static void main(String[] args
        AlarmaLibro a = new AlarmaLibro()
        a.attach(new Compras());
        a.attach(new Administracion());
        a.attach(new Stock());

        Libro libro = new Libro();
        libro.setEstado("MALO");

        Biblioteca b = new Biblioteca();
        b.devuelveLibro(libro);

    }
```

Problems | Javadoc | Declaration | Console

```
<terminated> Main (12) [Java Application] C:\Program Files\Java\jre
Solicito nueva cotizacion...
Administracion:
Envio una queja formal...
Stock:
Le doy de baja...
```

Consecuencias

- Permite modificar las clases subjects y las observers independientemente.
- Permite añadir nuevos observadores en tiempo de ejecución, sin que esto afecte a ningún otro observador.
- Permite que dos capas de diferentes niveles de abstracción se puedan comunicar entre sí sin romper esa división.
- Permite comunicación broadcast, es decir, un objeto subject envía su notificación a todos los observers sin enviárselo a ningún observer en concreto (el mensaje no tiene un destinatario concreto). Todos los observers reciben el mensaje y deciden si hacerle caso o ignorarlo.
- La comunicación entre los objetos subject y sus observadores es limitada: el evento siempre significa que se ha producido algún cambio en el estado del objeto y el mensaje no indica el destinatario.

Variaciones posibles del caso práctico

Si los observadores pueden observar a varios objetos subject a la vez, es necesario ampliar el servicio update () para permitir conocer a un objeto observer dado cuál de los objetos subject que observa le ha enviado el mensaje de notificación.
Una forma de implementarlo es añadiendo un parámetro al servicio update () que sea el objeto subject que envía la notificación (el remitente). Y añadir una lista de objetos subject observados por el objeto observer en la clase Observer.
Si los objetos observers observan varios eventos de interés que pueden suceder con los objetos subjects, es necesario ampliar el servicio add() y el update() además de la implementación del mapeo subject-observers en la clase abstracta Subject. Una forma de implementarlo consiste en introducir un nuevo parámetro al servicio add() que indique el evento de interés del observer a añadir e introducirlo también como un nuevo parámetro en el servicio update() para que el subject que reciba el mensaje de notificación sepa qué evento ha ocurrido de los que observa.

Implementación de Java del patrón Observer

Posee una Interfaz java.util.Observer: una clase puede implementar la interfaz Observer cuando dicha clase quiera ser informada de los cambios que se produzcan en los objetos observados. Tiene un servicio que es el siguiente: void update (Observable o, Object arg)
Este servicio es llamado cuando el objeto observado es modificado.
Además Java nos ofrece los siguientes servicios:
void addObserver (Observer o)
protected void clearChanged()
int countObservers()
void deleteObserver (Observer o)
void deleteObservers()
boolean hasChanged()
void notifyObservers()
void notifyObservers (Object arg)
protected void setChanged()

Posee una clase llamada java.util.Observable: esta clase representa un objeto Subject.
Veamos el mismo ejemplo con el estandard de Java:

```java
import java.util.Observable;
import java.util.Observer;

public class Administracion implements Observer{

    public void update(Observable arg0, Object arg1) {
        System.out.println(arg1);
        System.out.print("Administracion: ");
        System.out.println("Envio una queja formal...");
    }
}
```

```java
import java.util.Observer;
import java.util.Observable;

public class Compras implements Observer {

    public void update(Observable arg0, Object arg1) {
        System.out.println(arg1);
        System.out.print("Compras: ");
        System.out.println("Solicito nueva cotizacion...");
    }

}
```

```java
import java.util.Observer;
import java.util.Observable;

public class Stock implements Observer {
    public void update(Observable arg0, Object arg1) {
        System.out.println(arg1);
        System.out.print("Stock: ");
        System.out.println("Le doy de baja...");
    }
}
```

```java
import java.util.Observable;

public class AlarmaLibro extends Observable {

    public void disparaAlarma(Libro libro) {
        setChanged();
        notifyObservers("Rompieron el libro: "+libro.getTiulo());
    }
}
```

```java
public class Biblioteca {

    public void devuelveLibro(Libro libro) {
        if (libro.getEstado().equals("MALO")) {
            AlarmaLibro a = new AlarmaLibro();
            a.addObserver(new Compras());
            a.addObserver(new Administracion());
            a.addObserver(new Stock());
            a.disparaAlarma(libro);
        }
    }
}
```

```
    public static void main(String[] args) {
        Libro libro = new Libro();
        libro.setTiulo("Windows es estable");
        libro.setEstado("MALO");

        Biblioteca b = new Biblioteca();
        b.devuelveLibro(libro);

    }

}
```