Observer Pattern

Introducción y nombre

Observer. De Comportamiento. Permite reaccionar a ciertas clases llamadas observadores sobre un evento determinado.

Intención

Este patrón permite a los objetos captar dinámicamente las dependencias entre objetos, de tal forma que un objeto notificará a los observadores cuando cambia su estado, siendo actualizados automáticamente.

También conocido como

Dependents, Publish-Subscribe, Observador.

Motivación

Este patrón 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.

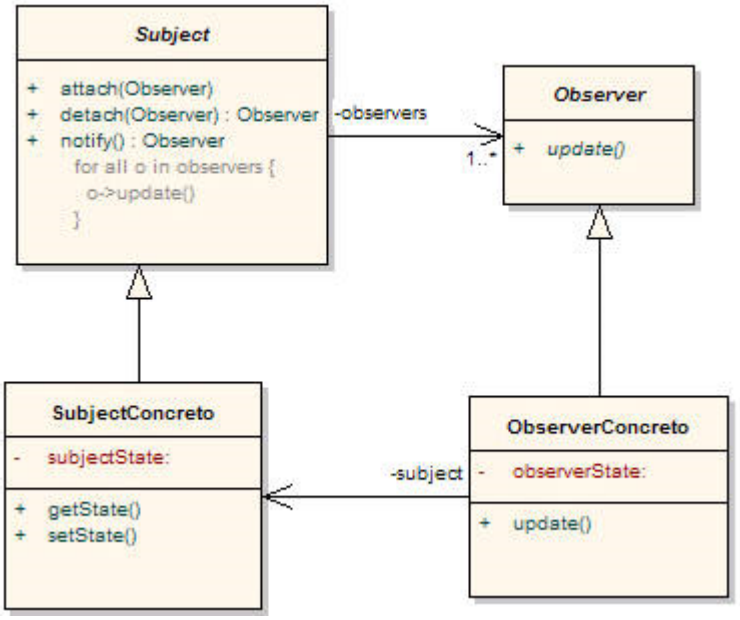
Solución

Este patrón 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.

Diagrama UML



Participantes

Subject: conoce a sus observadores y ofrece la posibilidad de añadir y eliminar observadores.

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.

Colaboraciones

El subject posee un método llamado attach() y otro detach() que sirven para agregar o remover observadores en tiempo de ejecución.

El subjectConcreto notifica a sus observadores cada vez que se produce el evento en cuestión. Esto suele realizarlo mediante el recorrido de una Collection.

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 ó 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.

Implementación

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.
Cabe destacar que Java tiene una propuesta para el patrón observer:
Posee una Interfaz java.util.Observer:
public interface 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.
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:
public Class Observable extends Object

Esta clase representa un objeto Subject.
En el código de muestra haremos un ejemplo estándard y otro con la propuesta de Java.

Código de muestra

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:

[code]
```
public interface ILibroMalEstado {
   public void update();
}
```

La interfaz IlibroMalEstado debe ser implementada por todos los observadores:

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

public class Administracion implements ILibroMalEstado {
    @Override
    public void update() {
        System.out.println("Administracion: ");
        System.out.println("Envio una queja formal...");
    }
}

public class Compras implements ILibroMalEstado {
    @Override
    public void update() {
        System.out.println("Compras: ");
        System.out.println("Solicito nueva cotizacion...");
    }
}
```

Ahora realizaremos la clase que notifica a los observadores:

```
public interface Subject {
    public void attach(ILibroMalEstado observador);

    public void dettach(ILibroMalEstado observador);

    public void notifyObservers();
}

public class AlarmaLibro implements Subject {
    private List<ILibroMalEstado> observadores = new ArrayList<ILibroMalEstado>();

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

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

    @Override
    public void notifyObservers() {
        for (ILibroMalEstado observador : observadores) {
            observador.update();
        }
    }
}
```

Y, por último, la clase que avisa que ocurrió un evento determinado:

```
/*
 * Un libro seguramente tendrá más atributos
 * como autor, editorial, etc pero para nuestro
 * ejemplo no son necesarios
 */
public class Libro {
    private String tiulo;
    private String estado;
```

```
    public Libro() {
    }

    public String getTiulo() {
        return tiulo;
    }

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

    public String getEstado() {
        return estado;
    }

    public void setEstado(String estado) {
        this.estado = estado;
    }
}

public class Biblioteca {
    private AlarmaLibro alarma;

    public Biblioteca() {
    }

    public AlarmaLibro getAlarma() {
        return alarma;
    }

    public void setAlarma(AlarmaLibro alarma) {
        this.alarma = alarma;
    }

    public void devuelveLibro(Libro libro) {
        if (libro.getEstado().equals("MALO")) {
            if (alarma != null) {
                alarma.notifyObservers();
            }
        }
    }
}
```

Veamos como funciona este ejemplo:

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

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

        Biblioteca biblioteca = new Biblioteca();
        biblioteca.setAlarma(alarmas);
        biblioteca.devuelveLibro(libro);
    }
}
```

La salida por consola es:

```
Compras:
Solicito nueva cotizacion...
Administracion:
Envio una queja formal...
Stock:
```

Le doy de baja...

Aquí vemos el mismo ejemplo, pero con el API de Java:

```
public class Administracion implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println(arg);
        System.out.print("Administracion: ");
        System.out.println("Envio una queja formal...");
    }
}

public class Compras implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println(arg);
        System.out.print("Compras: ");
        System.out.println("Solicito nueva cotizacion...");
    }
}

public class Stock implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println(arg);
        System.out.print("Stock: ");
        System.out.println("Le doy de baja...");
    }
}

/*
 * Un libro seguramente tendrá más atributos
 * como autor, editorial, etc pero para nuestro
 * ejemplo no son necesarios
 */
public class Libro {
    private String tiulo;
    private String estado;

    public Libro() {
    }

    public String getTiulo() {
        return tiulo;
    }

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

    public String getEstado() {
        return estado;
    }

    public void setEstado(String estado) {
        this.estado = estado;
    }
}

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

```
public class Biblioteca {
    private AlarmaLibro alarma;

    public Biblioteca() {
    }

    public AlarmaLibro getAlarma() {
        return alarma;
    }

    public void setAlarma(AlarmaLibro alarma) {
        this.alarma = alarma;
    }

    public void devuelveLibro(Libro libro) {
        if (libro.getEstado().equals("MALO")) {
            if (alarma != null) {
                alarma.disparaAlarma(libro);
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        AlarmaLibro alarmas = new AlarmaLibro();
        alarmas.addObserver(new Compras());
        alarmas.addObserver(new Administracion());
        alarmas.addObserver(new Stock());

        Libro libro = new Libro();
        libro.setTiulo("Windows es estable");
        libro.setEstado("MALO");

        Biblioteca biblioteca = new Biblioteca();
        biblioteca.setAlarma(alarmas);
        biblioteca.devuelveLibro(libro);
    }
}

La salida por consola es:
Rompieron el libro: Windows es estable
Stock: Le doy de baja...
Rompieron el libro: Windows es estable
Administracion: Envio una queja formal...
Rompieron el libro: Windows es estable
Compras: Solicito nueva cotizacion…
[/code]
```

Cuándo utilizarlo

Este patrón tiene un uso muy concreto: varios objetos necesitan ser notificados de un evento y cada uno de ellos deciden como 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.

Patrones relacionados

Se pueden encapsular semánticas complejas entre subjects y observers mediante el patrón Mediator.
Dicha encapsulación podría ser única y globalmente accesible si se usa el patrón Singleton.
Singleton: el Subject suele ser un singleton.