Unidad 5: Patrones de Comportamiento

Introducción y orientaciones para el estudio

En el desarrollo de este módulo abordaremos:

- Concepto de patrones de comportamiento.
- ¿Cómo reconocer su necesidad?
- State, Strategy.
- Mediator, Observer.
- Iterator, Command.

Objetivos

Pretendemos que al finalizar de estudiar esta unidad el alumno logre:

- Comprender qué es y cómo se aplica un patrón de comportamiento.
- Definir el ámbito de aplicación.
- Poder utilizar los patrones definidos en la unidad.

Aclaraciones previas al estudio

En este módulo, el alumno encontrará:

- Contenidos
- Conceptualizaciones centrales

Usted debe tener presente que los contenidos presentados en el módulo no ahondan profundamente en el tema, sino que pretenden ser un recurso motivador para que, a través de la lectura del material, la bibliografía sugerida y el desarrollo de las actividades propuestas alcance los objetivos planteados.

Cada módulo constituye una guía cuya finalidad es facilitar su aprendizaje.

Concepto de patrones de comportamiento.

Se definen como patrones de diseño software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

State (Estado) (Tiempo estimado: 3 hs)

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. En determinadas ocasiones, cuando el contexto en el que se está desarrollando requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra, resulta complicado poder manejar el cambio de comportamientos y los estados de dicho

objeto, todos dentro del mismo bloque de código. El patrón State propone una solución a esta complicación, creando básicamente, un objeto por cada estado posible del objeto que lo llama.

El problema / motivación

El patrón State está motivado por aquellos objetos en que, según su estado actual, varía su comportamiento ante los diferentes mensajes. Como ejemplo se toma una clase TCPConnection que representa una conexión de red, un objeto de esta clase tendrá diferentes respuestas según su estado (Listening, Closed o Established). Por ejemplo la invocación al método Open de un objeto de la clase TCPConnection diferirá su comportamiento si la conexión se encuentra en Closed o en Established.

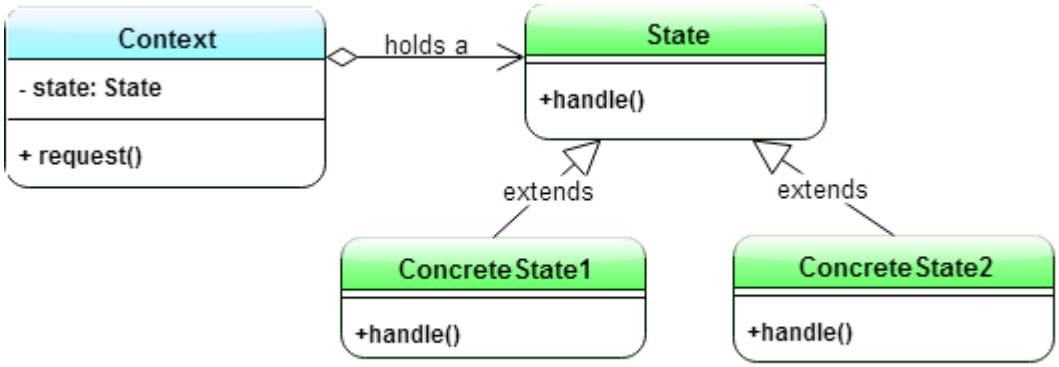
Existe una extrema complejidad en el código cuando se intenta administrar comportamientos diferentes según una cantidad de estados diferentes. Asimismo el mantenimiento de este código se torna dificultoso, e incluso se puede llegar en algunos casos puntuales a la incongruencia de estados actuales por la forma de implementación de los diferentes estados en el código (por ejemplo con variables para cada estado).

Se debe contemplar la complejidad comparada con otras soluciones.

Solución

Se implementa una clase para cada estado diferente del objeto y el desarrollo de cada método según un estado determinado. El objeto de la clase a la que le pertenecen dichos estados resuelve los distintos comportamientos según su estado, con instancias de dichas clases de estado. Así, siempre tiene presente en un objeto el estado actual y se comunica con éste para resolver sus responsabilidades.

La idea principal en el patrón State es introducir una clase abstracta TCPState que representa los estados de la conexión de red y una interfaz para todas las clases que representan los estados propiamente dichos. Por ejemplo la clase TCPEstablished y la TCPClose implementan responsabilidades particulares para los estados establecido y cerrado respectivamente del objeto TCPConnection. La clase TCPConnection mantiene una instancia de alguna subclase de TCPState con el atributo state representando el estado actual de la conexión. En la implementación de los métodos de TCPConnection habrá llamadas a estos objetos representados por el atributo state para la ejecución de las responsabilidades, así según el estado en que se encuentre, enviará estas llamadas a un objeto u otro de las subclases de TCPState.



Participantes

- **Context(Contexto):** Este integrante define la interfaz con el cliente. Mantiene una instancia de ConcreteState (Estado Concreto) que define su estado actual.
- **State (Estado):**Define una interfaz para el encapsulamiento de la responsabilidades asociadas con un estado particular de Context.
- **Subclase ConcreteState:**Cada una de estas subclases implementa el comportamiento o responsabilidad de Context.

El Contexto (Context) delega el estado específico al objeto ConcreteState actual Un objeto Context puede pasarse a sí mismo como parámetro hacia un objeto State. De esta manera la clase State puede acceder al contexto si fuese necesario. Context es la interfaz principal para el cliente. El cliente puede configurar un contexto con los objetos State. Una vez hecho esto, los clientes no tendrán que tratar con los objetos State directamente. Tanto el objeto Context cómo los objetos de ConcreteState pueden decidir el cambio de estado.

Cuando emplearlo

Está apuntado a cuando un determinado objeto tiene diferentes estados y también distintas responsabilidades según el estado en que se encuentre en determinado instante. También puede utilizarse para simplificar casos en los que se tiene un complicado y extenso código de decisión que depende del estado del objeto

Ventajas y Desventajas

Se encuentran las siguientes ventajas:

- Se localizan fácilmente las responsabilidades de los estados específicos, dado que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior. Esta facilidad la brinda el hecho que los diferentes estados están representados por un único atributo (state) y no envueltos en diferentes variables y grandes condicionales.
- Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables, mientras que aquí al estar representado cada estado.

- Los objetos State pueden ser compartidos si no contienen variables de instancia, esto se puede lograr si el estado que representan esta enteramente codificado en su tipo. Cuando se hace esto estos estados son Flyweights sin estado intrínseco.
- Facilita la ampliación de estados
- Permite a un objeto cambiar de clase en tiempo de ejecución dado que al cambiar sus responsabilidades por las de otro objeto de otra clase la herencia y responsabilidades del primero han cambiado por las del segundo.

Se encuentran la siguiente desventaja:

- Se incrementa el número de subclases.

Implementación

```
public class Manejador
 {
      public static void main( String arg[] )
      {
            try
            {
                  State state = new ConcreteStateA();
                  Context context = new Context();
                  context.setState( state );
                  context.request();
            }
            catch( Exception e )
            {
                  e.printStackTrace();
            }
      }
 }

public class Context
 {
      private State state;

      public void setState( State state )
      {
            this.state = state;
      }

      public State getState()
      {
            return state;
      }

      public void request()
      {
            state.handle();
```

```
        }
  }

public interface State
  {
        void handle();
  }

public class ConcreteStateA implements State
  {
        public void handle()
        {
        }
  }

 public class ConcreteStateB implements State
  {
        public void handle()
        {
        }
  }

/**
  * Tenemos una clase específica (Contexto) que administra los cambios de estado de una clase externa mediante la creación de diferentes instancias, dependiendo de el estado que desea adoptar.
  * Cada clase que creas implementa una interfaz (Estado) que define el nombre del método que deben implementar.
  *
  */

public class StatePattern {
        public void main(String args[]){
              try{
                    State state;
                    Context context = new Context();
                    SocketChannel socketChannel = null;
                    //-----------------------------\\
                    //     OPEN/LISTENING SOCKET   \\
                    //-----------------------------\\
                    //First State:
                    state = new ConnectSocketState(socketChannel);
                    context.setState( state );
                    socketChannel = context.request();
                    //-----------------------------\\
                    //           CLOSE SOCKET      \\
                    //-----------------------------\\
                    //Second State:
                    state = new CloseSocketState(socketChannel);
```

```
                    context.setState( state );
                    socketChannel = context.request();
              }catch( Exception e ) {
                    e.printStackTrace();
              }
        }

        public class Context
        {
              private State state;

              public void setState( State state )
              {
                    this.state = state;
              }

              public State getState()
              {
                    return state;
              }

              public SocketChannel request()
              {
                    return state.processState();
              }
        }

        public interface State
        {
              SocketChannel processState();
        }

        public class ConnectSocketState implements State
        {
               SocketChannel socketChannel;
              public ConnectSocketState(SocketChannel socketChannel){
                    this.socketChannel=socketChannel;
              }
              public SocketChannel processState()
              {
                    try {
                          int port = 21;
                          InetAddress                host                =
InetAddress.getByName("192.168.1.1");
                          SocketAddress         adress         =         new
InetSocketAddress(host, port);
                          socketChannel = SocketChannel.open(adress);
                          socketChannel.configureBlocking(true);
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return socketChannel;
            }
        }

        public class CloseSocketState implements State
        {
            SocketChannel socketChannel;
            public CloseSocketState(SocketChannel socketChannel){
                 this.socketChannel=socketChannel;
            }
            public SocketChannel processState(){
                try {
                        socketChannel.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return socketChannel;
            }
        }


}
```

Strategy (Estrategia) (Tiempo estimado: 3 hs)

El patrón Estrategia (Strategy) es un patrón de diseño para el desarrollo de software. Se clasifica como patrón de comportamiento porque determina cómo se debe realizar el intercambio de mensajes entre diferentes objetos para resolver una tarea. El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

Motivación

Suponiendo un editor de textos con diferentes algoritmos para particionar un texto en líneas (justificado, alineado a la izquierda, etc.), se desea separar las clases clientes de los diferentes algoritmos de partición, por diversos motivos:

- Incluir el código de los algoritmos en los clientes hace que estos sean demasiado grandes y complicados de mantener y/o extender.
- El cliente no va a necesitar todos los algoritmos en todos los casos, de modo que no queremos que dicho cliente los almacene si no los va a usar.
- Si existiesen clientes distintos que usasen los mismos algoritmos, habría que duplicar el código, por tanto, esta situación no favorece la reutilización.

La solución que el patrón estrategia supone para este escenario pasa por encapsular los distintos algoritmos en una jerarquía y que el cliente trabaje contra un objeto intermediario

contexto. El cliente puede elegir el algoritmo que prefiera de entre los disponibles, o el mismo contexto puede ser el que elija el más apropiado para cada situación.

Aplicabilidad

Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón estrategia. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución. Si muchas clases relacionadas se diferencian únicamente por su comportamiento, se crea una superclase que almacene el comportamiento común y que hará de interfaz hacia las clases concretas.

Si un algoritmo utiliza información que no deberían conocer los clientes, la utilización del patrón estrategia evita la exposición de dichas estructuras. Aplicando el patrón a una clase que defina múltiples comportamientos mediante instrucciones condicionales, se evita emplear estas instrucciones, moviendo el código a clases independientes donde se almacenará cada estrategia.

Efectivamente, como se comentó anteriormente, este patrón de diseño nos sirve para intercambiar un sin número de estrategias posibles.

Participantes

Contexto (*Context*) : Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria. Puede definir una interfaz que permita a la estrategia el acceso a sus datos en caso de que fuese necesario el intercambio de información entre el contexto y la estrategia. En caso de no definir dicha interfaz, el contexto podría pasarse a sí mismo a la estrategia como parámetro.

Estrategia (*Strategy*): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.

EstrategiaConcreta (*ConcreteStrategy*): Implementa el algoritmo utilizando la interfaz definida por la estrategia.

Colaboraciones

Es necesario el intercambio de información entre estrategia y contexto. Este intercambio puede realizarse de dos maneras:

- Mediante parámetros en los métodos de la estrategia.
- Mandando se, el contexto, a sí mismo a la estrategia.

Los clientes del contexto lo configuran con una estrategia concreta. A partir de ahí, solo se interactúa con el contexto.

Consecuencias

La herencia puede ayudar a factorizar las partes comunes de las familias de algoritmos (sustituyendo el uso de bloques de instrucciones condicionales). En este contexto es común la aparición conjunta de otros patrones como el patrón plantilla.

El uso del patrón proporciona una alternativa a la extensión de contextos, ya que puede realizarse un cambio dinámico de estrategia.

Los clientes deben conocer las diferentes estrategias y debe comprender las posibilidades que ofrecen.

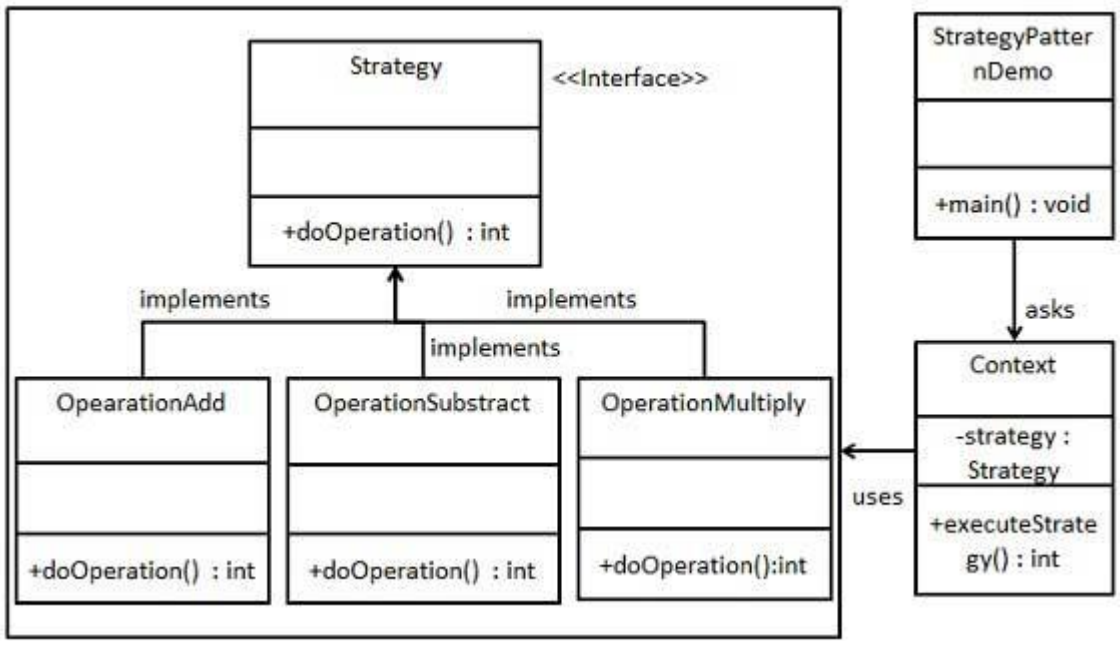
Como contrapartida, aumenta el número de objetos creados, por lo que se produce una penalización en la comunicación entre estrategia y contexto (hay una indirección adicional).

Implementación

Entre las posibilidades disponibles a la hora de definir la interfaz entre estrategia y contexto, están:

- Pasar como parámetro la información necesaria para la estrategia implica un bajo acoplamiento y la posibilidad de envío de datos innecesarios.
- Pasar como parámetro el contexto y dejar que la estrategia solicite la información que necesita supone un alto acoplamiento entre ellos.
- Mantener en la estrategia una referencia al contexto (similar al anterior).

También puede ocurrir que se creen y se utilicen los objetos estrategia en el contexto solo si es necesario, en tal caso las estrategias serán opcionales.



Vamos a crear la interfaz Strategy en donde definiremos una acción y clases concretas de estrategia que la implementen. Context es una clase que usa a Strategy.

StrategyPatternDemo (Manejador), nuestra clase de demostración, utilizará objetos de contexto y estrategia para demostrar el cambio en el comportamiento de contexto basado en la estrategia que implementa o usa.

Step 1

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

Step 2

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Step 3

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Step 4

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

Step 5

Verificamos el resultado de la ejecución.

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

Observer (Observador) (Tiempo estimado: 3 hs)

Observador (en inglés: Observer) es un patrón de diseño de *software* que define una dependencia del tipo *uno a muchos* entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes. Se trata de un *patrón de comportamiento* (existen de tres tipos: creación, estructurales y de comportamiento), por lo que está relacionado con algoritmos de funcionamiento y asignación de *responsabilidades* a clases y objetos.

Los patrones de comportamiento describen no solamente estructuras de relación entre objetos o clases sino también *esquemas de comunicación* entre ellos y se pueden clasificar en función de que trabajen con clases (método plantilla) u objetos (cadena de responsabilidad, comando, iterador, recuerdo, observador, estado, estrategia, visitante).

La variación de la encapsulación es la base de muchos patrones de comportamiento, por lo que cuando un aspecto de un programa cambia frecuentemente, estos patrones definen un objeto que encapsula dicho aspecto. Los patrones definen una clase abstracta que describe la encapsulación del objeto.

Este patrón también se conoce como el patrón de publicación-inscripción o modelo-patrón. Estos nombres sugieren las ideas básicas del patrón, que son: el objeto de datos, que se le puede llamar Sujeto a partir de ahora, contiene atributos mediante los cuales cualquier objeto observador o vista se puede suscribir a él pasándole una *referencia* a sí mismo. El Sujeto mantiene así una lista de las referencias a sus observadores. Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el Sujeto es capaz de notificar a sus observadores suscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el Sujeto, ejecutado, por ejemplo, por alguno de los observadores, el objeto de datos puede recorrer la lista de observadores avisando a cada uno. Este patrón suele utilizarse en los entornos de trabajo de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir *listeners* a los objetos que pueden disparar eventos.

El patrón observador es la clave del patrón de arquitectura Modelo Vista Controlador (MVC).[1] De hecho el patrón fue implementado por primera vez en el MVC de Smalltalk basado en un entorno de trabajo de interfaz.[2] Este patrón está implementado en numerosos bibliotecas y sistemas, incluyendo todos los *toolkits* de GUI.

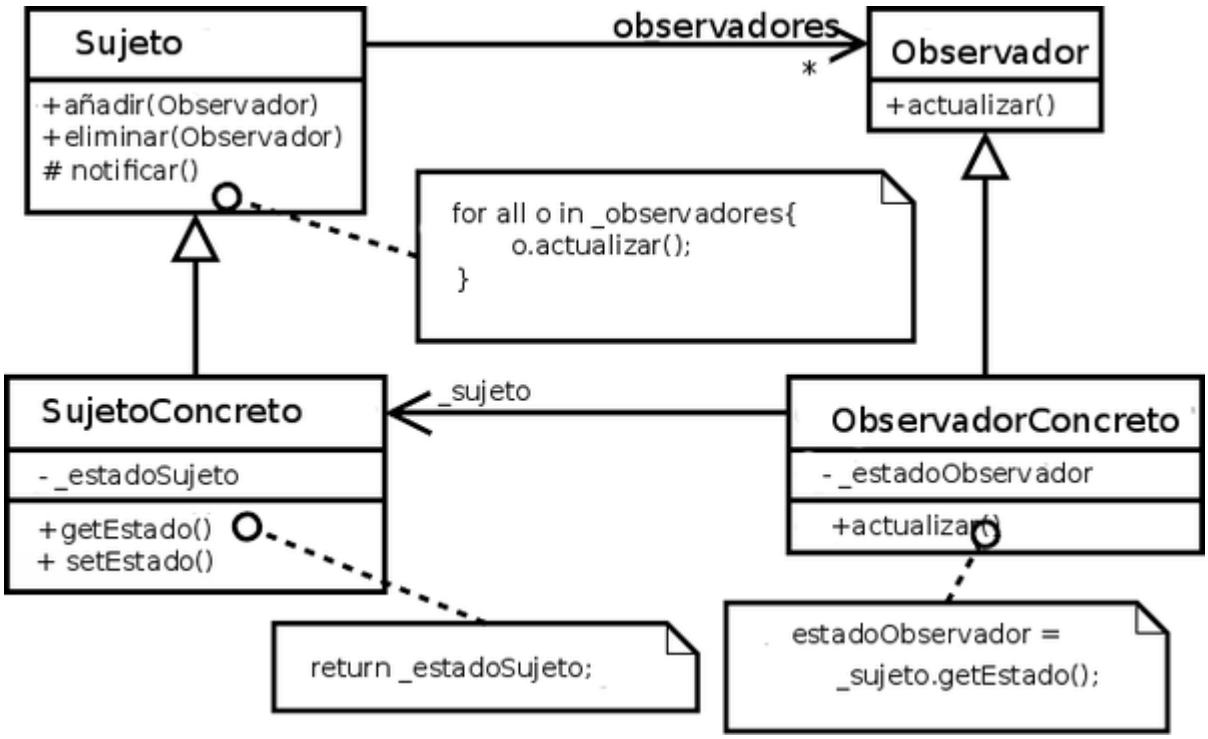
Objetivo

Definir una dependencia uno a muchos entre objetos, de tal forma que cuando el objeto cambie de estado, todos sus objetos dependientes sean notificados automáticamente. Se trata de desacoplar la clase de los objetos clientes del objeto, aumentando la modularidad del lenguaje,

creando las mínimas dependencias y evitando bucles de actualización (espera activa o sondeo). En definitiva, normalmente, se usará el patrón observador cuando un elemento *quiere* estar pendiente de otro, sin tener que estar comprobando de forma continua si ha cambiado o no.

Motivación
Si se necesita consistencia entre clases relacionadas, pero con independencia, es decir, con un bajo acoplamiento.

Ámbito de Aplicación
Puede pensarse en aplicar este patrón cuando una modificación en el estado de un objeto requiere cambios de otros, y no se desea que se conozca el número de objetos que deben ser cambiados. También cuando se quiere que un objeto sea capaz de notificar a otros objetos sin hacer ninguna suposición acerca de los objetos notificados y cuando una abstracción tiene dos aspectos diferentes, que dependen uno del otro; si se encapsulan estos aspectos en objetos separados se permitirá su variación y reutilización de modo independiente.



Participantes

Habrá sujetos concretos cuyos cambios pueden resultar interesantes a otros y observadores a los que al menos les interesa estar pendientes de un elemento y en un momento dado, reaccionar ante sus notificaciones de cambio. Todos los sujetos tienen en común que un conjunto de objetos quieren estar pendientes de ellos. Cualquier elemento que quiera ser observado tiene que permitir indicar:

1. “Estoy interesado en tus cambios”.
2. “Ya no estoy interesado en tus cambios”.

El observable tiene que tener, además, un mecanismo de aviso a los interesados.

A continuación se detallan a los participantes de forma desglosada:

- **Sujeto (*subject*):**

El sujeto proporciona una interfaz para agregar (*attach*) y eliminar (*detach*) observadores. El Sujeto conoce a todos sus observadores.

- **Observador (*observer*):**

Define el método que usa el sujeto para notificar cambios en su estado (*update*/*notify*).

- **Sujeto concreto (*concrete subject*):**

Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.

- **Observador concreto (*concrete observer*):**

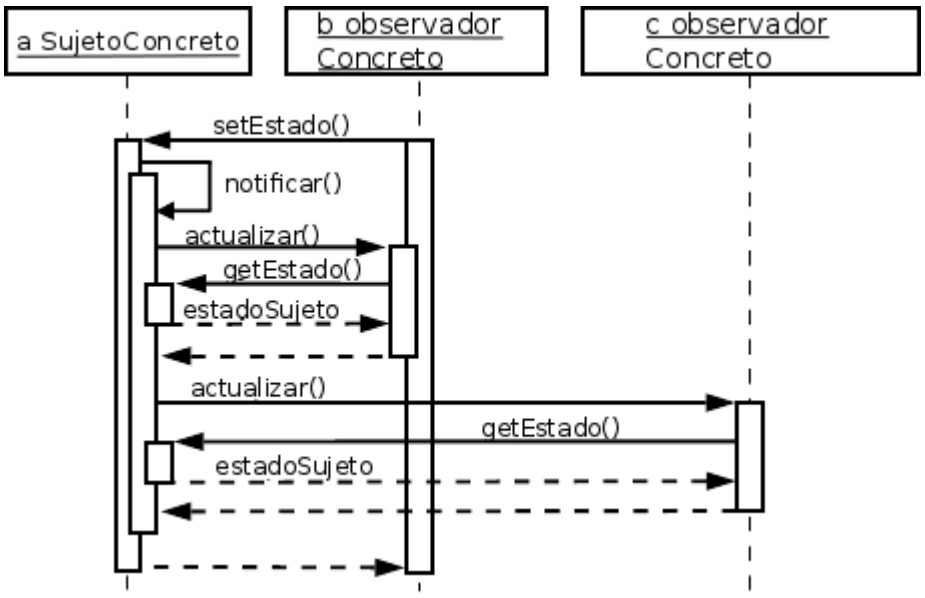
Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

Colaboración

La colaboración más importante en este patrón es entre el sujeto y sus observadores, ya que en el momento en el que el sujeto sufre un cambio, este notifica a sus observadores.

Consecuencias

Las consecuencias de aplicar este patrón pueden ser tanto beneficiosas como pueden perjudicar algunos aspectos. Por una parte abstrae el acoplamiento entre el sujeto y el observador, lo cual es beneficioso ya que se consigue una mayor independencia y además el sujeto no necesita especificar los observadores afectados por un cambio. Por otro lado, con el uso de este patrón ocurre que se van a desconocer las consecuencias de una actualización, lo cual, dependiendo del problema, puede afectar en mayor o menor medida (por ejemplo, al rendimiento).



Implementación

A continuación se detallan una serie de problemas que se pueden presentar a la hora de implementar este patrón:

- **Problema 1:**

Para evitar que el observador concreto tenga una asociación con el sujeto concreto, se podría hacer que la relación entre sujeto y observador fuese bidireccional, evitando así asociaciones concretas, el problema es que dejaría de ser una interfaz. El que deje de ser una interfaz puede producir problemas si el lenguaje de programación no soporta la herencia múltiple.

Se podría eliminar la bidireccionalidad de la asociación pasando el sujeto como parámetro al método actualizar y ya no se tendría que referenciar el objeto observado. Esto podría causar problemas si se observan varios objetos, tanto de la misma clase como de distintas, ya que no elimina dependencias, y para hacer operaciones específicas sobre el objeto actualizado obliga a hacer en la implementación.

- **Problema 2:**

Si hay muchos sujetos sin observador, la estructura de los observadores está desaprovechada, para solucionarlo se puede tener un intermediario que centralice el almacenamiento de la asociación de cada sujeto con sus observadores. Para esta solución se crea ese gestor de observadores usando el patrón *singleton* (instancia única), ya que proporciona una única referencia y no una por cada sujeto. El gestor aunque mejora el aprovechamiento del espacio, hace que se reduzca el rendimiento y se pierde eficiencia en el método notificar.

- **Problema 3:**

El responsable de iniciar la comunicación es el sujeto concreto, pero se puede dar un problema cuando el objeto concreto está siendo actualizado de forma continua ya que debido a esto se tendría que realizar muchas actualizaciones en muy poco tiempo. La solución sería suspender temporalmente las llamadas al método de actualización/notificación; por ejemplo, haciendo que el cliente pueda activar o desactivar las notificaciones y notificar todos los cambios cuando las vuelva a habilitar. El patrón Estado sería una posible solución para diseñar esta variante de no notificar si no se han dado cambios o hacerlo en caso de que si.

- **Problema 4 (referencias inválidas):**

A la hora de implementar este patrón se debe tener cuidado cuando un elemento observable desaparece. En ciertos lenguajes será el gestor de memoria el que cada cierto tiempo debe de limpiar las referencias liberadas, pero si un observable que sigue siendo observado puede no liberarse nunca. Para solucionar este problema puede crearse una función *destruir* que notifique al gestor que el elemento observable va a desaparecer y si no se está usando la variante del gestor el observable directamente desregistrará a sus observadores. Antes de esto hay que eliminar las referencias a este elemento, por tanto, hay que eliminar a los observadores antes de eliminar al observable, ya que así se evitará tanto que aparezcan referencias inválidas al objeto una vez éste haya sido eliminado, como que se produzcan operaciones inválidas intentando invocarlo.

Se puede avisar a los observadores creando un método actualizar especial, en el que se tendrían dos opciones:

1. El observador también muere.

2. El observador sigue vivo, pero apunta a nulo.

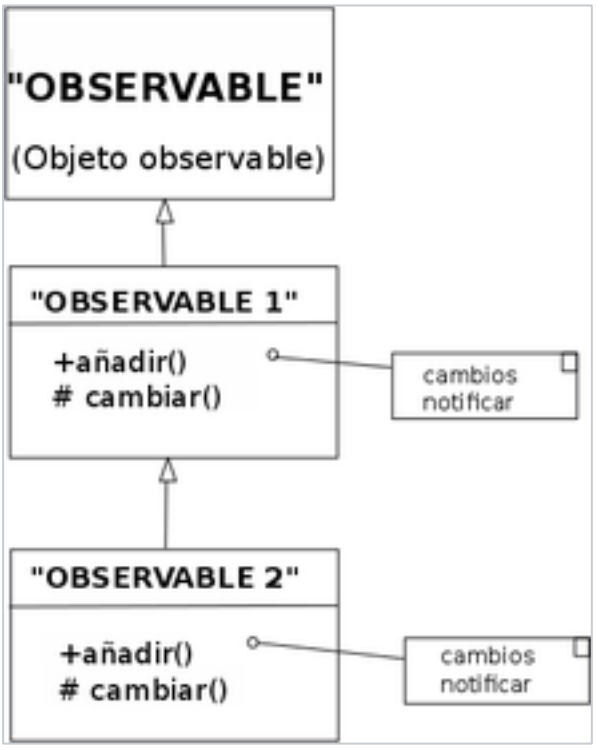
- **Problema 5:**

Ya que se debe asegurar la consistencia del estado del sujeto antes de iniciar una notificación, siempre se notificará al final, ya que aunque en entorno multihilo se notifica antes de hacer los cambios, puede que los observadores soliciten información al observable cuando aún se van a hacer más cambios y se darían problemas de consistencia si se accede a un estado que aún no es el definitivo. De esta forma, los observadores ya no accederán a sujetos en estado inconsistente.

Por ejemplo:

```
Secuencia incorrecta:
                        a
                b
                c
                notificar()
                d
                e
                f
Secuencia correcta:
                      a
                b
                c
                d
                e
                f
                notificar()
```

Jerarquía con varios tipos des observadores: en este caso el hilo redefine cambios, no los notifica.



Jerarquía de varios observadores

- **Problema 6:**

En mecanismos de notificación tradicionalmente hay dos opciones: *pull* que es la que propone el patrón observador; y *push* que es la que se tendría si se incluye información como parámetros en el mecanismo de actualización. El problema de hacer esto es que la interfaz del observador se vuelve más específica y por tanto menos genérica y reutilizable.

PULL: los objetos avisan de que han cambiado y el observador pregunta cuál ha sido el cambio.

PUSH: minimiza (eficiencia) que cuando algo cambia y se informará a todos los interesados, se realicen el menor número de llamadas posibles.

Dependiendo del problema que haya que resolver, se habrá de valorar que implementación se ajusta mejor para resolverlo de la forma más eficiente y efectiva o si las variantes anteriores pueden combinarse entre sí dependiendo de las características de escenario concreto. Por ejemplo, la opción 2 podría aplicarse cuando interese aplicar en un sujeto concreto n métodos seguidos y no se quiere notificar hasta que todos finalicen su ejecución.

Ejemplo

Los siguientes ejemplos muestran un programa que lee del teclado, y cada línea se tratará como un evento. Cuando una línea es obtenida, se llama al método que notificara a los observadores, tal que todos los observadores fueran avisados de la ocurrencia del evento mediante sus métodos de actualización.

Ejemplo 1 en Java

En el caso de Java, el método de actualización sería notifyObservers.

```
import java.util.*;
class FuenteEvento extends Observable implements Runnable {
    public void run() {
        while (true) {
            String respuesta = new Scanner(System.in).next();
            setChanged();
            notifyObservers(respuesta);
        }
    }
}

import java.util.Observer;
class MiApp extends Observable {
    public static void main(String[] args) {
        MiApp miApp = new MiApp();
        System.out.println("Introducir Texto >");
        FuenteEvento fuenteEvento = new FuenteEvento();
        fuenteEvento.addObserver(miApp);
        new Thread(fuenteEvento).start();
    }

    void update(Observable o, Object arg) {
       System.out.println("\nRespuesta recibida: " + arg);
    }

}
```

Mediator (Mediador) (Tiempo estimado: 3 hs)

El **patrón mediador** define un objeto que encapsula como un conjunto de objetos interactúan. Este patrón de diseño está considerado como un **patrón de comportamiento** debido al hecho de que puede alterar el comportamiento del programa en ejecución.

Habitualmente un programa está compuesto de un número de clases (muchas veces elevado). La lógica y computación es distribuida entre esas clases. Sin embargo, cuantas más clases son desarrolladas en un programa, especialmente durante mantenimiento y/o refactorización, el problema de comunicación entre estas clases quizás llegue a ser más complejo. Esto hace que el programa sea más difícil de leer y mantener. Además, puede llegar a ser difícil cambiar el programa, ya que cualquier cambio podría afectar código en muchas otras clases.

Con el **patrón mediador**, la comunicación entre objetos es encapsulada con un objeto **mediador**. Los objetos no se comunican de forma directa entre ellos, en lugar de ello se comunican mediante el mediador. Esto reduce las dependencias entre los objetos en comunicación, reduciendo entonces la Dependencia de código.

Definición

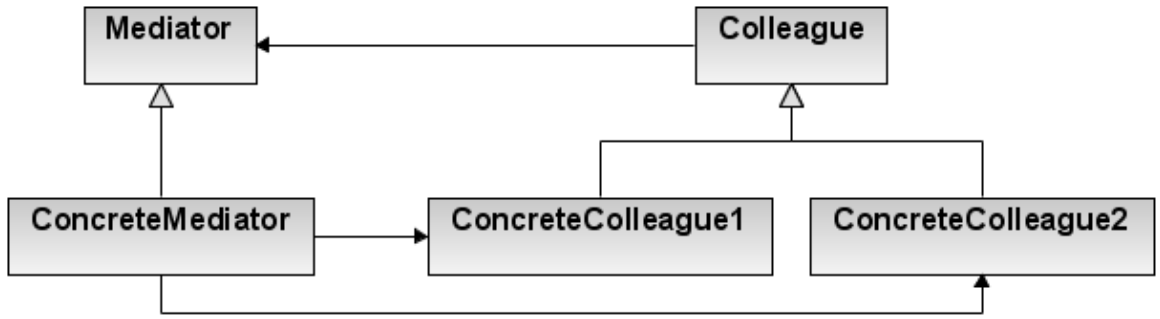
La esencia del Patrón Mediador es "Definir un objeto que encapsula como un conjunto de objetos interactúan. El mediador busca reducir la dependencia evitando que los objetos se relacionen entre ellos de forma explícita, y permitiendo variar cualquier interacción independientemente"

Participantes

Mediador - define la interfaz para la comunicación entre objetos *amigos*

MediadorConcreto - implementa la interfaz Mediador y coordina la comunicación entre objetos *amigos*. Es consciente de todos los *amigos* y su propósito en lo que concierne a la comunicación entre ellos.

AmigoConcreto - se comunica con otros *amigos* a través de su *Mediador*




```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

//Interfaz Amigo
interface Command {
    void execute();
}

//Mediador Abstracto
interface IMediator {
    void book();
    void view();
    void search();
    void registerView(BtnView v);
    void registerSearch(BtnSearch s);
```

```
    void registerBook(BtnBook b);
    void registerDisplay(LblDisplay d);
}

//Mediador Concreto
class Mediator implements IMediator {

    BtnView btnView;
    BtnSearch btnSearch;
    BtnBook btnBook;
    LblDisplay show;

    //....
    void registerView(BtnView v) {
        btnView = v;
    }

    void registerSearch(BtnSearch s) {
        btnSearch = s;
    }

    void registerBook(BtnBook b) {
        btnBook = b;
    }

    void registerDisplay(LblDisplay d) {
        show = d;
    }

    void book() {
        btnBook.setEnabled(false);
        btnView.setEnabled(true);
        btnSearch.setEnabled(true);
        show.setText("booking...");
    }

    void view() {
        btnView.setEnabled(false);
        btnSearch.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("viewing...");
    }

    void search() {
        btnSearch.setEnabled(false);
        btnView.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("searching...");
```

```
    }

}

//Un amigo concreto
class BtnView extends JButton implements Command {

    IMediator med;

    BtnView(ActionListener al, IMediator m) {
        super("View");
        addActionListener(al);
        med = m;
        med.registerView(this);
    }

    public void execute() {
        med.view();
    }

}

//Un amigo concreto
class BtnSearch extends JButton implements Command {

    IMediator med;

    BtnSearch(ActionListener al, IMediator m) {
        super("Search");
        addActionListener(al);
        med = m;
        med.registerSearch(this);
    }

    public void execute() {
        med.search();
    }

}

//Un amigo concreto
class BtnBook extends JButton implements Command {

    IMediator med;

    BtnBook(ActionListener al, IMediator m) {
        super("Book");
        addActionListener(al);
```

```
        med = m;
        med.registerBook(this);
    }

    public void execute() {
        med.book();
    }

}

class LblDisplay extends JLabel {

    IMediator med;

    LblDisplay(IMediator m) {
        super("Just start...");
        med = m;
        med.registerDisplay(this);
        setFont(new Font("Arial", Font.BOLD, 24));
    }

}

class MediatorDemo extends JFrame implements ActionListener {

    IMediator med = new Mediator();

    MediatorDemo() {
        JPanel p = new JPanel();
        p.add(new BtnView(this, med));
        p.add(new BtnBook(this, med));
        p.add(new BtnSearch(this, med));
        getContentPane().add(new LblDisplay(med), "North");
        getContentPane().add(p, "South");
        setSize(400, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent ae) {
        Command comd = (Command) ae.getSource();
        comd.execute();
    }

    public static void main(String[] args) {
        new MediatorDemo();
    }

}
```

Iterator (Iterador)

Trabajo autónomo.

Command (Orden).

Trabajo autónomo..