Builder Pattern

Introducción y nombre

Builder. Creacional.

Permite la creación de una variedad de objetos complejos desde un objeto fuente, el cual se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo.

Intención

Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

También conocido como

Patrón constructor o virtual builder.

Motivación

Los objetos que dependen de un algoritmo tendrán que cambiar cuando el algoritmo cambia. Por lo tanto, los algoritmos que estén expuestos a dicho cambio deberían ser separados, permitiendo de esta manera reutilizar algoritmos para crear diferentes representaciones. En otras palabras, permite a un cliente construir un objeto complejo especificando sólo su tipo y contenido, ocultándole todos los detalles de la construcción del objeto.

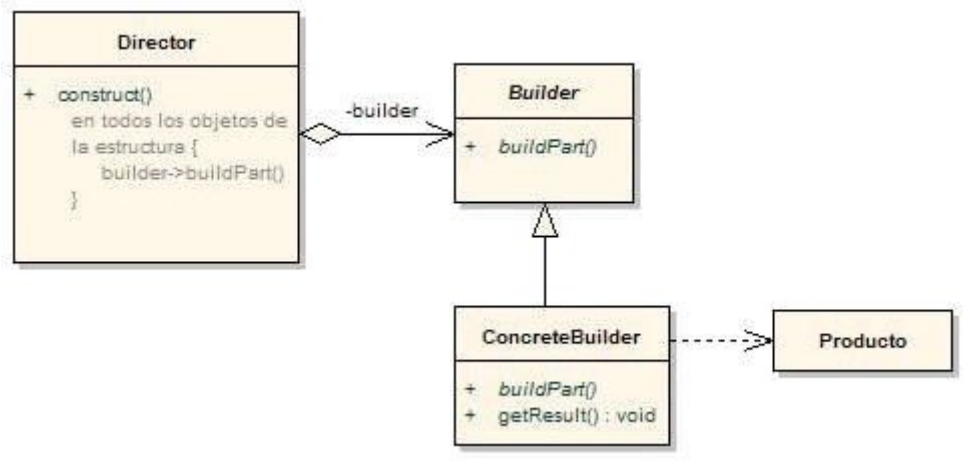
Solución

Se debe utilizar este patrón cuando sea necesario:

Independizar el algoritmo de creación de un objeto complejo de las partes que constituyen el objeto y cómo se ensamblan entre ellas.

Que el proceso de construcción permita distintas representaciones para el objeto construido, de manera dinámica.

Diagrama UML



Participantes

Producto: representa el objeto complejo a construir.

Builder: especifica una interface abstracta para la creación de las partes del Producto. Declara las operaciones necesarias para crear las partes de un objeto concreto.

ConcreteBuilder: implementa Builder y ensambla las partes que constituyen el objeto complejo.

Director: construye un objeto usando la interfaz Builder. Sólo debería ser necesario especificar su tipo y así poder reutilizar el mismo proceso para distintos tipos.

Colaboraciones

El Cliente crea el objeto Director y lo configura con el objeto Builder deseado.

El Director notifica al constructor cuándo una parte del Producto se debe construir.

El Builder maneja los requerimientos desde el Director y agrega partes al producto.

El Cliente recupera el Producto desde el constructor.

Consecuencias

Permite variar la representación interna de un producto.

El Builder ofrece una interfaz al Director para construir un producto y encapsula la representación interna del producto y cómo se juntan sus partes.

Si se cambia la representación interna basta con crear otro Builder que respete la interfaz.

Separa el código de construcción del de representación.

Las clases que definen la representación interna del producto no aparecen en la interfaz del Builder.

Cada ConcreteBuilder contiene el código para crear y juntar una clase específica de producto.

Distintos Directores pueden usar un mismo ConcreteBuilder.

Da mayor control en el proceso de construcción.

Permite que el Director controle la construcción de un producto paso a paso.

Sólo cuando el producto está acabado lo recupera el director del builder.

Implementación

Generalmente un Builder abstracto define las operaciones para construir cada componente que el Director podría solicitar.

El ConcreteBuilder implementa estas operaciones y le otorga la inteligencia necesaria para su creación.

Para utilizarlo el Director recibe un ContreBuilder.

Código de muestra

Realizaremos un ejemplo de un auto, el cual consta de diferentes partes para poder construirse.

[code]

```
public class Motor {
   private Integer numero;
   private String potencia;

   public Motor() {
   }
```

```
    public Integer getNumero() {
        return numero;
    }

    public void setNumero(Integer numero) {
        this.numero = numero;
    }

    public String getPotencia() {
        return potencia;
    }

    public void setPotencia(String potencia) {
        this.potencia = potencia;
    }
}

public class Auto {

    private int cantidadDePuertas;
    private String modelo;
    private String marca;
    private Motor motor;

    public int getCantidadDePuertas() {
        return cantidadDePuertas;
    }

    public void setCantidadDePuertas(int cantidadDePuertas) {
        this.cantidadDePuertas = cantidadDePuertas;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public Motor getMotor() {
        return motor;
    }

    public void setMotor(Motor motor) {
        this.motor = motor;
    }
}
```

Utilizaremos la clase AutoBuilder para que sirve para base de construcción de los distintos tipos de Autos:

```
public abstract class AutoBuilder {
    protected Auto auto = new Auto();

    public Auto getAuto() {
        return auto;
    }

    public void crearAuto() {
```

```
        auto = new Auto();
    }

    public abstract void buildMotor();

    public abstract void buildModelo();

    public abstract void buildMarca();

    public abstract void buildPuertas();
}
```

Realizaremos dos builders concretos que son: FordBuilder y FiatBuilder. Cada Builder tiene el conocimiento necesario para saber como se construye su auto.

```
public class FiatBuilder extends AutoBuilder {

    @Override
    public void buildMarca() {
        auto.setMarca("Fiat");
    }

    @Override
    public void buildModelo() {
        auto.setModelo("Palio");
    }

    @Override
    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(232323);
        motor.setPotencia("23 HP");
        auto.setMotor(motor);
    }

    @Override
    public void buildPuertas() {
        auto.setCantidadDePuertas(2);
    }
}
```

A modo de simplificar el aprendizaje, estos builders construyen objetos relativamente sencillos. Se debe tener en cuenta que la complejidad para la construcción de los objetos suele ser mayor.

```
public class FordBuilder extends AutoBuilder {

    @Override
    public void buildMarca() {
        auto.setMarca("Ford");
    }

    @Override
    public void buildModelo() {
        auto.setModelo("Focus");
    }

    @Override
    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(21212);
        motor.setPotencia("20 HP");
        auto.setMotor(motor);
    }

    @Override
    public void buildPuertas() {
```

```
        auto.setCantidadDePuertas(4);
    }
}
```

Por último, realizaremos la clase Concesionario. Lo primero que debe hacerse con esta clase es enviarle el tipo de auto que se busca construir (Ford, Fiat, etc). Luego, al llamar al método constructAuto(), la construcción se realizará de manera automática.

```
public class Concesionaria {

    private AutoBuilder autoBuilder;

    public void construirAuto() {
        autoBuilder.buildMarca();
        autoBuilder.buildModelo();
        autoBuilder.buildMotor();
        autoBuilder.buildPuertas();
    }

    public void setAutoBuilder(AutoBuilder ab) {
        autoBuilder = ab;
    }

    public Auto getAuto() {
        return autoBuilder.getAuto();
    }
}
```

La invocación desde un cliente sería:

```
public class Main {
    public static void main(String[] args) {
        Concesionaria concesionaria = new Concesionaria();
        concesionaria.setAutoBuilder(new FordBuilder());
        concesionaria.construirAuto();
        Auto auto = concesionaria.getAuto();
        System.out.println(auto.getMarca());
    }
}
```

[/code]

Cuándo utilizarlo

Esta patrón debe utilizarse cuando el algoritmo para crear un objeto suele ser complejo e implica la interacción de otras partes independientes y una coreografía entre ellas para formar el ensamblaje. Por ejemplo: la construcción de un objeto Computadora, se compondrá de otros muchos objetos, como puede ser un objeto PlacaDeSonido, Procesador, PlacaDeVideo, Gabinete, Monitor, etc.

Patrones relacionados

Con el patrón Abstract Factory también se pueden construir objetos complejos, pero el objetivo del patrón Builder es construir paso a paso, en cambio, el énfasis del Abstract Factory es tratar familias de objetos.

El objeto construido con el patrón Builder suele ser un Composite.

El patrón Factory Method se puede utilizar el Builder para decidir qué clase concreta instanciar para construir el tipo de objeto deseado.

El patrón Visitor permite la creación de un objeto complejo, en vez de paso a paso, dando todo de golpe como objeto visitante.

State Pattern

Introducción y nombre

State. De Comportamiento. Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

Intención

Busca que un objeto pueda reaccionar según su estado interno. Si bien muchas veces esto se puede solucionar con un boolean o utilizando constantes, esto suele terminar con una gran cantidad de if-else, código ilegible y dificultad en el mantenimiento. La intención del State es desacoplar el estado de la clase en cuestión.

También conocido como

Objects for State, Patrón de estados.

Motivación

En determinadas ocasiones se requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra. Esto resulta complicado de manejar, sobretodo cuando se debe tener en cuenta el cambio de comportamientos y estados de dicho objeto, todos dentro del mismo bloque de código. El patrón State propone una solución a esta complicación, creando un objeto por cada estado posible.

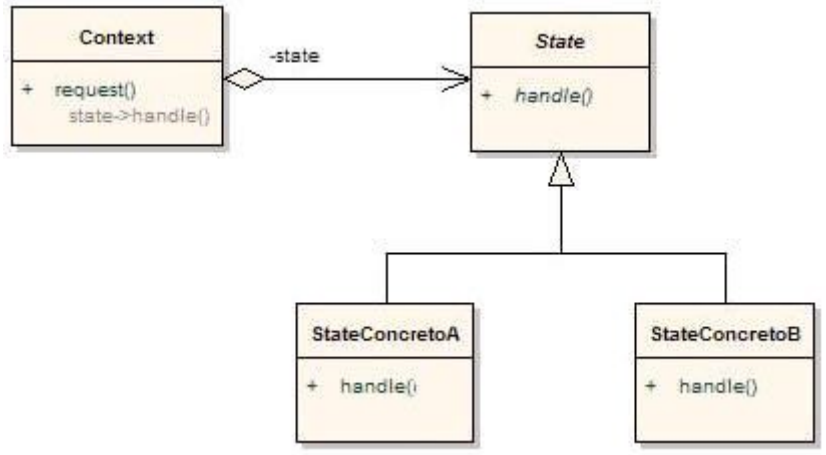
Solución

Este patrón debe ser utilizado cuando:

El comportamiento de un objeto depende de un estado, y debe cambiar en tiempo de ejecución según el comportamiento del estado.

Cuando las operaciones tienen largas sentencias con múltiples ramas que depende del estado del objeto.

Diagrama UML



Participantes

Context: mantiene una instancia con el estado actual

State: define interfaz para el comportamiento asociado a un determinado estado del Contexto.

StateConcreto: cada subclase implementa el comportamiento asociado con un estado del contexto.

Colaboraciones

El Context delega el estado específico al objeto StateConcreto 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 estos clientes no tendrán que tratar con los objetos State directamente. Tanto el objeto Context como los objetos de StateConcreto pueden decidir el cambio de estado.

Consecuencias

Se localizan fácilmente las responsabilidades de los estados concretos, 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.

Facilita la ampliación de estados mediante una simple herencia, sin afectar al Context.

Permite a un objeto cambiar de estado en tiempo de ejecución.

Los estados pueden reutilizarse: varios Context pueden utilizar los mismos estados.

Se incrementa el número de subclases.

Implementación

La clase Context envía mensajes a los estados concretos dentro de su código para brindarle a éstos la responsabilidad que debe cumplir el objeto Context. Así el objeto Context va cambiando las responsabilidades según el estado en que se encuentra. Para llevar a cabo esto se puede utilizar dos tácticas: que el State sea una interfaz o una clase abstracta. Para resolver este problema, siempre se debe intentar utilizar una interfaz, exceptuando aquellos casos donde se necesite repetir un comportamiento en los estados concretos: para ello lo ideal es utilizar una clase abstracta (state) con los métodos repetitivos en código, de manera que los estados concretos lo hereden. En este caso, el resto de los métodos no repetitivos deberían ser métodos abstractos.

Un tipo muy importante a tener en cuenta: el patrón no indica exactamente dónde definir las transiciones de un estado a otro. Existen dos formas de solucionar esto: definiendo estas transiciones dentro de la clase contexto, la otra es definiendo estas transiciones en las subclaes de State. Es más conveniente utilizar la primer solución cuando el criterio a aplicar es fijo, es decir, no se modificará. En cambio la segunda resulta conveniente cuando este criterio es dinámico, el inconveniente aquí se presenta en la dependencia de código entre las subclases.

Código de muestra

Veamos un código muy sencillo de entender:

[code]

```
public interface SaludState {
    public String comoTeSentis();
}

public class DolorDeCabeza implements SaludState {
    @Override
    public String comoTeSentis() {
        return "Todo mal: me duele la cabeza";
    }
}

public class DolorDePanza implements SaludState {
    @Override
    public String comoTeSentis() {
        return "Todo mal: me duele la panza";
    }
}

public class Saludable implements SaludState {
    @Override
    public String comoTeSentis() {
        return "Pipi Cucu!!";
    }
}

public class Persona {
    private String nombre;
    private SaludState salud;

    public Persona() {
        salud = new Saludable();
    }

    public void estoyBien() {
        salud = new Saludable();
    }

    public void dolorDeCabeza() {
        salud = new DolorDeCabeza();
    }

    public void dolorDePanza() {
        salud = new DolorDePanza();
    }

    public String comoTeSentis() {
        return salud.comoTeSentis();
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public SaludState getSalud() {
        return salud;
    }

    public void setSalud(SaludState salud) {
        this.salud = salud;
    }
```

```
}

public class Main {
   public static void main(String[] args) {
      Persona p = new Persona();
      p.setNombre("Juan");
      System.out.println(p.comoTeSentis());

      p.dolorDeCabeza();
      System.out.println(p.comoTeSentis());
   }
}

La salida por consola es:
Pipi Cucu!!
Todo mal: me duele la cabeza
[/code]
```

Cuándo utilizarlo

Este patrón se utiliza 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.

Imaginemos que vamos a un banco y cuando llegamos nos colocamos en la fila de mostrador: si la misma esta abierta, seguiremos en la fila. En cambio, si esta cerrada nos colocaremos en otra fila o tomaremos alguna decisión acorde. Por otro lado, si vemos un cartel que dice "enseguida vuelvo" quizás tenemos que contemplar el tiempo disponible que tenemos. Es decir, para nosotros, el comportamiento de un banco cambia radicalmente según el estado en el que se encuentre. Para estas ocasiones, es ideal el uso de un patrón de estados.

Patrones relacionados

El patrón State puede utilizar el patrón Singleton cuando requiera controlar que una sola instancia de cada estado. Lo puede utilizar cuando se comparten los objetos como Flyweight existiendo una sola instancia de cada estado y esta instancia es compartida con más de un objeto.

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.

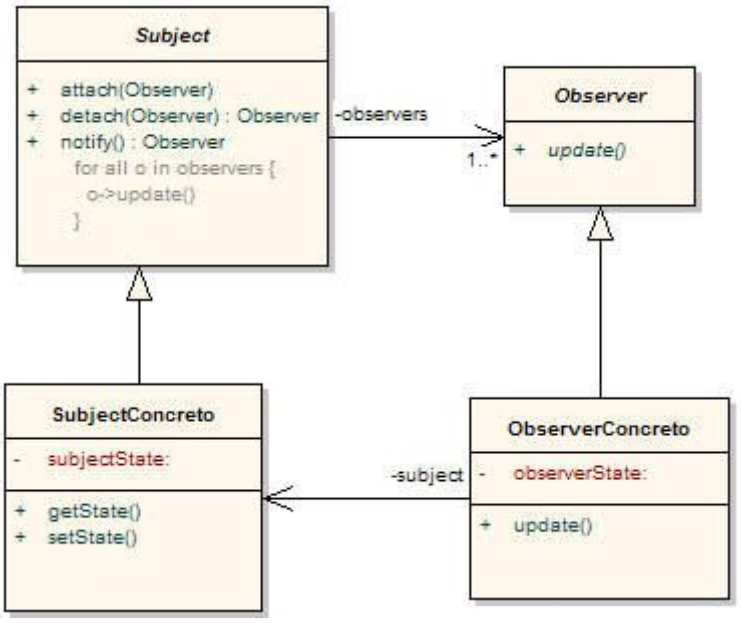
Solución

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.

Mediator Pattern

Introducción y nombre

Mediator. De Comportamiento. Define un objeto que hace de procesador central.

Intención

Un Mediator es un patrón de diseño que coordina las relaciones entre sus asociados o participantes. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones. Todos los objetos se comunican con un mediador y es éste quién realiza la comunicación con el resto.

También conocido como

Mediador, Intermediario.

Motivación

Cuando muchos objetos interactúan con otros objetos, se puede formar una estructura muy compleja, con muchas conexiones entre distintos objetos. En un caso extremo cada objeto puede conocer a todos los demás objetos. Para evitar esto, el patrón Mediator, encapsula el comportamiento de todo un conjunto de objetos en un solo objeto.

Solución

Usar el patrón Mediator cuando:

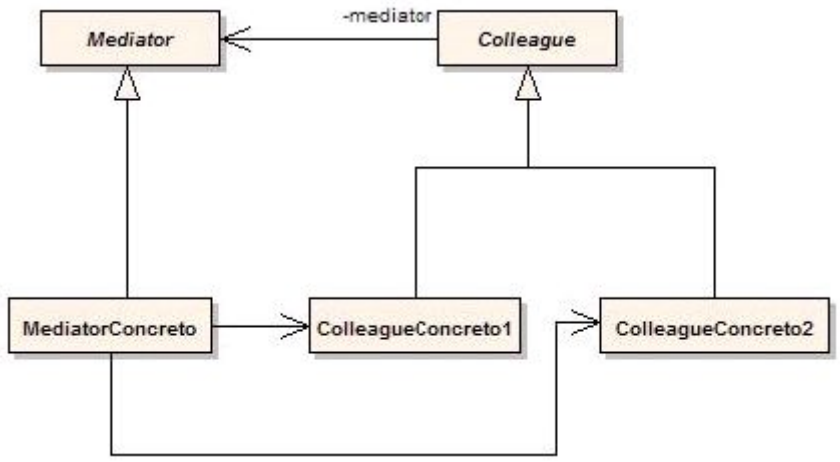
Un conjunto grande de objetos se comunica de una forma bien definida, pero compleja.

Reutilizar un objeto se hace difícil por que se relaciona con muchos objetos.

Las clases son difíciles de reutilizar porque su función básica esta entrelazada con relaciones de dependencia.

Diagrama UML

Estructura en un diagramas de clases.



Participantes

Responsabilidad de cada clase participante.

Mediator: define una interface para comunicarse con los objetos colegas.

MediatorConcreto: implementa la interface y define como los colegas se comunican entre ellos. Además los conoce y mantiene, con lo cual hace de procesador central de todos ellos.

Colleague: define el comportamiento que debe implementar cada colega para poder comunicarse el mediador de una manera estandarizada para todos.

ColleagueConcreto: cada colega conoce su mediador, y lo usa para comunicarse con otros colegas.

Colaboraciones

Los colegas envían y reciben requerimientos de un objeto mediador. El mediador gestiona cada mensaje y se lo comunica a otro colega si fuese necesario.

Consecuencias

Desacopla a los colegas: el patrón Mediator promueve bajar el acoplamiento entre colegas. Se puede variar y reusar colegas y mediadores independiéntemente.

Simplifica la comunicación entre objetos: los objetos que se comunican de la forma "muchos a muchos" puede ser remplazada por una forma "uno a muchos" que es menos compleja y más elegante. Además esta forma de comunicación es más fácil de entender. Es decir, un objeto no necesita conocer a todos los objetos, tan sólo a un mediador.

Clarifica como los objetos se relacionan en un sistema.

Centraliza el control: el mediador es el que se encarga de comunicar a los colegas, este puede ser muy complejo, difícil de entender y modificar. Para que quién conoce el framework Struts, es muy similar al concepto del archivo struts-config.xml: centraliza el funcionamiento de la aplicación, aunque si llega a ser una aplicación muy compleja el archivo se vuelve un tanto complicado de entender y seguir.

Implementación

Sabemos que el patrón Mediator introduce un objeto para mediar la comunicación entre "colegas". Algunas veces el objeto Mediador implementa operaciones simplemente para enviarlas o otros objetos; otras veces pasa una referencia a él mismo y por consiguiente utiliza la verdadera delegación.

Entre los colegas puede existir dos tipos de dependencias:

1. Un tipo de dependencia requiere un objeto para conseguir la aprobación de otros objetos antes de hacer tipos específicos de cambios de estado.
2. El otro tipo de dependencia requiere un objeto para notificar a otros objetos después de que este ha hecho un tipo específico de cambios de estado.

Ambos tipos de dependencias son manejadas de un modo similar. Las instancias de Colega1, Colega2, ... están asociadas con un objeto mediator. Cuando ellos quieren conseguir la aprobación anterior para un cambio de estado, llaman a un método del objeto Mediator. El método del objeto Mediator realiza cuidadoso el resto.

Pero hay que tener en cuenta lo siguiente con respecto al mediador: Poner toda la dependencia de la lógica para un conjunto de objetos relacionados en un lugar puede hacer incomprensible la dependencia lógica fácilmente. Si la clase Mediator llega a ser demasiado grande, entonces dividirlo en piezas más pequeñas puede hacerlo más comprensible.

Código de muestra

Nuestro ejemplo será un chat: donde habrá usuarios que se comunicaran entre sí en un salón de chat. Para ellos se define una interface llamada Chateable que todos los objetos que quieran participar de un chat deberán implementar.

[code]
```
public interface Chateable {
   public void recibe(String de, String msg);
   public void envia(String a, String msg);
}
```

La clase Ususario representa un usuario que quiera chatear.

```
public class Usuario implements Chateable {

    private String nombre;
    private SalonDeChat salon;

    public Usuario(SalonDeChat salonDeChat) {
        salon = salonDeChat;
    }

    public void recibe(String de, String msg) {
        String s = "el usuario " + de + " te dice: " + msg;
        System.out.println(nombre + ": " + s);
    }

    public void envia(String a, String msg) {
        salon.envia(nombre, a, msg);
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public SalonDeChat getSalon() {
        return salon;
    }

    public void setSalon(SalonDeChat salon) {
        this.salon = salon;
    }
}

public interface IChat {
    public abstract void registra(Usuario participante);
    public abstract void envia(String from, String to, String message);
}

public class SalonDeChat implements IChat {

    private Map<String, Usuario> participantes = new HashMap<String, Usuario>();

    @Override
    public void registra(Usuario user) {
        participantes.put(user.getNombre(), user);
    }

    @Override
    public void envia(String de, String a, String msg) {
        if (participantes.containsKey(de) && participantes.containsKey(a)) {
            Usuario u = participantes.get(a);
            u.recibe(de, msg);
        } else {
            System.out.println("Usuario inexistente");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        SalonDeChat salonDeChat = new SalonDeChat();
        Usuario usuario1 = new Usuario(salonDeChat);
        usuario1.setNombre("Juan");
```

```
        Usuario usuario2 = new Usuario(salonDeChat);
        usuario2.setNombre("Pepe");

        Usuario usuario3 = new Usuario(salonDeChat);
        usuario3.setNombre("Pedro");

        salonDeChat.registra(usuario1);
        salonDeChat.registra(usuario2);
        salonDeChat.registra(usuario3);

        usuario1.envia("Pepe", "Hola como andas?");
        usuario2.envia("Juan", "Todo ok, vos?");
        usuario3.envia("Martin", "Martin estas?");
    }
}
```

El resultado por consola es:
Pepe: el usuario Juan te dice: Hola como andas?
Juan: el usuario Pepe te dice: Todo ok, vos?
Usuario inexistente
[/code]

Cuándo utilizarlo

Un ejemplo real que se puede comparar con este patrón es un framework que se llama Struts. Struts posee un clase que hace de Mediadora que se llama ActionServlet. Esta clase lee un archivo de configuración (el struts-config.xml) para ayudarse con la mediación, pero lo importante es que se encarga de comunicar el flujo de información de todos los componentes web de una aplicación. Si no utilizamos Struts, entonces cada página debe saber hacia donde debe dirigir el flujo y el mantenimiento de dicha aplicación puede resultar complicado, especialmente si la aplicación es muy grande.

En cambio, si todas las páginas se comunican con un mediador, entonces la aplicación es mucho más robusta: con cambiar un atributo del mediador todo sigue funcionando igual.

Como conclusión podemos afirmar que este patrón debe ser utilizado en casos donde convenga utilizar un procesador central, en vez de que cada objeto tenga que conocer la implementación de otro. Imaginemos un aeropuerto: que pasaría si no tuviese una torre de control y todos los aviones que deban aterrizar/despegar se tienen que poner todos de acuerdo para hacerlo. Además cada avión debe conocer detalles de otros aviones (velocidad de despegue, nafta que le queda a cada uno que quiera aterrizar, etc).

Para evitar esto se utiliza un torre de control que sincroniza el funcionamiento de un aeropuerto. Esta torre de control se puede ver como un mediador entre aviones.

Patrones relacionados

Patrones con los que potencialmente puede interactuar:

Observer: los colegas pueden comunicarse entre ellos mediante un patrón observador.

Facade: es un concepto similar al mediador, pero este último es un poco más completo y la comunicación es bidireccional.

Singleton: el mediador puede ser Singleton.

Proxy Pattern

Introducción y nombre

Proxy, Estructural. Controla el acceso a un recurso.

Intención

El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.

El patrón obliga que las llamadas a un objeto ocurran indirectamente a través de un objeto proxy, que actúa como un sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.

También conocido como

Surrogate, Virtual Proxy.

Motivación

Necesitamos crear objetos que consumen muchos recursos, pero no queremos instanciarlos a no ser que el cliente lo solicite o se cumplan otras condiciones determinadas.

Puede haber ocasiones en que se desee posponer el coste de la creación de un objeto hasta que sea necesario usarlo.

Solución

Este patrón se debe utilizar cuando:

Se necesite retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario.

Se necesita una referencia a un objeto más flexible o sofisticada que un puntero.

Algunas situaciones comunes de aplicación son:

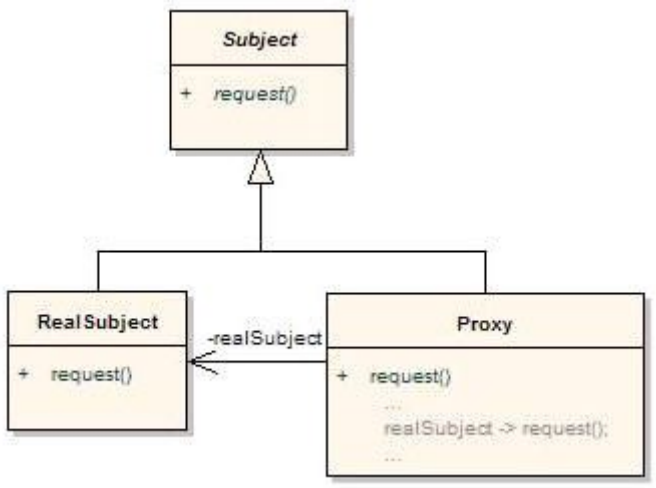
Proxy remoto: representa un objeto en otro espacio de direcciones. Esto quiere decir que el proxy será utilizado de manera tal que la conexión con el objeto remoto se realice de forma controlada sin saturar el servidor.

Proxy virtual: crea objetos costosos por encargo. Cuando se utiliza un software no siempre se cargan todas las opciones por default. Muchas veces se habilitan ciertos módulos sólo cuando el usuario decide utilizarlos.

Proxy de protección: controla el acceso a un objeto. Controla derechos de acceso diferentes.

Referencia inteligente: sustituto de un puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto (ej. contar el número de referencias, cargar un objeto persistente en memoria, bloquear el objeto para impedir acceso concurrente, ...).

Diagrama UML



Participantes

Subject: interfaz o clase abstracta que proporciona un acceso común al objeto real y su representante (proxy).

Proxy: mantiene una referencia al objeto real. Controla la creación y acceso a las operaciones del objeto real.

RealSubject: define el objeto real representado por el Proxy.

Colaboraciones

Cliente: solicita el servicio a través del Proxy y es éste quién se comunica con el RealSubject.

Consecuencias

Introduce un nivel de dirección con diferentes usos:

Un proxy remoto puede ocultar el hecho de que un objeto reside en otro espacio de direcciones.

Un proxy virtual puede realizar optimizaciones, como la creación de objetos bajo demanda.

Los proxies de protección y las referencias inteligentes permiten realizar tareas de mantenimiento adicionales al acceder a un objeto.

Copiar un objeto grande puede ser costoso. Si la copia no se modifica, no es necesario incurrir en dicho gasto.

Implementación

Tenemos un objeto padre Subject del que heredan: RealSubject y Proxy, todos ellos tienen un método request(). El cliente llamaría al método request () de Subject, el cual pasaría la petición a Proxy, que a su vez instanciaría RealSubject y llama a su request().

Esto nos permite controlar las peticiones a RealSubject mediante el Proxy, por ejemplo instanciando RealSubject cuando sea necesario y eliminándolo cuando deje de serlo.

Código de muestra

Vamos a realizar un ejemplo de un proxy remoto: la finalidad es que nuestra aplicación guarde datos en un servidor remoto, pero vamos a impedir se la aplicación se conecte todo el tiempo, sino que aproveche a guardar todo cuando se encuentre conectada. Caso contrario guardará en el disco duro local la información hasta que sea el momento adecuado.

[code]

```
// Esta clase deberia ser un singleton
public class ConnectionManager {

    private static boolean hayConexion;

    public ConnectionManager() {
        hayConexion = false;
    }

    public static void conectate() {
        // Se abre la conexion
        hayConexion = true;
    }

    public static void desconectate() {
        // Se cierra la conexion
        hayConexion = false;
    }

    public static boolean hayConexion() {
        return hayConexion;
    }
}

public interface IGuardar {
    public void save(List datosAGuardar);
}

public class GuardarDiscoDuro implements IGuardar {
    @Override
    public void save(List datosAGuardar) {
        System.out.println("Guardando datos en el HD...");
    }
}

public class ObjetoRemoto implements IGuardar {
    @Override
    public void save(List datosAGuardar) {
        System.out.println("Guardando datos en el objeto remoto...");
    }
}

public class GuardarDatos implements IGuardar {
    @Override
    public void save(List datosAGuardar) {
        if (ConnectionManager.hayConexion()) {
            new ObjetoRemoto().save(datosAGuardar);
        } else {
            new GuardarDiscoDuro().save(datosAGuardar);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        List<String> datos = new ArrayList<String>();
        datos.add("Datos a guardar!!");

        IGuardar g = new GuardarDatos();
        g.save(datos);

        ConnectionManager.conectate();
        g.save(datos);
    }
}
[/code]
```

Cuándo utilizarlo

El patrón Proxy es muy versátil. Puede ser utilizado en infinitas ocasiones y se le puede otorgar varios usos. Tiene una gran ventaja y es que no obliga al desarrollador a crear demasiada estructura para realizar este patrón, sino que es una forma estándar de acceder a una clase que potencialmente puede ser conflictiva. Por otro lado, no ayuda al desarrollador a crear un algoritmo, sino que el desarrollador tiene que hacer toda la lógica.
Por estas razones, es un patrón donde no siempre se puede saber a priori cuando utilizarlo. Sin embargo, un Proxy es un concepto utilizado fuera del ámbito de los patrones de diseño: un servidor proxy es un equipo intermediario situado entre el sistema del usuario e Internet. Puede utilizarse para registrar el uso de Internet y también para bloquear el acceso a una sede Web. El servidor de seguridad del servidor proxy bloquea algunas redes o páginas Web por diversas razones. En consecuencia, es posible que no pueda descargar el entorno de ejecución de Java (JRE) o ejecutar algunos applets de Java.
Es decir, los servidores proxy funcionan como filtros de contenidos. Y también mejoran el rendimiento: guardan en la memoria caché las páginas Web a las que acceden los sistemas de la red durante un cierto tiempo. Cuando un sistema solicita la misma página web, el servidor proxy utiliza la información guardada en la memoria caché en lugar de recuperarla del proveedor de contenidos. De esta forma, se accede con más rapidez.
Este mismo concepto se intenta llevarlo a cabo a nivel código con el patrón Proxy. Cuando un objeto debe ser controlado de alguna manera, ya sea por simple control de acceso o por estar en un sitio remoto o por ser muy pesado y se quiera limitar su uso, es ideal utilizar este patrón.

Patrones relacionados

Facade: en ciertos casos puede resultar muy similar al facade, pero con mayor inteligencia.
Decorator: ciertas implementaciones se asemejan al decorador.
Singleton: la mayoría de las clases que hacen de proxy son Singletons.