Patrones de Diseño

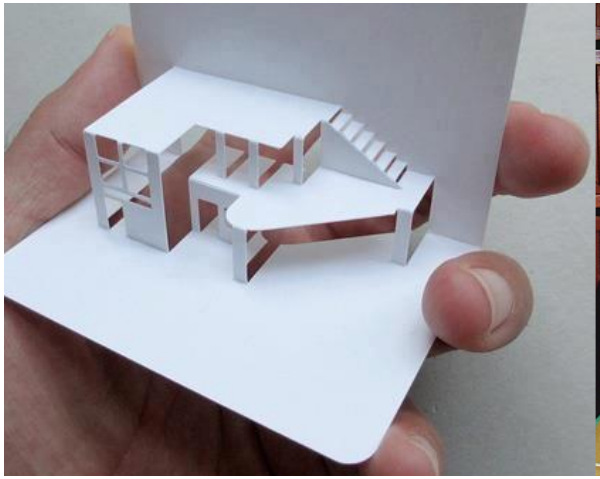
Julio Ariel Hurtado Alegría

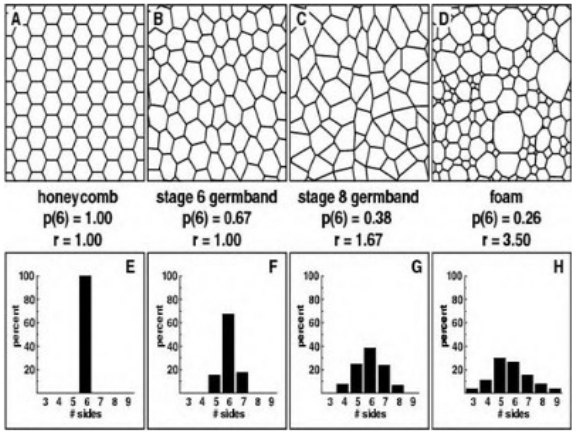
8 de abril de 2013

Contenido

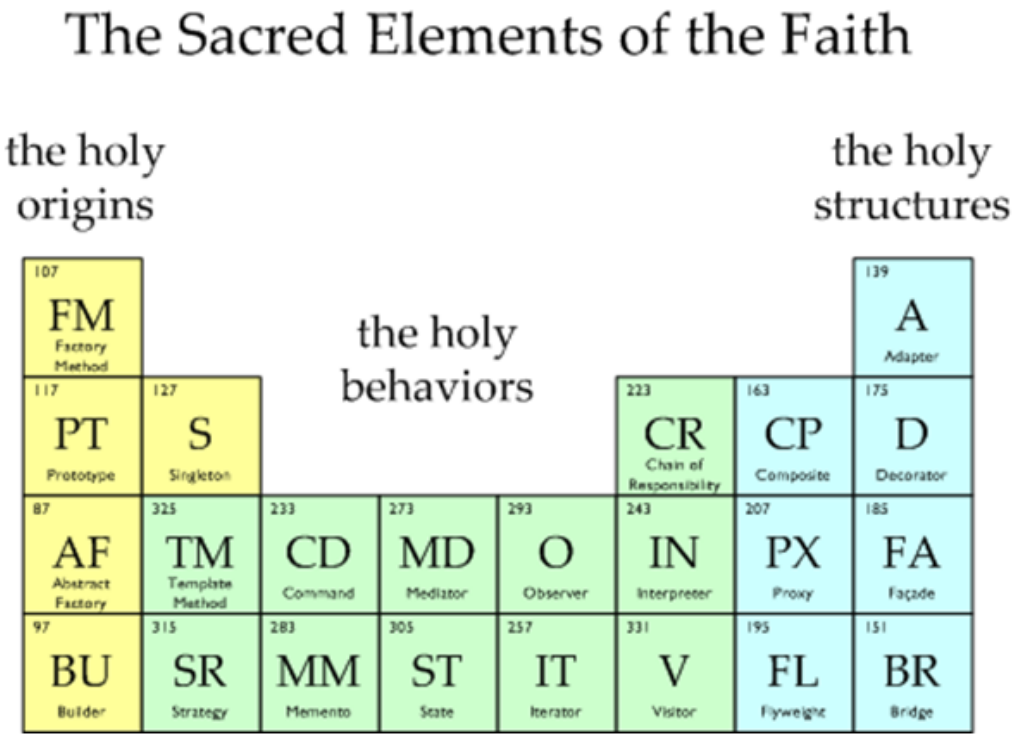
Motivación

La vida real está llena de esquemas que se repiten, por ejemplo la forma en que trabajan un puerto marítimo, un aeropuerto, un terminal de buses y un protocolo de red a nivel de transporte.





Motivación


The Sacred Elements of the Faith
the holy origins
the holy behaviors
the holy structures
107 FM Factory Method
139 A Adapter
117 PT Prototype
127 S Singleton
223 CR Chain of Responsibility
163 CP Composite
175 D Decorator
87 AF Abstract Factory
325 TM Template Method
233 CD Command
273 MD Mediator
293 O Observer
243 IN Interpreter
207 PX Proxy
185 FA Façade
97 BU Builder
315 SR Strategy
283 MM Memento
305 ST State
257 IT Iterator
331 V Visitor
195 FL Flyweight
151 BR Bridge

Definición

Los patrones de diseño facilitan la reutilización del conocimiento en un diseño. Un patrón es un fragmento identificable de conocimiento instructivo, que captura la estructura esencial e interna de una familia de soluciones con probado éxito sobre un problema recurrente dentro de un cierto contexto y fuerzas del software. En otras palabras, un patrón de diseño facilita reutilizar buenas soluciones.

Patrones de Diseño OO

En el enfoque orientada a objetos, un patrón de diseño es una realización probada (diagrama de clases, objetos e interacción) que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de software orientado a objetos. Se describen fundamentalmente en forma textual, acompañados de diagramas y de seudo-código.

Un poco de Historia

1. Alexander, Christopher. A Pattern Language: Towns, Buildings, Construction. 1977
2. Alexander, Christopher. The Timeless Way of Building. 1979
3. Gamma et al. Design Patterns: Elementos of Reusable Object-Oriented Software. 1994
4. Bushmann et al. Pattern-Oriented Software Architecture: A System of Patterns. 1996
5. Coplien y Schmidth. Pattern Languages of Program Design. 1995

Tipos de Patrones

- De arquitectura.
- De diseño.
- Idioms

Aspectos positivos de un patrón de diseño

- Son soluciones concretas, algo así como un conjunto de recetas de diseño.
- Son soluciones técnicas. Hay patrones específicos para una plataforma determinada y otros de carácter más general.
- Se aplican en situaciones muy comunes provenientes de la experiencia.
- Son soluciones simples. Indican cómo resolver un problema particular utilizando un pequeño número de clases relacionadas de forma determinada. No indican cómo diseñar un determinado sistema sino sólo aspectos puntuales del mismo.

Aspectos negativos de un patrón de diseño

- Facilitan la reutilización de las clases y de la propia estructura del diseño.
- El uso de un determinado patrón no se refleja claramente en el código.
- Referencias a this (self)
- Reutilización de la implementación de un patrón.
- Los patrones suponen una sobrecarga de trabajo a la hora de implementar.

Clasificación de los patrones

Creación	*Estructural*	*De Conducta*
Método de fábrica	Adaptador (clases)	Interprete Plantilla
Fábrica Abstracta	Adaptador (objetos)	Cadena de Responsabilidad
Constructor	Puente	Comando
Prototipo	Composición	Iterador
Singleton	Decorador	Intermediario
	Fachada	Observador
	Flyweight	Estado
	Proxy	Estrategia

Patrón Observador

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 observadores del objeto observado, disminuyendo el acoplamiento entre las partes de la solución, creando las mínimas dependencias y evitando ciclos de actualización(espera activa). En definitiva, normalmente, se usa el patrón Observador cuando un elemento quiere estar pendiente de otro, sin tener que estar preguntando de forma permanente si éste ha cambiado o no.

Motivación

Disminuir el acoplamiento entre clases sin perder la consistencia de su comportamiento.

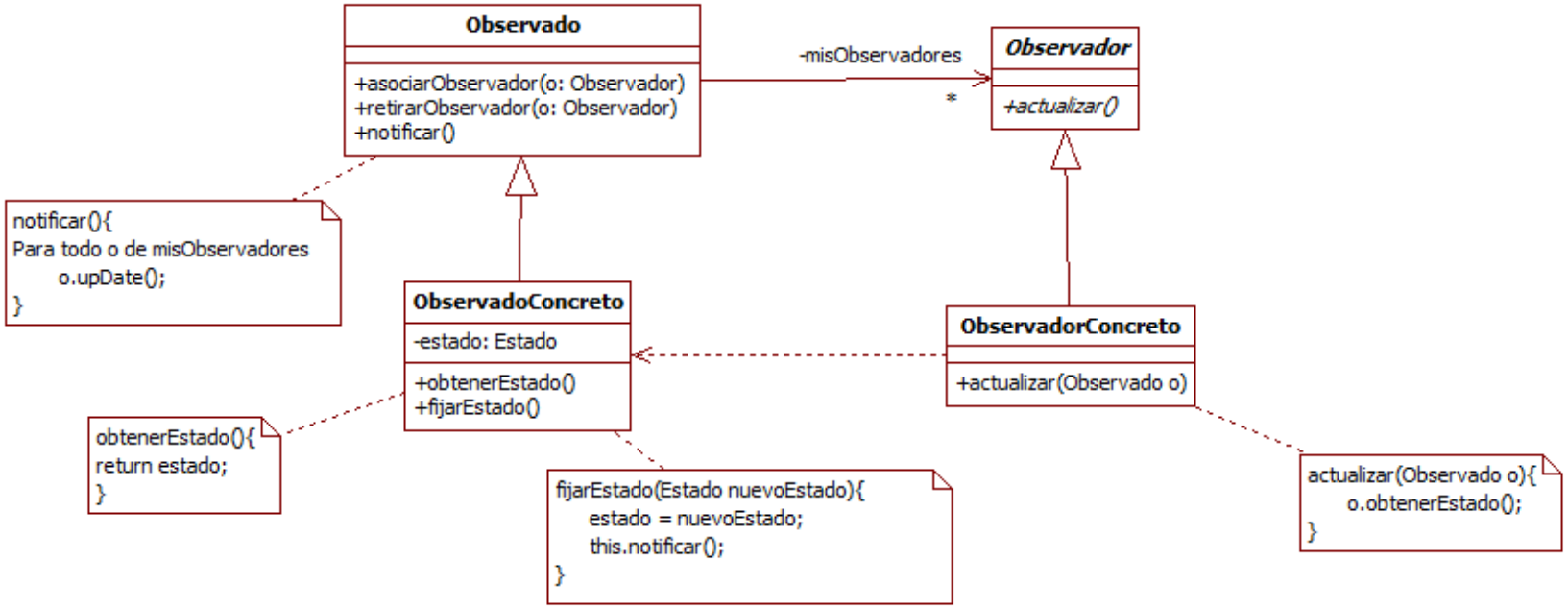
Motivación

Disminuir el acoplamiento entre clases sin perder la consistencia de su comportamiento.

Aplicabilidad

Este patrón aplica cuando una modificación en el estado de un objeto requiere cambios de otros, y no deseamos que se conozca el número de objetos que deben ser cambiados. También cuando queremos 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 encapsulamos estos aspectos en objetos separados permitiremos su variación y reutilización de modo independiente.

Estructura


Observado
+asociarObservador(o: Observador)
+retirarObservador(o: Observador)
+notificar()
-misObservadores
*
Observador
+actualizar()
notificar(){
Para todo o de misObservadores
o.upDate();
}
ObservadoConcreto
-estado: Estado
+obtenerEstado()
+fijarEstado()
ObservadorConcreto
+actualizar(Observado o)
obtenerEstado(){
return estado;
}
fijarEstado(Estado nuevoEstado){
estado = nuevoEstado;
this.notificar();
}
actualizar(Observado o){
o.obtenerEstado();
}

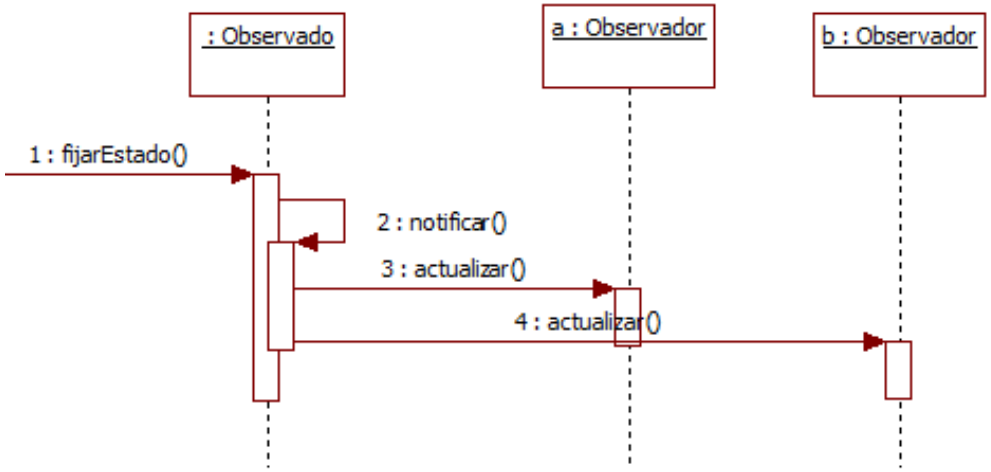
Participantes (1/2)

- *Observado:* el observado proporciona una interfaz concreta para asociar y retirar observadores. El Observado conoce a todos sus observadores.
- *Observador:* define el método abstracto que usa el observado para notificar cambios en su estado (actualizar).

Participantes(2/2)

- *ObservadoConcreto:* 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.
- *ObservadorConcreto:* mantiene una referencia al observado 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. Pueden no tener esta referencia y recibir una referencia al objeto observado a través del mismo método de actualización.

Colaboraciones



Consecuencias

Abstrae el acoplamiento entre el observado y el observador, lo cual es beneficioso ya que conseguimos una mayor independencia y además el observado no necesita especificar los observadores afectados por un cambio. Por otro lado, con el uso de este patrón el observado desconoce las consecuencias de una actualización, lo cual, dependiendo del problema, puede afectarnos el comportamiento del observador o de todo el sistema (por ejemplo, el rendimiento).

Implementación

- Mapeo de los observados a sus observadores.
- Observando más de un observado.
- ¿Quién dispara la actualización?
  1. En las operaciones de cambio.
  2. Hacer a los clientes responsables de llamar el método notificar.

Implementación

- Referencias colgantes a observados eliminados.
- Estar seguro que el estado del observado es auto-consistente antes de la notificación.
- Evitar protocolos de actualización específicos al observador: los modelos push(empujar) y pull(jalar)
- Especificar modificaciones de interés explícitamente.
- Encapsular semántica de actualización compleja.
- Combinando las clases Observado y Observable.

Código Fuente Interface Observador

```
public interface Observador {
    public void actualizar(Observable o);
}
```

Código Fuente Clase Observado

```
public class Observado{
 private LinkedList<Observador> observadores;
 public void agregarObservador(Observador obs){
  if(observadores == null)
    observadores = new LinkedList<Observador>();
    observadores.add(obs);
 }
```

Código Fuente Clase Observado ... Continuación

```
public void notificar(){
  for(Iterator<Observador> e =
    observadores.iterator(); e.hasNext(); ){
      e.next().actualizar(this);
    }
 }
}
```

Código Fuente Clase Observado

```
public class Reloj extends Observado {
 private Manecilla menor;
 public Reloj(Manecilla menor){
   this.menor = menor;}
 public void correr(){
   while(true)
     { menor.incrementar();
      this.notificar();}
   }
  public String getHora(){
   return menor.getValue();
   }
}
```

Código Fuente Visualizador

```
public class VisualizadorReloj extends JFrame
                                implements Observador{
  private Reloj relojito = null;
  public void actualizar(Observable o){
      if(relojito == o) mostrar();}
  public void clickInicio(){
      relojito.correr();
 }
  public void mostrar(){
      etiquetaHora.setText(relojito.getHora());
      this.repaint();
  }
}
```

Código Fuente Alarma

```
public class Alarma {
    private Reloj relojito = null;
    private String horaAlarma;

    public void actualizar(Observable o){
     if(relojito == o)
      if (relojito.getHora()==horaAlarma)
          this.activar();}

     private void activar(){
      // Código de activación de la alarma}
 }
```

Código Fuente Aplicativo - Composición

```
public class AplicativoReloj extends JFrame{
  VisualizadorReloj vistaReloj;
Manecilla hora;
Manecilla min;
Manecilla seg;
Manecilla menor;
Reloj relojito;
```

Código Fuente Aplicativo Reloj Creación

```
public AplicativoReloj(){}
  hora = new Manecilla(0,24,23,null);
  min = new Manecilla(0,60,59,hora);
  seg = new Manecilla(0,60,50,min);
  relojito = new Reloj(seg);
  vistaReloj = new VisualizadorReloj(relojito);
  relojito.agregarObservador(vistaReloj);
  alarmita = new Alarma(relojito);
  relojito.agregarObservador(alarmita);
}
```

Código Fuente Aplicativo Reloj Arranque

```
    public static void main(String args[]){
      AplicativoReloj app = new AplicativoReloj();
      app.setVisible(true);

    }
}
```

Usos conocidos

El uso más concido del patrón es la solución del framework de interface de usuario de Smalltalk Modelo-Vista-Controlador, donde el Modelo es el Observado y la Vista es el observador, el controlador es un intermediario que facilita la observación de los eventos registrados.

Patrones Relacionados

En el *Patrón Mediador*, cuando la semántica de actualización es compleja, por ejemplo un Manejador de Eventos, donde actua como un mediador entre observadores y observados. En el*Patrón Singleton*, el Manejador de Eventos podría ser una sola instancia accesible como una variable global.

Descripción de Un Patrón de Diseño

Dependiendo del autor, del nivel de abstracción y de la publicación misma se han presentado varios formatos para encapsular la información de un patrón. Los puntos más significativos que debe contener un patrón son:

- Nombre
- Problema
- Contexto
- Fuerzas

Descripción de Un Patrón de Diseño

...

- Solución
- Ejemplos
- Contexto Resultante
- Rationale
- Relaciones estáticas y dinámicas del patrón con otros patrones
- Usos conocidos

Descripción Propuesta por Gamma

La propuesta de Gamma (1995), por ejemplo, propone que los elementos esenciales de un patrón son los siguientes:

1. Un nombre del patrón
2. El problema describe cuando aplicar el patrón.
3. La solución describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones.
4. Las consecuencias

Descripción de Un Patrón de Diseño

Extendiendo a a Gamma, tenemos la siguiente lista y descripción de secciones dentro de la plantilla que describe cada patrón. Este formato estructurado es útil puesto que permite la separación semántica de lo que podría haber sido un texto completo y permite además su almacenamiento en una base de datos para su posterior reutilización.

1) Nombre del Patrón y Clasificación, 2) Intención, 3) También conocido como (Sinónimo), 4) Motivación, 5) Aplicabilidad, 6) Estructura, 7) Participantes, 8) Colaboraciones, 9) Consecuencias, 10) Implementación, 11) Ejemplo de código, 12) Usos conocidos y Patrones relacionados.

Conclusión

Los patrones de diseño son aporte significativo a la ingeniería de software porque:

1. Establecen un vocabulario común
2. Promueven un buen diseño OO
3. Sirven para ´preparar nuevos diseñadores
4. Estandarizan la forma en que los diseños son desarrollados
5. Enseñan la Orientación a Objetos
6. Ayudan a hacer refactorizaciones

Patrones de Diseño

Julio Ariel Hurtado Alegría

8 de abril de 2013