PATRONES DE DISEÑO DE SOFTWARE A NIVEL DE CLASE

Nelson Cuvi

e-mail: neo_cuvi@hotmail.es

RESUMEN: Este documento presenta la descripción de lo que son los patrones de diseño de software, sus objetivos, sus características, así también como el estudio de tres de los patrones mas conocidos con sus respectivos ejemplos y gráficos, con lo que se podrá sacar conclusiones acerca de los beneficios adquiridos con la utilización de los patrones de diseño de software.

Palabras Claves: Diseño de Software, Patrones de Diseño, Objetos Reutilizables, Repositorios de Objetos.

1. INTRODUCCIÓN

Es evidente que a lo largo de multitud de diseños de aplicaciones hay problemas que se repiten o que son análogos, es decir, que responden a un cierto patrón. Sería deseable tener una colección de dichos patrones con las soluciones más óptimas para cada caso [1].

Los patrones de diseño no son fáciles de entender, pero una vez entendido su funcionamiento, los diseños serán mucho más flexibles, modulares y reutilizables. Han revolucionado el diseño orientado a objetos y todo buen arquitecto de software debería conocerlos.

Patrones de diseño o más comúnmente conocidos como "Design Patterns". Son soluciones simples y elegantes a problemas específicos y comunes del diseño orientado a objetos. Son soluciones basadas en la experiencia y que se ha demostrado que funcionan.

2. OBJETIVOS DE LOS PATRONES

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

3. CATEGORÍAS DE LOS PATRONES

3.1 PATRONES DE ARQUITECTURA:

Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.

3.2 PATRONES DE DISEÑO: Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.

3.3 DIALECTOS: Patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

3.4 INTERACCIÓN: Son patrones que nos permiten el diseño de interfaces web.

4. BUILDER

El patrón Builder o Constructor es otro de los patrones creacionales. Este patrón permite separar la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda crear diferentes representaciones de este objeto [2].

4.1 DIAGRAMA DE CLASES

- **Builder**

Interfaz abstracta para crear productos.

- **Concrete Builder**

Implementación del Builder, construye y reúne las partes necesarias para construir los productos

- **Director**

Construye un objeto usando el patrón Builder

- **Producto**

El objeto complejo bajo construcción, (Figura 1).

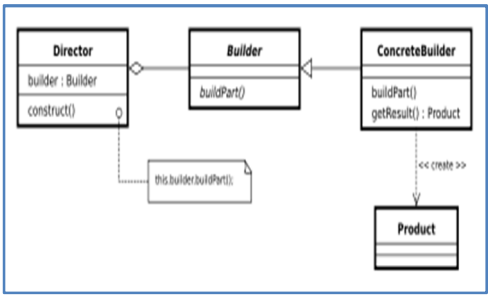

Figura 1. Diagrama de Clases

4.2 VENTAJAS

- Reduce el acoplamiento.

- Permite variar la representación interna de estructuras compleja, respetando la interfaz común de la clase Builder.
- Se independiza el código de construcción de la representación.
- Cada ConcreteBuilder tiene el código específico para crear y modificar una estructura interna concreta.
- Distintos Director con distintas utilidades pueden utilizar el mismo ConcreteBuilder.
- Permite un mayor control en el proceso de creación del objeto.

4.3 EJEMPLO

```
/** "Producto" */
class Pizza {
   private String masa = "";
   private String salsa = "";
   private String relleno = "";

   public void setMasa(String masa)     { this.masa = masa; }
   public void setSalsa(String salsa)     { this.salsa = salsa; }
   public void setRelleno(String relleno) { this.relleno = relleno; }
}


/** "Abstract Builder" */
abstract class PizzaBuilder {
   protected Pizza pizza;

   public Pizza getPizza() { return pizza; }
   public void crearNuevaPizza() { pizza = new Pizza(); }


   public abstract void buildMasa();
   public abstract void buildSalsa();
   public abstract void buildRelleno();
}


/** "ConcreteBuilder" */
class HawaiPizzaBuilder extends PizzaBuilder {
   public void buildMasa()   { pizza.setMasa("suave"); }
   public void buildSalsa()   { pizza.setSalsa("dulce"); }
   public                void                buildRelleno()                {
pizza.setRelleno("chorizo+alcachofas"); }
}
```

5. ITERADOR

El patrón de diseño Iterador (Figura 2), define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección [3]. Algunos de los métodos que podemos definir en la interfaz Iterador son:

Primero(), Siguiente(), HayMas() y ElementoActual().

5.1 PARTICIPANTES

Iterador: Define una interfaz para recorrer los agregados.

IteradorConcreto: Implementa la interfaz iterador.

Agregado: Define la interfaz para crear un objeto iterador.

AgregadoConcreto: Implementa la interfaz de creacion de un iterador para devolver un iterador concreto.

Se han dibujado dos tipos de clientes también:

- ClienteA Este cliente tiene un objeto de la clase Lista y usa iteradores.
- ClienteB Este otro cliente, no contiene al objeto de la clase Lista (como el anterior), sino que sólo la usa, y gracias al interfaz Iterador está desacoplado de la implementación particular.

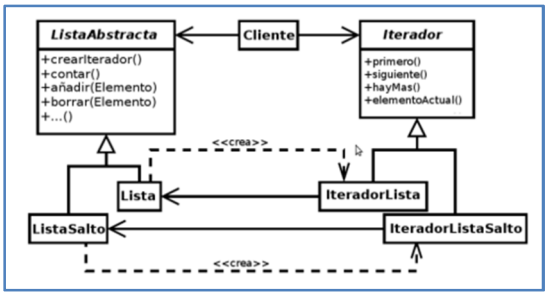

Figura 2. Patrón de Diseño Iterador

5.2 EJEMPLO

```
void ClienteA::MetodoA ()
{
   // Declaramos dos iteradores.
   Lista::Iterador inicio, fin;
   // Declaramos un objeto ClienteB.
   ClienteB clienteB;

   // Inicializamos los iteradores por medio de la lista.
   inicio = lista.Inicio ();
   fin = lista.Fin ();

   // Ahora se los pasamos a Cliente B.
   clienteB.MetodoB (&inicio, &fin);
}


void ClienteB::MetodoB (Iterador *inicio, Iterador *fin)
{
   // Iteramos hasta alcanzar el final, consiguiendo no
acoplarnos con el tipo final.
   while ((*inicio) != (*fin))
   {
      // Hacemos algo con el dato del iterador...
      **inicio;
      // Avanzamos la posición del iterador.
      (*inicio)++;
   }
}
```

6. VISITOR

El patrón visitor es una forma de separar el algoritmo de la estructura de un objeto [4]. Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

Proporcionar una forma fácil y sostenible de ejecutar acciones en una familia de clases. Este patrón centraliza los comportamientos y permite que sean modificados o ampliados sin cambiar las clases sobre las que actúan(Figura 3).

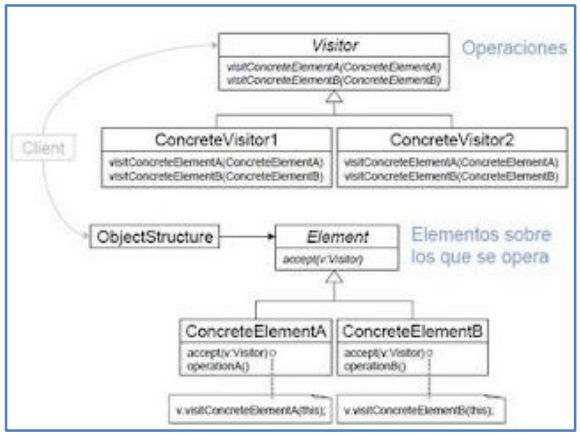

Figura 3. Patrón de Diseño Visitor

6.1 PARTICIPANTES

Visitor: Define una operación de visita para cada clase de elemento concreto en la estructura de objetos.

ConcreteVisitor: Implementa la interfaz Visitor, Cada operación implementa un fragmento de la labor global del visitor concreto, pudiendo almacenar información local.

Element: Define una operación accept con un visitor como argumento.

ConcreteElement: Implementa la operación accept.

ObjectStructure: Gestiona la estructura de objetos, y puede enumerar sus elementos, Puede ser un compuesto (patrón composite) o una colección de objetos, Puede ofrecer una interfaz que permita al visitor visitar a sus elementos.

6.2 CONSECUENCIAS

- Facilita la definición de nuevas operaciones.
- Agrupa operaciones relacionadas.
- Añadir nuevas clases ConcreteElement es costoso.
- Permite atravesar jerarquías de objetos que no están relacionados por un padre común.
- El visitor puede acumular el estado de una operación al visitar la estructura de objetos, en vez de pasarlo como argumento o usar variables globales.
- Rompe la encapsulación.

6.3 EJEMPLO

```
package expresion;
public abstract class Expresion {
  abstract public void aceptar(VisitanteExpresion v);
}

package expresion;
public class Constante extends Expresion {
  public Constante(int valor) { _valor = valor; }
  public void aceptar(VisitanteExpresion v) {
v.visitarConstante(this); }
  int _valor;
}
public class PrettyPrinterExpresion implements
VisitanteExpresion {

  // visitar la variable en este caso es guardar en el resultado la
variable
  // asociada al objeto... Observe que accedemos al estado
interno del objeto
  // confiando en la visibilidad de paquete...

  public void visitarVariable(Variable v) {
       _resultado = v._variable;
  }

  public void visitarConstante(Constante c) {
       _resultado = String.valueOf(c._valor);
}
```

7. REFERENCIAS

[1] Gracia, J. (2005, Mayo 27), *"Patrones de Diseño"*, [En línea]. Disponible en: http://www.ingenierosoftware.com/analisisydiseno/patrones-diseno.php

[2] Roldan, J. (2010, Febrero 9), *"Patrones de Diseño"*, [En línea]. Disponible en: http://tratandodeentenderlo.blogspot.com/2010/02/patrones-de-diseno-builder.html

[3] Perdomo, E. (2010, Marzo), "Iterator", [En línea]. Disponible en: http://ud-csharp.blogspot.com/2010/03/iterator.html

[4] Acosta, O. (2009, Mayo 25), "Patron de Diseño Visitor", [En línea]. Disponible en: http://omaracostacasas.wordpress.com/2009/01/25/patron-de-diseno-visitor/