Web Application Development

Ensayo - Patrones de Diseño

Hernández Salinas Octavio Iván

22 de febrero de 2019

1. Introducción

¿Por qué los patrones de diseño son tan importantes a la hora de desarrollar un sistema computacional?
Esa pregunta no me la hice hasta estar en 5to semestre de la carrera, solo tuvieron que pasar cuatro años para llegar a hacerme esta cuestión desde que escribí mi primer linea de código.

En cursos de Ingeniería de Software te mencionan la palabra «Patrones de Diseño», pero muy pocas veces se llegan a ver con profundidad. Sin embargo, su relevancia en es de suma importante ya que «cualquiera puede programar», pero hacerlo de la forma correcta, muy pocos.

Dicho lo anterior, buscando un poco en libros y notas referenciadas, me encontré con que los patrones de diseño son: «soluciones a problemas comunes en el desarrollo de software», «la experiencia acumulada de programadores que han enfrentado el mismo problema» o, «el esqueleto de las soluciones a problemas comunes en el desarrollo de software». Todas las definiciones anteriores tienen algo en común, **solucionar problemas comunes, mediante experiencias pasadas**.

2. Clasificación de los patrones de diseño

Sabemos que un patrón de diseño es una solucion que surge a partir de que alguien se dio cuenta que se tenia el mismo problema una y otra vez, en determinario escenario.

Posteriormente, se hizo una clasificación al observar similitudes y diferencias entre el conjunto de soluciones que se obtenian a determinados problemas, por lo que se obta por categorizar estas soluciones así, los patrones de diseño popularmente se clasifican de la siguiente forma:

- Patrones Creacionales
- Patrones Estructurales
- Patrones de Comportamiento

3. Patrones Creacionales

Los patrones creacionales se encargan de la inicialización y confugración de los objetos, es decir el como y con que se crearan los objetos del sistema, de esta forma, haciendo más eficiente la creación, configuración y destrucción de los objetos del sistema. Entre los patrones creacionales más conocidos estan:

- Abstract Factory
- Factory Method
- Prototype
- Singleton
- Model View Controller

3.1. Abstract Factory

El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).

3.2. Factory Method

Parte del principio de que las subclases determinan la clase a implementar:

```
public class ConcreteCreator extends Creator
{
  protected Product FactoryMethod()
  {
       return new ConcreteProduct();
  }
```

```
}

public interface Product{}

public class ConcreteProduct implements Product{}

public class Client
{
    public static void main(String args[])
    {
          Creator UnCreator;
          UnCreator = new ConcreteCreator();
          UnCreator.AnOperations();
    }
}
```

3.3. Prototype

Se basa en la clonación de ejemplares copiándolos de un prototipo.

3.4. Singleton

Restringe la instanciación de una clase o valor de un tipo a un solo objeto:

```
public sealed class Singleton
{
    private static volatile Singleton instance;
    private static object syncRoot = new Object();
    private Singleton()
    {
          System.Windows.Forms.MessageBox.Show("Nuevo Singleton");
    }
    public static Singleton GetInstance
    {
        get
        {
            if (instance == null)
            {
                 lock(syncRoot)
                 {
                        if (instance == null){
                            instance = new Singleton();
                        }
                 }
```

```
            }
            return instance;
        }
    }
}
```

3.5. Model View Controller

Este patrón plantea la separación del problema en tres capas: la capa **model**, que representa la realidad; la capa **controler** , que conoce los métodos y atributos del modelo, recibe y realiza lo que el usuario quiere hacer; y la **capa** vista, que muestra un aspecto del modelo y es utilizada por la capa anterior para interaccionar con el usuario.

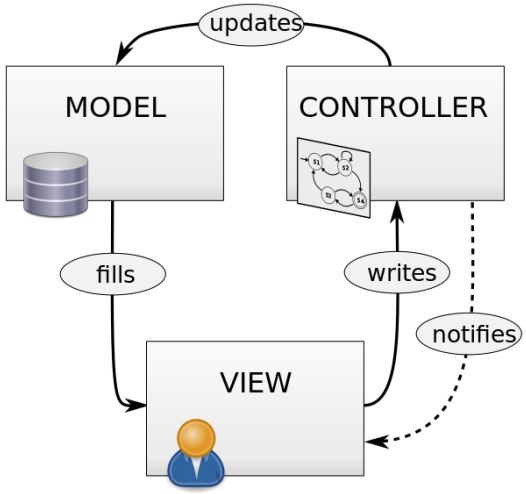

Figura 1: Diagrama patrón de diseño: Model View Controller

4. Patrones Estructurales

Los patrones estructurales se enfocan en como las clases y objetos se componen para formar estructuras mayores, los patrones estructurales describen como las estructuras compuestas por clases crecen para crear nuevas funcionalidades de manera de agregar a la estructura flexibilidad y que la misma pueda cambiar en tiempo de ejecución lo cual es imposible con una composición de clases estáticas. Algunos patrones estructurales son:

- **Adapter**: Convierte una interfaz en otra.

- **Bridge**: Desacopla una abstracción de su implementación permitiendo modificarlas independientemente.
- **Composite**: Utilizado para construir objetos complejos a partir de otros más simples, utilizando para ello la composición recursiva y una estructura de árbol.
- **Decorator**: Permite añadir dinámicamente funcionalidad a una clase existente, evitando heredar sucesivas clases para incorporar la nueva funcionalidad.
- **Facade**: Permite simplificar la interfaz para un subsistema.
- **Flyweight**:Elimina la redundancia o la reduce cuando tenemos gran cantidad de objetos con información idéntica.
- **Proxy**: Un objeto se aproxima a otro.

5. Patrones de comportamiento

Son aquellos que están relacionados con algoritmos y con la asignación de responsabilidades a los objetos. Describen no solamente patrones de objetos o de clases, sino que también engloban patrones de comunicación entre ellos.

- **Chain of responsibility**: La base es permitir que más de un objeto tenga la posibilidad de atender una petición.
- **Command**: Encapsula una petición como un objeto dando la posibilidad de "deshacer" la petición.
- **Interpreter**: Intérprete de lenguaje para una gramática simple y sencilla.
- **Iterator**: Define una interfaz que declara los métodos necesarios para acceder secuencialmente a una colección de objetos sin exponer su estructura interna.
- **Mediator**: Coordina las relaciones entre sus asociados. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones.
- **Memento**: Almacena el estado de un objeto y lo restaura posteriormente.
- **Observer**: Notificaciones de cambios de estado de un objeto.
- **Server**: Se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo.
- **Strategy**: Utilizado para manejar la selección de un algoritmo.
- **Template Method**: Algoritmo con varios pasos suministrados por una clase derivada.

- **Visitor**: Operaciones aplicadas a elementos de una estructura de objetos heterogénea.

6. Finalmente...

Como pudimos ver existen muchos patrones de diseño, los mencionados anteriormente son algunos, sin embargo, existen libros y libros de cientos de hojas que tratan de explicar a detalle los más importantes. Esto debido a la necesidad de construir software que tenga en cierta medida calidad, con los patrones de diseño atacamos una «problemática» el cual es DRY, «**DON'T REPEAT YOURSELF**», entre otros problemas. La razón de ser de los patrones de diseño es juntar la experiencia de proyectos pasados para dar solución a los problemas más comunes a la hora del desarrollo de software.