Patrones de Diseño

Percy Taquila Carazas, Katerin Merino Quispe, Abraham Lipa Calabilla, Edwart Balcon Coahila, Lisbeth Espinoza Caso

October 10, 2020

Abstract

Design patterns provide a coded mechanism for describing problems and their solution in a way that allows the software engineering community to design knowledge for reuse. A pattern describes a problem, indicates the context and allows the user to understand the environment in which the problem occurs, and lists a system of forces that indicate how the problem can be interpreted in context, and the way in which the problem is applied. solution. The Abstract Factory pattern is usually implemented with manufacturing methods that are also generally called from within the Template Method.

Resumen

Los patrones de diseño dan un mecanismo codificado para describir problemas y su solución en forma tal que permiten que la comunidad de ingeniería de software diseñe el conocimiento para que sea reutilizado. Un patrón describe un problema, indica el contexto y permite que el usuario entienda el ambiente en el que sucede el problema, y enlista un sistema de fuerzas que indican cómo puede interpretarse el problema en su contexto, y el modo en el que se aplica la solución. El patron Abstract Factory suele implementarse con metodos de fabricacion que tambien generalmente son llamados desde el interior de Template Method.

I. Introduccion

Los patrones de diseño son muy interesantes para los programadores, ya que nos ofrecen soluciones a problemas comunes y cuotidianos a la hora de diseñar una aplicación. Existen infinidad de casos en que el problema sigue el mismo patrón, solo cambia el contexto; un patrón de diseño te propone una solución a este tipo de problemas..

La manera de utilizarlos depende de dos factores: comprender correctamente cuando se pueden usar y tenerlos presentes a la hora de diseñar. Lo primero se consigue habiéndolos estudiado y puesto en práctica en diferentes contextos. Lo segundo, que también incluye su dificultad, es la capacidad de encontrarse con un problema, y ser capaz de relacionarlo con un patrón de diseño que conozcas.

Las características de un patrón son tres:

- Contexto: situación en la que se presenta el problema de diseño.
- Problema: descripción del problema a resolver, y enumeración de las fuerzas a equilibrar (requisitos no funcionales como eficiencia, portabilidad, cambiabilidad, ...).
- Solución: conjunto de medidas que se han de tomar, como crear alguna clase, atributo o método, nuevos comportamientos entre clases, ...

II. Desarrollo

i. Patrones de diseño estructurales

Consisten en configurar la estructura de nuestra aplicación para cumplir con los principios SOLID, así como mejorar la usabilidad y la mantenibilidad del código. Podemos aplicar los muy conocidos métodos de:

- Herencia

- Composición
- Agregación

También debemos tener en cuenta que hay tres formas de definir estructuras de datos:

- Estáticamente
- Generado por código
- Dinámicamente

i.1 Adaptadores

Tal como el nombre lo dice, mediante este patrón de diseño, podemos adaptar la funcionalidad de una clase a través de una interfaz.

Si tenemos una clase Punto y una clase Linea:

```
public class Punto
{
    public int X, Y;
}
public class Linea
{
    public Punto Inicio, Fin;
}
```

Pensamos en una Figura como una colección de Líneas:

```
public abstract class Figura :
    Collection<Linea> {}
```

Y creamos una clase Rectangulo a partir de la clase Figura (mediante herencia)

```
public class Rectangulo : Figura
{
    public Rectangulo(int x, int y
        , int ancho, int alto)
    {
        // Codigo que agrega
            lineas a la coleccion
    }
}
```

Nos toparemos con un problema. En la interfaz que tenemos para pintar en pantalla tenemos un método que solo funciona con Puntos.

```
public static void PintarPunto(
    Punto p)
{
    // Codigo para pintar en las
        coordenadas X, Y de p
}
```

Para solucionar eso, tenemos los Adaptadores. Necesitamos que el adaptador "convierta" una Línea en una colección de Puntos para que puedan ser pintados con la interfaz.

```
public class AdaptadorLineaAPunto
    : Collection<Punto>
{
    public AdaptadorLineaAPunto(
        Linea linea) {
        // Codigo para agregar los
            puntos en el
            recorrido de las
            Lineas
    }
}
```

Finalmente, implementamos la solución de la siguiente forma.

```
private static void PintarPuntos()
{
    foreach (var linea in
        rectangulo)
    {
        var adaptador = new
            AdaptadorLineaAPunto(
            linea);
        adaptador.ForEach(
            PintarPunto);
    }
}
```

Tomamos cada Linea en el Rectangulo, almacenamos en una variable una instancia del adaptador y pintamos cada Punto en el adaptador.

ii. Patrones de diseño de comportamiento

Consiste en definir un algoritmo como un esqueleto de operaciones y dejar los detalles para que sean implementados por las clases secundarias. La clase principal conserva la

estructura y secuencia general del algoritmo.

Es decir usa la herencia para la distribución del comportamiento. En el patrón del método de plantilla, un algoritmo particular se define en el método de plantilla, pero los pasos exactos de este algoritmo se pueden definir en subclases. El método de plantilla se implementa en una clase abstracta.

Ejemplo: Comida es una clase abstracta con un método de plantilla llamado imprimir () que define los pasos involucrados en una comida. Declaramos el método como final para que no se pueda anular.

```java
package template;

public abstract class Comida {

    // template method
        public final void imprimir() {
            ingredientes();
            cocinar();
            comer();
        }

        public abstract void ingredientes();
        public abstract void cocinar();
        public abstract void comer();

}
```

La clase Hamburguesa extiende de Comida e implementa los tres métodos abstractos de Comida.

```java
package template;

public class Hamburguesa extends Comida {

    @Override
    public void ingredientes() {
        System.out.println("Pollo, papa, carne");
    }

    @Override
    public void cocinar() {
        System.out.println("Cocinar la hamburguesa");
    }

    @Override
    public void comer() {
        System.out.println("Son muy sabrosos!!");
    }
}
```

La clase Salchipapa extiende de Comida e implementa los tres métodos abstractos de Comida.

```java
package template;

public class Salchipapa extends Comida{

    @Override
    public void ingredientes() {
        System.out.println("Papa, hot dog");
    }

    @Override
    public void cocinar() {
        System.out.println("freir las papas, hot dog");
    }


    @Override
    public void comer() {
        System.out.println("Estan ricos!!");
    }

}
```

La clase Demo crea un objeto Hamburguesa y llama al metodo imprimir, luego se crea un objeto Salchipapa y llama a imprimir.

```java
package template;

public class Demo {

    public static void main(String[] args) {

        Comida comida1 = new Hamburguesa();
        comida1.imprimir();

        System.out.println();

        Comida comida2 = new Salchipapa();
        comida2.imprimir();
    }
}
```

La salida de la consola de la ejecución.

```
Console
<terminated> Demo [Java Application] C:\Program Files\Java\jdk1.8.0_201\bin\javaw.exe (10/10/2020 02:50:21 AM – 02:50:22.
Pollo, papa, carne
Cocinar la hamburguesa
Son muy sabrosos!!

Papa, hot dog
freir las papas, hot dog
Estan ricos!!
```

iii. Patrones de diseño creacionales

Consisten en configurar la estructura de nuestra aplicación para cumplir con los principios SOLID, así como mejorar la usabilidad y la mantenibilidad del código. Podemos aplicar los muy conocidos métodos de: Se ocupa de los mecanismos de creación de objetos, tratando de crear objetos de una manera adecuada a la situación. En otras palabras, este patrón busca, de alguna manera "despreocupar" al sistema de cómo sus objetos son creados o compuestos.

iii.1 Singleton

Este patrón involucra una sola clase que es responsable de crear un objeto mientras se asegura de que solo se cree un objeto.

Si tenemos una clase Conexión, creamos un objeto, hacemos que el constructor sea privado para que no pueda ser instanciado y creamos un método para obtener la instancia únicamente por el mismo.

```
public class clsConexion
{
    private static clsConexion
        cnxMySQL;

    private clsConexion() {
    }

    public static clsConexion
        getInstancia() {
        if (cnxMySQL == null) {
            cnxMySQL = new
                clsConexion();
        }
        return cnxMySQL;
    }

    public Connection
        getConnection() {
        Connection Mysql = null;
        try {
            MysqlConnectionPool
            DataSource ds = new
                MysqlConnectionPool
                 DataSource();
            ds.setServerName("
                localhost");
            ds.setPort(3306);
            ds.setDatabaseName("
                db_modelo");
            Mysql = ds.
                getConnection("
                root", "");

        } catch (Exception ex) {
            JOptionPane.
                showMessageDialog(
                null, 'Error de
                conexion a la BD')
                ;
        }
        return Mysql;
    }
}
```

Y en las demás clases podemos instanciar sin necesidad del operador "new":

```
public class clsNegocioActividad
    implements
    clsInterfaceActividad {

    clsConexion c = clsConexion.
        getInstancia();

    @Override
    public void AgregarActividad(
        clsEntidadActividad
        Actividad) {
        try {
            Connection conexion =
                c.getConnection();
            CallableStatement cst
                = conexion.
                prepareCall("{call
                USP_Actividad_I
                (?,?,?)}");
            cst.setString("
                psemestre_id",
                Actividad.
                getSemestre_id());
            cst.setString("
                pcriterio_id",
                Actividad.
                getCriterio_id());
            cst.setString("pnombre
                ", Actividad.
                getNombre());
            cst.execute();
        } catch (SQLException ex)
            {
            ex.printStackTrace();
        }
    }
}
```

iii.2 Fábrica

Sirve para construir una jerarquía de clases. Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.

Primero creamos una clase interface

```
public interface
    clsInterfaceConexion {

    void getConnection();
}
```

Luego declaramos clases que implementen la interface ya creada

```
public class clsConexionMySQL
    implements
    clsInterfaceConexion {

    @Override
    public void getConnection() {
        Connection Mysql = null;
        try {
            MysqlConnectionPool
                DataSource ds =
                new
                MysqlConnectionPool
                 DataSource();
            ds.setServerName("
                localhost");
            ds.setPort(3306);
            ds.setDatabaseName("
                db_modelo");
            Mysql = ds.
                getConnection("
                root", "");

        } catch (Exception ex) {
            JOptionPane.
                showMessageDialog(
                null, 'Error de
                conexion a la BD')
                ;
        }
    }

}
```

Creamos la clase fábrica para generar un objeto de una clase según la información dada

```
public class ConexionFabrica {

    public clsInterfaceConexion
        getConnection(String motor
        ) {
        if (motor == null) {
            return new
                clsConexionVacia()
                ;
        } else if (motor.
            equalsIgnoreCase("
            MySQL")) {
            return new
                clsConexionMySQL()
                ;
        } else if (motor.
            equalsIgnoreCase("
            Oracle")) {
            return new
                clsConexionOracle
                ();
        }
        return new
            clsConexionVacia();
    }

}
```

Y por último usamos la clase fábrica para obtener el objeto de la clase que solicitamos pasando algún tipo de información

```
public class Aplicacion {
    public static void main(String
        [] args) {

        ConexionFabrica fabrica =
            new ConexionFabrica();

        clsInterfaceConexion cx1 =
            fabrica.getConnection
            ("MySQL");
        cx1.getConnection();

        clsInterfaceConexion cx2 =
            fabrica.getConnection
            ("Oracle");
```

```
        cx2.getConnection();

    }
}
```

III. Conclusiones

La conclusión es sencilla, si no usas patrones, deberías hacerlo. Los patrones ayudan a estandarizar el código, haciendo que el diseño sea más comprensible para otros programadores. Son muy buenas herramientas, y como programadores, siempre deberíamos usar las mejores herramientas a nuestro alcance

IV. Recomendaciones

- Cuando se conoce el efecto colateral que conlleva el patrón de diseño y es viable la aparición de este efecto.
- Suministrar alternativas de diseño para poder tener un software flexible y reutilizable.

Referencias

[1] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John(1995).Design Patterns: Elements of Reusable Object- Oriented Software. Reading,Massachusetts: Addison Wesley Longman, Inc.

[2] Nesteruk, D. (2019). Design Patterns in .NET: Reusable Approaches C# in and F# for Object-Oriented Software Design (1st ed.). Apress.

[3] Patrones de diseño en Java: Los 23 modelos de diseño: descripción y solución ilustradas en UML 2 y Java Autor: Laurent Debrauwer

[4] Patrones de Diseño. Elementos de software orientado a objetos reutilizable. ERICH GAMMA. RICHARD HELM. RALPH JOHNSON. JOHN VLISSIDES.