Patrones de Diseño

Flores Navarro Deivis Jhonatan, Gonzales Franco Daniel Alejandro,
Delgado Castillo Jesus, Jimenez Silva Darold

March 29, 2022

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 (patrón de diseño) representa una mejor práctica, a menudo es adoptado por los desarrolladores de software orientado a objetos experimentados.

Utilizar patrones de diseño es una de las maneras de aumentar la flexibilidad de los componentes de software y hacerlos más fáciles de reutilizar. Sin embargo, en ocasiones esto tiene el precio de complicar los componentes.

Proyecto de uso racional de los patrones de diseño puede ser la solución perfecta para muchos problemas, cada modo corresponde a un principio de la realidad, cada modo describe un problema que se repite constantemente a nuestro alrededor, así como el problema de solución central, que es ampliamente utilizado en patrones de diseño.

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y describe la esencia de la solución a ese problema, de tal modo que pueda utilizarse esta solución un millón de veces más, sin siquiera hacerlo de la misma manera dos veces.

El uso de patrones ayuda a obtener un software de calidad (reutilización y extensibilidad)

II. Desarrollo

Los design patterns, son una solución general, reutilizable y aplicable a diferentes problemas de diseño de software. Se trata de plantillas que identifican problemas en el sistema y proporcionan soluciones apropiadas a problemas generales a los que se han enfrentado los desarrolladores durante un largo periodo de tiempo, a través de prueba y error.

Todos los patrones de diseño tienen algunas características de uso común: Logotipo(Un nombre de patrón), Planteamiento del problema(Un enunciado del problema) y Solución(a solution)Efecto(consequences).

Clasificación de los patrones de diseño

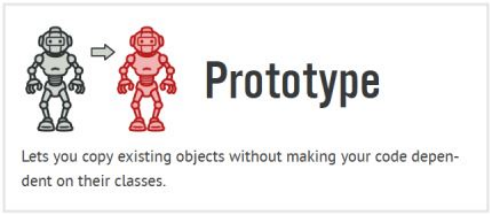
1. Según su propósito (para qué se utiliza el modelo), se puede dividir en tres tipos:
   - Creativo:
   - Estructural
   - Comportamiento.

2. Según el alcance (Se utiliza principalmente para tratar la relación entre clases o entre objetos), Se puede dividir en dos tipos:
   - Patrón de clase
   - Modo de objeto

Tipos de patrones de diseño de software
Los patrones de diseño más utilizados se clasifican en tres categorías principales, cada patrón de diseño individual conforma un total de 23 patrones de diseño. Las categorías principales son:

i. Patron creacional

Proporcionan diversos mecanismos de creación de objetos, que aumentan la flexibilidad y la reutilización del código existente de una manera adecuada a la situación.



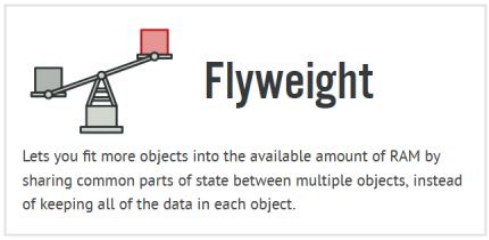
Ejemplo del Patrón Singleton:

```
public class Main {
    public static void main(String[] args) {
//        Singlenton x = new Singlenton("a");
        Singlenton juan = Singlenton.getInstance();
        Singlenton jose = Singlenton.getInstance();

        juan.setName("juan");
        System.out.println(jose.getName());

        jose.setName("jose");
        System.out.println(juan.getName());
    }
}

public class Singleton {
    private String name;
    private static Singleton singlenton;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (singlenton == null) {
```

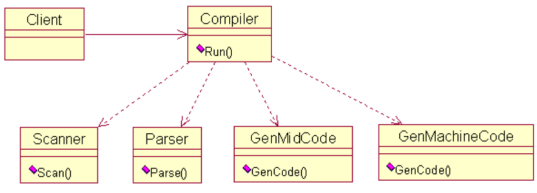
```
            singlenton = new Singleton();
        }
        return singlenton;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

ii. Patron Estructural

Facilitan soluciones y estándares eficientes con respecto a las composiciones de clase y las estructuras de objetos. El concepto de herencia se utiliza para componer interfaces y definir formas de componer objetos para obtener nuevas funcionalidades



Ejemplo:

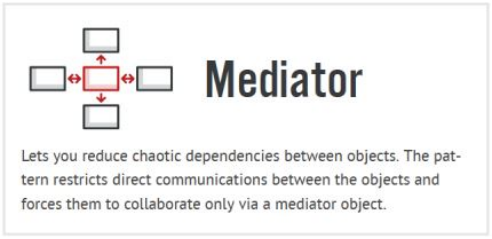


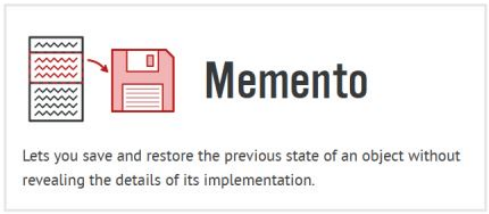
```cpp
#include <iostream>
#include <memory>
#include <utility>

class Scanner
{
public:
        void Scan () {std :: cout << "Análisis léxico" << std :: endl;}
};
class Parser
{
public:
        void Parse () {std :: cout << "Análisis de sintaxis" << std :: endl;}
};
class GenMidCode
{
public:
        void GenCode () {std :: cout << "Generar código intermedio" << std :: endl;}
};
class GenMachineCode
{
public:
        void GenCode () {std :: cout << "Generar código de máquina" << std :: endl;}
};
 // Interfaz de alto nivel
class Compiler
{
public:
       Compiler()
       :scanner{ std::make_unique<Scanner>() }
       ,parser{ std::make_unique<Parser>() }
       ,genMidCode{ std::make_unique<GenMidCode>() }
       ,genMacCode{ std::make_unique<GenMachineCode>() }
       {
       }
       void run()
       {
              scanner->Scan();
              parser->Parse();
              genMidCode->GenCode();
              genMacCode->GenCode();
       }

private:
       std::unique_ptr<Scanner> scanner;
       std::unique_ptr<Parser> parser;
       std::unique_ptr<GenMidCode> genMidCode;
       std::unique_ptr<GenMachineCode> genMacCode;
};

int main()
{
       Compiler compiler;
       compiler.run();

       return 0;
}
```

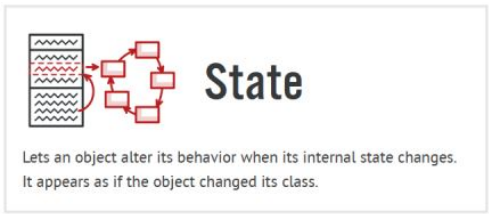
iii. Patrones de comportamiento

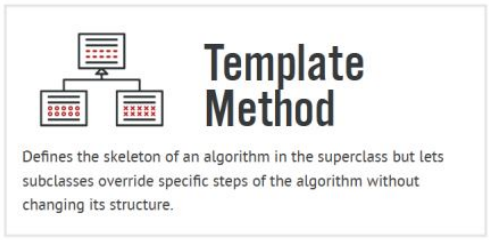
Se ocupa de la comunicación entre objetos de clase. Se utilizan para detectar la presencia de patrones de comunicación ya presentes y pueden manipular estos patrones.

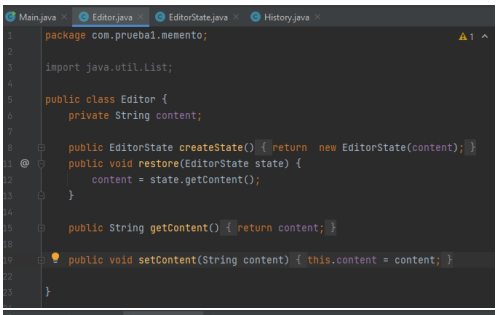
Estos patrones de diseño están específicamente relacionados con la comunicación entre objetos.



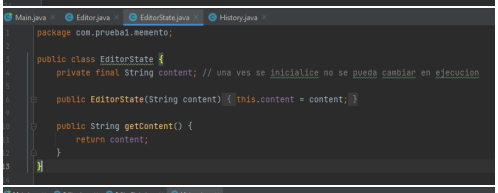




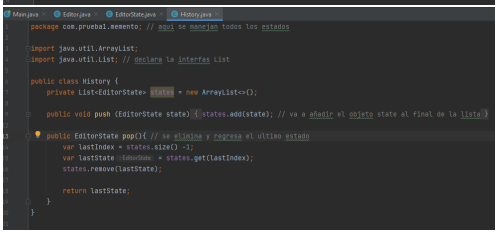




```java
package com.prueba1.memento;

import java.util.List;

public class Editor {
    private String content;

    public EditorState createState() { return  new EditorState(content); }
    public void restore(EditorState state) {
        content = state.getContent();
    }

    public String getContent() { return content; }

    public void setContent(String content) { this.content = content; }

}
```

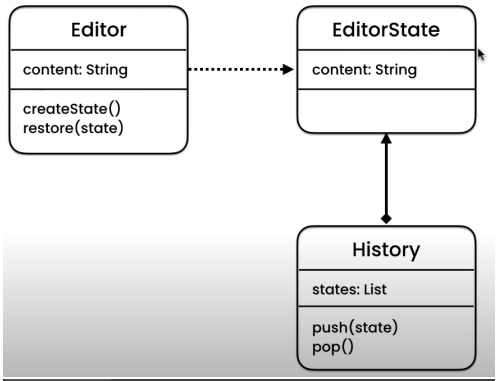


```java
package com.prueba1.memento;

public class EditorState {
    private final String content; // una ves se inicialice no se pueda cambiar en ejecucion

    public EditorState(String content) { this.content = content; }

    public String getContent() {
        return content;
    }
}
```



```java
package com.prueba1.memento; // aqui se manejan todos los estados

import java.util.ArrayList;
import java.util.List; // declara la interfas List

public class History {
    private List<EditorState> states = new ArrayList<>();

    public void push (EditorState state){ states.add(state); // va a añadir el objeto state al final de la lista }

    public EditorState pop(){ // se elimina y regresa el ultimo estado
        var lastIndex = states.size() -1;
        var lastState = states.get(lastIndex);
        states.remove(lastState);

        return lastState;
    }
}
```

Ejemplo:




```java
package com.prueba1;

import com.prueba1.memento.Editor;
import com.prueba1.memento.History;

public class Main {

    public static void main(String[] args) {
        var editor = new Editor();
        var history = new History();
        editor.setContent("a");
        history.push(editor.createState());

        editor.setContent("b");
        history.push(editor.createState());

        editor.setContent("c");
        editor.restore(history.pop());
        editor.restore(history.pop());

        System.out.println(editor.getContent());
    }

}
```

Principios de patrones de diseño
Algunos de los principios centrales son:

1. Abierto-Cerrar"Principio(Open - Closed Principle Abreviatura:OCP) Abierto a extensión, cerrado a modificación:
   Un buen sistema puede extender su función sin modificar el código fuente. La clave para lograr el principio de apertura y cierre es abstracto.

   Al expandir los sistemas de software existentes, se pueden proporcionar nuevos comportamientos para satisfacer las nuevas demandas de software y hacer que el software cambiante sea adaptable y flexible.

   En el principio de "abrir-cerrar", no se permite modificar clases o interfaces abstractas, y se puede extender clases de implementación específicas. Las clases e interfaces abstractas juegan un papel extremadamente importante en el principio de "abrir-cerrar". Es necesario anticipar los requisitos que pueden cambiar.

2. Principio de sustitución de Leeb: Donde puede aparecer cualquier clase base,También pueden aparecer subclases. Principio de sustitución de Liskov:
   Las subclases deben poder reemplazar donde puede aparecer la clase base. Las subclases también pueden agregar comportamientos basados en la clase base. Este Habla sobre la relación entre la clase base y las subclases, solo cuando esta relación existe el principio de sustitución de Liskov.
   Un cuadrado es un rectángulo es un ejemplo clásico de entender el Principio de sustitución de Liskov.

3. Principio de dependencia: Confíe en la abstracción, no en la implementación concreta. En pocas palabras, el principio de inversión de dependencia requiere que los clientes confíen en el acoplamiento abstracto.
   Declaración de principios: La abstracción no debe depender de los detalles; los detalles deben depender de la abstracción.

   Si el principio de apertura y cierre es el objetivo, el principio de inversión de dependencia es el medio para alcanzar el principio de "apertura y cierre". Si desea alcanzar el mejor principio de "apertura y cierre", debe seguir el principio de inversión de dependencia tanto como sea posible.

4. Principio de síntesis - agregación multiplexación (CARP):Intente utilizar el principio de síntesis en lugar de herencia para lograr el propósito de reutilizar el software.

   Tambien se le conoce como el Principio de reutilización compuesto o CRP. Se utiliza algunos objetos existentes en un nuevo objeto para que forme parte del nuevo objeto. Los nuevos objetos logran el propósito de reutilizar las funciones existentes delegando en estos objetos. En resumen, use la composición/agregación tanto como sea posible e intente no usar la herencia.

5. La Ley de Demeter (Ley de Demeter o LoD para abreviar) también se conoce como el Principio de menor conocimiento (LKP para abreviar), lo que significa que un objeto debe tener el menor conocimiento posible sobre otros objetos.

   Otras expresiones: solo comunícate con tus amigos directos, no hables con "extraños". Una clase debe saber lo menos sobre la clase que necesita ser acoplada o invocada. No me importa cuán compleja sea (clase acoplada o invocada).

6. Reglas de aislamiento de la interfaz: Esta ley es igual a la ley de Dimit.
   El principio de segregación de interfaz dice que siempre es mejor usar múltiples interfaces especializadas que una sola interfaz total. En otras palabras, desde la perspectiva de una clase de cliente: la dependencia de una clase de otra clase debe basarse en la interfaz más pequeña.

Una interfaz demasiado hinchada es una contaminación de la interfaz. Los clientes no deberían verse obligados a confiar en los métodos que no utilizan.

III. Conclusiones

Como se puede apreciar a lo largo de este documento, la utilización de patrones provee una flexibilidad que permite el desarrollo de aplicaciones más abiertas al cambio y a las actualizaciones. Sin embargo, estos beneficios podrían presentar algunas desventajas que deben considerarse. Los patrones de diseño pueden complicar el período de pruebas o la complejidad del código, y por ende queda a criterio de los desarrolladores analizar durante el proceso de modelado, el impacto de su aplicación.

IV. Recomendaciones

- Cuando se conoce el efecto colateral que conlleva el patrón de diseño y es viable la aparición de este efecto.
- No es necesario aprender todos los patrones de diseños, lo más importante es entender o investigar los patrones que nos puedan servir en cada situación.
- Si aplicando un patrón a un proyecto se obtienen igual o mas desventajas que ventajas, lo mejor es cambiar de patrón.
- No pensar en un lenguaje de programación cuando se esta eligiendo un patron de diseño. Los patrones no son dependientes de un lenguaje, esto permite de alguna forma acomodarse lo mejor posible a los proyectos.
- 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] Canelo, M. M. (2021, December 24). ¿Qué son los patrones de diseño de software? Profile Software Services. https://profile.es/blog/patrones-de-diseno-de-software/

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Addison Wesley (GoF- Gang of Four) - Design Patterns. Elements of Reusable Object-Oriented Software

[4] E. (2022, March 29). Patrones de diseño e implementación - Cloud Design Patterns. Microsoft Docs. https://docs.microsoft.com/es-es/ azure/architecture/patterns/category/design-implementation

[5] Pavon Mestras, J. (2004, June 5). Patrones de diseño orientado a objetos. Facultad de Informatica UCM https://www.fdi.ucm.es/ profesor/jpavon/poo/2.14pdoo.pdf

[6] Shvets, A. (2019). DESIGN PATTERNS: Vol. 1.7 (1ra. ed.). Jorge F. Ramírez Ariza. https://refactoring.guru/files/design-patterns-es-demo.pdf

[7] Guerrero, C. A. (2013). Patrones de Diseño GOF (The Gang of Four) en el contexto de Procesos de Desarrollo de Aplicaciones Orientadas a la Web. Scielo. https://www.scielo.cl/scielo.php ?script=sci$_a rttext pid = S0718 - 07642013000300012f$

[8] Design patterns Tutorial-Comenzando con los patrones de diseño. (2022, March 1). https://learntutorials.net/es/design-patterns/topic/1012/comenzando-con-los-patrones-de-diseno

[9] Mosh Hamedani (2020, January 6) - Design Patterns in plain english. https://www.youtube.com/watch?v=NU$_1$*StN*5*Tkk*

[10] Marinescu, F. (2002). EJB Design Patterns: Vol. 1.0 (3rd Edition ed.). Robert Elliott http://staff.cs.utu.fi/staff/jouni.smed/doos$_0$6/*material*/*DesignPrinciplesAndPatterns.pdf*