Asignatura	Datos del alumno	Fecha
Métodos avanzados de programación científica y computación.	**Apellidos:** Balsells Orellana	25/01/2021
	Nombre: Jorge A.	

Laboratorio: Otros patrones de diseño.

Describe tres patrones de diseño diferentes a los que hemos estudiado en el tema. Debes indicar de qué tipo son, así como el problema que resuelven, la solución y las consecuencias de aplicar cada uno de ellos. Realizar una implementación de ejemplo en Java.

Resumen

Los patrones de diseño son soluciones a problemas frecuentes cuando se diseña software. Para diseñno, no solo dependemos de herramientas como los diagramas UML, sino también de buenas prácticas de programación para poder globalizar el código de una manera mas eficiente y comprensible. Los patrones de diseño no son funciones, librerías ni porciones de código a copiar y pegar en nuestro desarrollo, mas bien es un concepto para buenas prácticas ordenadas. Para comprender los patrones de diseño, es importante tener conocimiento básico sobre diagramas UML, el lenguaje de programación a utilizar(en este caso Java) y POO. también existe la necesidad de comprenderque es un método constructor. Un constructor es un método inicial el cual, al crearse, se ejecutaautomaticamente cuando se crea un objeto de una clase.[2]

1. Patrón de diseño 1:

1. **Tipo de patrón de diseño:** Creacional.
2. **Nombre de patrón de diseño:** Builder.
3. **Problema a resolver del patrón de diseño:** Crear múltiples objetos complejos a través de un objeto inicial o fuente. De manera concurrente, el patrón de diseño *Builder* construye el patrón de diseño *Composite*, que es un patrón estructural visto en clase. Este patrón de diseño se aplica cuando nuestro sistema trata con objetos compuestos con muchos atributos, pero el número de configuraciones es limitado. En este caso la solución será crear un constructor que permita construir todos los tipos de objetos.
4. **Solución del patrón de diseño:** Este patrón Abstrae los procesos de creación de un objeto complejo centralizando el proceso en un único punto de partida en una clase, de tal manera que

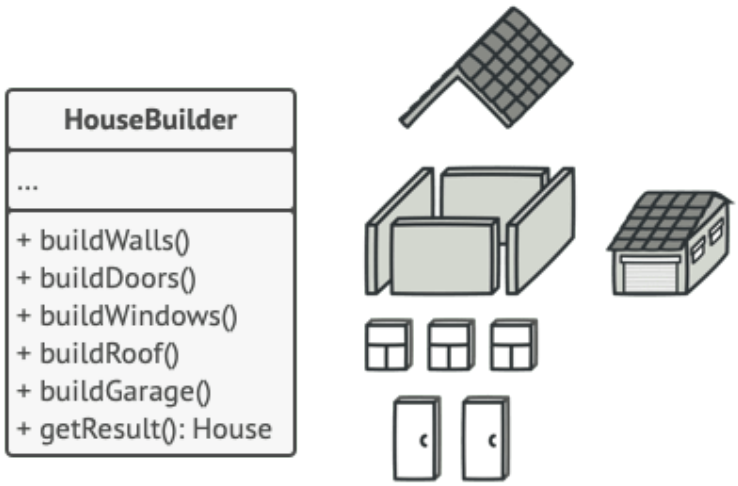

Figura 1: Representación gráfica de patrón creacional Builder. Fuente: *https://refactoring.guru/es/design-patterns/builder*

el mismo método constructor, pueda generar representaciones diferentes. Con *Builder* se puede crear una anidación de métodos, ya que cada método devielve solamente un *this*. Normalmente es una clase que contiene muchos métodos que retornan *this*, y *this* hace referencia al objeto que se va a crear una vez que la clase principal sea instanciada. A diferencia de otros patrones de diseño, en este caso no se devuelve el objeto, solamente estamos exportando la clase. Básicamente, separamos el constructor de un objeto complejo, para que el mismo proceso de construcción, pueda crear diferentes representaciones. [3]. Entre las ventajas de este patrón de diseño tenemos las siguientes:

a) Mayor control en los procesos de creación de un objeto.

b) Se puede utilizar la misma clase para diferentes utilidades.

c) Cada m étodo interno tiene el código para crear o modificar una estructura interna concreta.

d) Se independiza el código de construcción de la representación.

e) Permite variar representaciones internas de estructuras complejas.

La figura 1 muestra un ejemplo de un problema a resolver por este patrón de diseño.

5. **Consecuencias del patrón de diseño:** Las consecuencias positivas de este patrón de diseño son que reduce el acoplamiento, y permite variar la representación interna del objeto. Tal y como

lo mencionamos arriba, permite independizar la construcción y la representación. De la misma manera, como consecuencias negativas tenemos que agrega una mayor complejidad al desarrollo de software, y si este no se encuentra bien documentado, puede ser una tendencia a generar problemas posteriores.

```
1  package patrones;
2
3  public class Computer {
4      private String HDD;
5      private String RAM;
6      private boolean isGraphicsCardEnabled;
7
8      public String getHDD() {
9          return HDD;}
10
11     public String getRAM() {
12         return RAM;}
13
14     public boolean isGraphicsCardEnabled() {
15         return isGraphicsCardEnabled;}
16
17     private Computer(ComputerBuilder builder) {
18         this.HDD=builder.HDD;
19         this.RAM=builder.RAM;
20         this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;}
21
22     public static class ComputerBuilder{
23         private String HDD;
24         private String RAM;
25         private boolean isGraphicsCardEnabled;
26
27         public ComputerBuilder(String hdd, String ram){
28             this.HDD=hdd;
```

```
29              this.RAM=ram;}
30
31          public ComputerBuilder setGraphicsCardEnabled(boolean ...
                isGraphicsCardEnabled) {
32              this.isGraphicsCardEnabled = isGraphicsCardEnabled;
33              return this;}
34
35          public Computer build(){
36              return new Computer(this);}
37      }
38  }
```

Listing 1: Código de ejemplo en Java de implementación de patron creacional Builder.[1]

2. Patrón de diseño 2

1. **Tipo de patrón de diseño:** Estructural.
2. **Nombre de patrón de diseño:** Adapter.
3. **Problema a resolver del patrón de diseño:** Transforma una interfaz en otra. Practicamente es útil en el caso en el que necesitemos crear nuevas clases en un futuro, y esto pueda llevar a cambios drásticos al crear clases nuevas. Básicamente sirve para reutilización de código y evitar problemas de compatibilidad con nuevos desarrollos por medio de adaptadores que crean compatibilidad dentro de las clases diferentes de una clase *Target* a una clase *Adaptee* a través de una clase *Adapter*. En otras palabras, se desea usar una clase existente, y la interfaz no es igual a lo que se necesita, o mas bien, permite que un conjunto de clases con interfaces incompatibles trabajen juntas.
4. **Solución del patrón de diseño:** Crear una clase intermedia que conecte las clases que se necesitan conectar. En este caso los participantes son **Target**, que define la interfaz específica de dominio utilizada. **Client**, que colabora con los objetos con interfaz definida por *Target*. **Adaptee**, que define una interfaz existente que necesita ser adaptada, y por último **Adapter**,

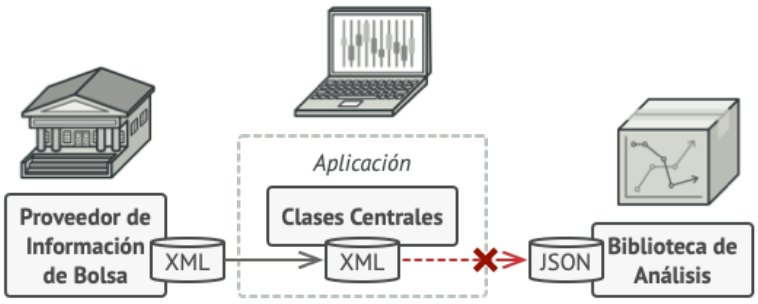

Figura 2: Representación gráfica de patrón estructural Adapter. Fuente: *https://refactoring.guru/es/design-patterns/adapter*

que es la interfaz adaptadora definida por el *Target*. Es necesario mencionar que la adaptación puede llegar a ser bidireccional. La figura 2 muestra un ejemplo de un problema a resolver por este patrón de diseño.

5. **Consecuencias del patrón de diseño:**La mayor consecuencia es que dificulta sobreescribir el comportamiento del *Adaptee*, y al ser una adaptación, agrega carga de procesamiento al desarrollo. También dificulta la comprensión del programa sin la documentación correcta, ya que una clase puede estar definida a funcionar de una manera específica, y a través de un adaptador los resultados pueden ser totalmente diferentes. Tampoco sirve para adaptar una clase y todas sus subclases. Una consecuencia positiva es que puede maximizar la reutilización de clases y se adapta dinámicamente a una de varias clases. Esto es útil clando se tienen creadas clases muy complejas y utilizadas en múltiples casos.

```
1 public class Main {
2  public static void main(String[] args) {
3     MediaPlayer player = new MP3();
4     player.play("file.mp3");
5     player = new FormatAdapter(new MP4());
6     player.play("file.mp4");
7     player = new FormatAdapter(new VLC());
8     player.play("file.avi");}}
```

Listing 2: Código de ejemplo en Java de implementación de patron estructural Adapter.[1]

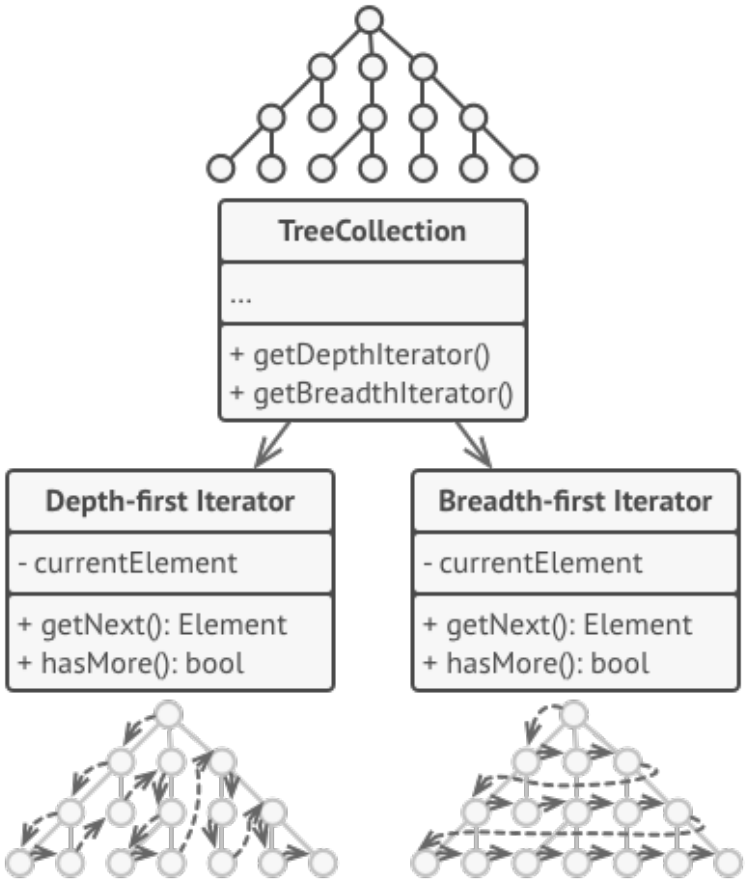

Figura 3: Representación gráfica de patrón de comportamiento Iterator. Fuente: *https://refactoring.guru/es/design-patterns/iterator*

3. Patrón de diseño 3

1. **Tipo de patrón de diseño:** De comportamiento.
2. **Nombre de patrón de diseño:** Iterator.
3. **Problema a resolver del patrón de diseño:**Este patrón de diseño se usa específicamente en el diseño de listas y conjuntos. se aplica cuando se debe trabajar con objetos que, internamente trabajan sobre un grupo de elementos y se deban manejar los elementos sin que un cambio en la implementación pueda afectar el sistema completo de manera global.
4. **Solución del patrón de diseño:** Se crea una interfaz *Iterator* la cuál, estandariza los métodos para tratar un conjunto de elementos, por lo cual, la interfaz definirá una serie de operaciones

para manipular los elementos del conjunto. La figura 3 muestra un ejemplo de un problema a resolver por este patrón de diseño.

5. **Consecuencias del patrón de diseño:**Entre las consecuencias positivas tenemos que estandariza el tratado de elementos de listas con implementaciones independientes. También, modificando solo la clase que contiene el iterador, permite variar el tratamiento de las listas, y se pueden realizar múltiples recorridos simultáneos. Esto como consecuencia negativa, genera un aumento en la jerarquía de clases.

```
1  public interface Iterator {
2  public void first();
3  public void next();
4  public boolean isDone();
5  public Object currentItem();
6  }
7  public interface Aggregate {
8  public Iterator createIterator();
9  public Object get(int);
10 public int count();
11 }
12 public class ListIterator {
13 private Aggregate a;
14 private int current;
15 ListIterator (Aggregate a); {
16 this.a = a;
17 current = 0;
18 }
19 public void first() { current = 0; }
20 public void next() { current++; }
21 public boolean isDone() { return current ≥ a.count(); }
22 public Object currentItem() { return a.get(current); }}
```

Listing 3: Código de ejemplo en Java de implementación de patron de comportamiento Iterator.[1]

Referencias

[1] Patrones de diseño. urlhttps://refactoring.guru/es/design-patterns/java.

[2] Maria Luisa. Díez Platas. Apuntes de clase de métodos avanzados de programación científica y computación., 2020.

[3] Johnson Ralph Vissides John Gamma Erich, Helm Richard. Design patterns - elements of reusable object-oriented software., 1994.