Ejemplificación patrones de diseño

Patrón de comportamiento

1. Memento

1.1 Definición Memento

De acuerdo con Guru Refactoring el patrón de diseño Memento "permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación."

1.2 Primer ejemplo Memento

Según Guru Refactoring se plantea la creación de una aplicación de edición de texto, en la cual el programa puede formatear e insertar imágenes en línea; es decir, antes de realizar cualquier operación, la aplicación registra el estado de los objetos y lo guarda en un almacenamiento y cuando se desea revertir la acción, la aplicación extrae la última instantánea del historial y la usa para restaurar el estado de los objetos.

En el ejemplo Guru Refactoring "utiliza el patrón Memento junto al patrón Command para almacenar instantáneas del estado complejo del editor de texto y restaurar un estado previo a partir de estas instantáneas cuando sea necesario".

1.2.1 Codigo fuente primer ejemplo

//El originador contiene información importante que puede cambiar con el paso del tiempo. También define un método para guardar su estado dentro de un memento, y otro método para restaurar el estado a partir de él.

```
class Editor is
    private field text, curX, curY, selectionWidth

    method setText(text) is
        this.text = text

    method setCursor(x, y) is
        this.curX = x
        this.curY = y

    method setSelectionWidth(width) is
        this.selectionWidth = width

//Guarda el estado actual dentro de un memento.

  method createSnapshot():Snapshot is
```

//El memento es un objeto inmutable; ese es el motivo por el que el originador pasa su estado a los parámetros de su constructor.

```
        return new Snapshot(this, text, curX, curY, selectionWidth)

//La clase memento almacena el estado pasado del editor.

class Snapshot is
    private field editor: Editor
    private field text, curX, curY, selectionWidth

    constructor Snapshot(editor, text, curX, curY, selectionWidth) is
        this.editor = editor
        this.text = text
        this.curX = x
        this.curY = y
        this.selectionWidth = selectionWidth

//En cierto punto, puede restaurarse un estado previo del editor utilizando un objeto memento.

method restore() is
        editor.setText(text)
        editor.setCursor(curX, curY)
        editor.setSelectionWidth(selectionWidth)

// Un objeto de comando puede actuar como cuidador. En este caso, el comando obtiene un memento
justo antes de cambiar el estado del originador. Cuando se solicita deshacer, restaura el estado del
originador a partir del memento.

class Command is
    private field backup: Snapshot

    method makeBackup() is
        backup = editor.createSnapshot()

    method undo() is
        if (backup != null)
            backup.restore()
    // ...
```

1.3 Segundo ejemplo Memento

Se requiere guardar el nombre de una persona que puede variar a lo largo del tiempo, en función de historial de persona encargada del manejo pasado de ciertas operaciones (Max, 2011).

1.3.1 Código fuente segundo ejemplo

```
public class EjemploMemento {

    private String estado;
```

```
    public EjemploMemento(String estado){
        this.estado=estado;
    }

    public String getSavedState(){
        return estado;
    }
}


public class Persona {

    private String nombre;

    public EjemploMemento saveToMemento(){
        System.out.println("Originador: Guardando Memento...");
        return new EjemploMemento(nombre);
    }

    public void restoreFromMemento(EjemploMemento m){
        nombre=m.getSavedState();
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre){
        this.nombre=nombre;
    }
}


import java.util.ArrayList;

public class Caretaker {

    private ArrayList<EjemploMemento>estados=new ArrayList<EjemploMemento>();

    public void addMemento (EjemploMemento m){
        estados.add(m);
    }

    public EjemploMemento getMemento(int index){
        return estados.get(index);
    }

    public static void main(String[] args) {
    Caretaker caretaker=new Caretaker();
```

```
    Persona p=new Persona();
    p.setNombre("Miguel");
    p.setNombre("Julian");

    caretaker.addMemento(p.saveToMemento());

    p.setNombre("Pablo");

    caretaker.addMemento(p.saveToMemento());

    p.setNombre("David");

    EjemploMemento m1=caretaker.getMemento(0);
    EjemploMemento m2=caretaker.getMemento(1);

        System.out.println(m1.getSavedState());
        System.out.println(m2.getSavedState());

    }

}
```

Patrón Estructural

2. Flyweight

2.1 Definición Flyweight

Según Guru Refactoring el patrón de diseño Flyweight “permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto”.

2.2 Primer ejemplo Flyweight

Conforme a Guru Refactoring se plantea crear un sencillo videojuego en el que los jugadores se tienen que mover por un mapa disparándose entre sí, en este se decide implementar un sistema de partículas realistas que lo distinga de otros juegos, lo que representan grandes cantidades de balas, misiles y metralla de las explosiones las cuales volarán por todo el mapa, ofreciendo así una apasionante experiencia al jugador.

Sin embargo al realizar testeo del juego se descubre que este al usarse en una máquina menos potente (Insuficiente RAM) presenta problemas como detenerse en un partida, esto se deriva del sistema de las partículas (bala, misil, metralla) implementadas en el juego ya que tienen gran cantidad de datos (objeto).

Asimismo en este caso se destacan otras partes dentro del objeto que son los estados de la partículas (coordenadas, vector de movimiento, velocidad) que cambian a lo largo del tiempo y que pueden ser

alteradas por otros objetos conocido como estado extrínseco, el cual para evitar fallas este será dependiente a métodos específicos de las partículas del juego; y, para el caso del estado intrínseco que se mantiene constante, que existen dentro del objeto y otros objetos pueden leerla y no cambiarla se manifiesta que este estado se almacene dentro del objeto para ser reutilizados en distintos contextos porque representan menos variaciones al otro estado (color y el sprite de las parttículas). Por ende en este caso se tienen en cuenta los tres objetos de las partículas del juego que son una bala, un misil y un trozo de metralla.

2.2.1 Código fuente primer ejemplo

```
//La clase flyweight contiene una parte del estado de un árbol. Estos campos almacenan valores que son únicos para cada árbol en particular. Por ejemplo, aquí no encontrarás las coordenadas del árbol. Pero la textura y los colores que comparten muchos árboles sí están aquí. Ya que esta cantidad de datos suele ser GRANDE, dedicarás mucha memoria a mantenerla en cada objeto árbol. En lugar de eso, podemos extraer la textura, el color y otros datos repetidos y colocarlos en un objeto independiente que muchos objetos individuales del árbol pueden referenciar.

class TreeType is
    field name
    field color
    field texture
    constructor TreeType(name, color, texture) { ... }
    method draw(canvas, x, y) is

// 1. Crea un mapa de bits de un tipo, color y textura concretos.
//2. Dibuja el mapa de bits en el lienzo con las coordenadas X y Y.


// La fábrica flyweight decide si reutiliza el flyweight existente o si crea un nuevo objeto.

class TreeFactory is
    static field treeTypes: collection of tree types
    static method getTreeType(name, color, texture) is
        type = treeTypes.find(name, color, texture)
        if (type == null)
            type = new TreeType(name, color, texture)
            treeTypes.add(type)
        return type

// El objeto contextual contiene la parte extrínseca del estado del árbol. Una aplicación puede crear millones de ellas, ya que son muy pequeñas: dos coordenadas en números enteros y un campo de referencia.

class Tree is
    field x,y
    field type: TreeType
    constructor Tree(x, y, type) { ... }
    method draw(canvas) is
        type.draw(canvas, this.x, this.y)
```

// Las clases Tree y Forest son los clientes de flyweight. Puedes fusionarlas si no tienes la intención de desarrollar más la clase Tree.

```
class Forest is
    field trees: collection of Trees

    method plantTree(x, y, name, color, texture) is
        type = TreeFactory.getTreeType(name, color, texture)
        tree = new Tree(x, y, type)
        trees.add(tree)

    method draw(canvas) is
        foreach (tree in trees) do
            tree.draw(canvas)
```

2.3 Segundo ejemplo Flyweight

Un colegio tiene promedio general de 6, se busca saber que porcentaje de desviación con respecto al promedio tiene cada alumno (Max,2011).

2.3.1 Código fuente segundo ejemplo

```
public class Alumno {

    private String nombre;
    private String apellido;
    private double promedio;
    private double promedioGeneral;

    public Alumno(double promedioGeneral){
        setPromedioGeneral(promedioGeneral);
    }

    public double compara(){
        return (((double)promedio)/promedioGeneral-1)*100;
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellido() {
        return apellido;
    }
```

```
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    public double getPromedio() {
        return promedio;
    }

    public void setPromedio(double promedio) {
        this.promedio = promedio;
    }

    public void setPromedioGeneral(double promedioGeneral) {
        this.promedioGeneral = promedioGeneral;
    }

    public double getPromedioGeneral() {
        return promedioGeneral;
    }


    public Alumno(String nombre, String apellido, double promedio, double promedioGeneral) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.promedio = promedio;
        this.promedioGeneral = promedioGeneral;
    }
}

public class Main {

    public static void main(String[] args) {

        double promedioGeneral=6;

        String nombres[]={"Julian","Miguel","Pablo"};
        String apellidos[]={"Petro","Laso","Mendez"};
        double promedios[]={6,7,9};

        Alumno alumno=new Alumno(promedioGeneral);
        for (int i = 0; i < nombres.length; i++) {
            alumno.setNombre(nombres[i]);
            alumno.setApellido(apellidos[i]);
            alumno.setPromedio(promedios[i]);
            System.out.println(nombres[i]+ ":"+ alumno.compara());
        }
    }
}
```

Referencias bibliográficas

Guru Refactoring. Memento. Recuperado de https://refactoring.guru/es/design-patterns/memento

Max. (2011). Mi granito de Java. Memento. Recuperado de http://migranitodejava.blogspot.com/2011/06/memento.html

Guru Refactoring. Flyweight. Recuperado de https://refactoring.guru/es/design-patterns/flyweight

Max. (2011). Mi granito de Java. Flyweight. Recuperado de http://migranitodejava.blogspot.com/search/label/Flyweight