Maria Fernanda Olaya Nieves

PATRONES DE DISEÑO

PROTOTYPE:

Es un patrón de diseño creacional que permite copiar objetos existentes sin que el código dependa de sus clases

Ejemplo 1:

proceso de la división mitótica de una célula. Tras la división mitótica, se forma un par de células idénticas. La célula original actúa como prototipo y asume un papel activo en la creación de la copia.

```
// Prototipo base.
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    // Un constructor normal.
    constructor Shape() is
        // ...

    // El constructor prototipo. Un nuevo objeto se inicializa
    // con valores del objeto existente.
    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    // La operación clonar devuelve una de las subclases de
    // Shape (Forma).
    abstract method clone():Shape


// Prototipo concreto. El método de clonación crea un nuevo
// objeto y lo pasa al constructor. Hasta que el constructor
// termina, tiene una referencia a un nuevo clon. De este modo
// nadie tiene acceso a un clon a medio terminar. Esto garantiza
// la consistencia del resultado de la clonación.
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        // Para copiar campos privados definidos en la clase
        // padre es necesaria una llamada a un constructor
        // padre.
        super(source)
```

```
        this.width = source.width
        this.height = source.height

    method clone():Shape is
        return new Rectangle(this)


class Circle extends Shape is
    field radius: int

    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius

    method clone():Shape is
        return new Circle(this)


// En alguna parte del código cliente.
class Application is
    field shapes: array of Shape

    constructor Application() is
        Circle circle = new Circle()
        circle.X = 10
        circle.Y = 10
        circle.radius = 20
        shapes.add(circle)

        Circle anotherCircle = circle.clone()
        shapes.add(anotherCircle)
        // La variable `anotherCircle` (otroCírculo) contiene
        // una copia exacta del objeto `circle`.

        Rectangle rectangle = new Rectangle()
        rectangle.width = 10
        rectangle.height = 20
        shapes.add(rectangle)

    method businessLogic() is
        // Prototype es genial porque te permite producir una
        // copia de un objeto sin conocer nada de su tipo.
        Array shapesCopy = new Array of Shapes.

        // Por ejemplo, no conocemos los elementos exactos de la
        // matriz de formas. Lo único que sabemos es que son
        // todas formas. Pero, gracias al polimorfismo, cuando
        // invocamos el método `clonar` en una forma, el
        // programa comprueba su clase real y ejecuta el método
        // de clonación adecuado definido en dicha clase. Por
        // eso obtenemos los clones adecuados en lugar de un
        // grupo de simples objetos Shape.
        foreach (s in shapes) do
            shapesCopy.add(s.clone())

        // La matriz `shapesCopy` contiene copias exactas del
        // hijo de la matriz `shape`.
```

Ejemplo 2:

El diseñador de la empresa SOCIOS&CIA ha implementado para su colección de verano una nueva camiseta color verde talla M con un espectacular estampado en la parte frontal de la misma que dice “Hola”; a los ejecutivos de la empresa les gusto el nuevo estilo y decidieron realizar la copia de 1000 camisetas para esta temporada de verano.

```
import java.util.Objects;

public abstract class Camisa {
    public int x;
    public int y;
    public String color;

    public Camisa() {
    }

    public Camisa(Camisa target) {
        if (target != null) {
            this.x = target.x;
            this.y = target.y;
            this.color = target.color;
        }
    }

    public abstract Camisa clone();

    @Override
    public boolean equals(Object object2) {
        if (!(object2 instanceof Camisa)) return false;
        Camisa camisa2 = (Camisa) object2;
        return camisa2.x == x && camisa2.y == y &&
Objects.equals(camisa2.color, color);
    }
}
```

OBSERVER:

Es un patrón de diseño de comportamiento que permite notificar a sus clientes las actuaciones o tareas que hace el operador

Ejemplo 1:

Si te suscribes a un periódico o una revista, ya no necesitarás ir a la tienda a comprobar si el siguiente número está disponible. En lugar de eso, el notificador envía nuevos números directamente a tu buzón justo después de la publicación, o incluso antes.

El notificador mantiene una lista de suscriptores y sabe qué revistas les interesan. Los suscriptores pueden abandonar la lista en cualquier momento si quieren que el notificador deje de enviarles nuevos números.

```
// La clase notificadora base incluye código de gestión de
// suscripciones y métodos de notificación.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)

// El notificador concreto contiene lógica de negocio real, de
// interés para algunos suscriptores. Podemos derivar esta clase
// de la notificadora base, pero esto no siempre es posible en
// el mundo real porque puede que la notificadora concreta sea
// ya una subclase. En este caso, puedes modificar la lógica de
// la suscripción con composición, como hicimos aquí.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Los métodos de la lógica de negocio pueden notificar los
    // cambios a los suscriptores.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)

    // ...


// Aquí está la interfaz suscriptora. Si tu lenguaje de
// programación soporta tipos funcionales, puedes sustituir toda
// la jerarquía suscriptora por un grupo de funciones.


interface EventListener is
    method update(filename)

// Los suscriptores concretos reaccionan a las actualizaciones
// emitidas por el notificador al que están unidos.
class LoggingListener implements EventListener is
    private field log: File
```

```
    private field message

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))


// Una  aplicación puede configurar notificadores y suscriptores
// durante el tiempo de ejecución.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)
```

Ejemplo 2:

la república de Colombia quiere informar a su población que ya están disponibles las segundas dosis para la vacunación contra el covid-19, se quiere hacer difusión mediante un mensaje de texto a las líneas móviles de dicha población.

```
package Observer;

 public class UnObservador implements IObservador

{

    private String nombre;

    // --------------------------------

    public UnObservador(String nombre) {

        this.setNombre(nombre);
```

```
        System.out.println("Observador [" + this.nombre + "] creado");
    }
    // ---------------------------------
    public String getNombre() {
        return this.nombre;
    }
    // ---------------------------------
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    // ---------------------------------
    @Override
     public void observadoActualizado() {
       System.out.println("Observador [" + this.getNombre() + "] recibe
la notificación");
    }
}
```