Patrones Creacionales

Los patrones creacionales se enfocan en la forma en que los objetos son creados en un sistema de software. Estos patrones proporcionan diversas formas de crear instancias de objetos, asegurando que el proceso de creación sea adecuado para la situación específica. El objetivo es aumentar la flexibilidad y reutilización del código al abstraer el proceso de instanciación, permitiendo que el sistema decida cómo y cuándo se crean los objetos.

Singleton

Asegura que una clase tenga solo una instancia y proporciona un punto global de acceso a esa instancia

```
// Producto abstracto
abstract class Producto {
    abstract void operacion();
}

// Productos concretos
class ProductoA extends Producto {
    void operacion() {
        System.out.println("Producto A");
    }
}

class ProductoB extends Producto {
    void operacion() {
        System.out.println("Producto B");
    }
}

// Creator abstracto
abstract class Creador {
    abstract Producto factoryMethod();

    void operacion() {
        Producto producto = factoryMethod();
        producto.operacion();
    }
}
```

```
// Creadores concretos
class CreadorA extends Creador {
    Producto factoryMethod() {
        return new ProductoA();
    }
}

class CreadorB extends Creador {
    Producto factoryMethod() {
        return new ProductoB();
    }
}

// Uso del patrón Factory Method
public class Main {
    public static void main(String[] args) {
        Creador creadorA = new CreadorA();
        creadorA.operacion();

        Creador creadorB = new CreadorB();
        creadorB.operacion();
    }
}
```

Abstract Factory

Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Es útil cuando el sistema debe ser independiente de cómo se crean y organizan los productos.

```
// Abstract Factory
interface GUIFactory {
    Boton crearBoton();
    CheckBox crearCheckBox();
}

// Concrete Factory 1
```

```
class WindowsFactory implements GUIFactory {
    public Boton crearBoton() {
        return new WindowsBoton();
    }
    public CheckBox crearCheckBox() {
        return new WindowsCheckBox();
    }
}

// Concrete Factory 2
class MacOSFactory implements GUIFactory {
    public Boton crearBoton() {
        return new MacOSBoton();
    }
    public CheckBox crearCheckBox() {
        return new MacOSCheckBox();
    }
}

// Productos abstractos
interface Boton {
    void pintar();
}

interface CheckBox {
    void marcar();
}

// Productos concretos
class WindowsBoton implements Boton {
    public void pintar() {
        System.out.println("Pintar botón estilo Windows.");
    }
}

class MacOSBoton implements Boton {
    public void pintar() {
        System.out.println("Pintar botón estilo MacOS.");
    }
}

class WindowsCheckBox implements CheckBox {
    public void marcar() {
```

```
        System.out.println("Marcar checkbox estilo Windows.");
    }
}

class MacOSCheckBox implements CheckBox {
    public void marcar() {
        System.out.println("Marcar checkbox estilo MacOS.");
    }
}

// Uso del patrón Abstract Factory
public class Application {
    private Boton boton;
    private CheckBox checkBox;

    public Application(GUIFactory factory) {
        boton = factory.crearBoton();
        checkBox = factory.crearCheckBox();
    }

    public void pintar() {
        boton.pintar();
        checkBox.marcar();
    }

    public static void main(String[] args) {
        GUIFactory factory = new WindowsFactory();
        Application app = new Application(factory);
        app.pintar();
    }
}
```

Builder

Separa la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción puede crear diferentes representaciones.

```
// Producto final
class Pizza {
    private String masa;
```

```
    private String salsa;
    private String relleno;

    public void setMasa(String masa) {
        this.masa = masa;
    }

    public void setSalsa(String salsa) {
        this.salsa = salsa;
    }

    public void setRelleno(String relleno) {
        this.relleno = relleno;
    }

    public void mostrar() {
        System.out.println("Pizza con masa: " + masa + ", salsa: " +
salsa + ", relleno: " + relleno);
    }
}

// Builder abstracto
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() {
        return pizza;
    }

    public void crearNuevaPizza() {
        pizza = new Pizza();
    }

    public abstract void construirMasa();
    public abstract void construirSalsa();
    public abstract void construirRelleno();
}

// Concrete Builder 1
class HawaiPizzaBuilder extends PizzaBuilder {
    public void construirMasa() {
        pizza.setMasa("suave");
    }
```

```
    public void construirSalsa() {
        pizza.setSalsa("dulce");
    }
    public void construirRelleno() {
        pizza.setRelleno("piña+jamón");
    }
}

// Concrete Builder 2
class PicantePizzaBuilder extends PizzaBuilder {
    public void construirMasa() {
        pizza.setMasa("corteza fina");
    }
    public void construirSalsa() {
        pizza.setSalsa("picante");
    }
    public void construirRelleno() {
        pizza.setRelleno("pepperoni+salami");
    }
}

// Director
class Cocina {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }

    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }

    public void construirPizza() {
        pizzaBuilder.crearNuevaPizza();
        pizzaBuilder.construirMasa();
        pizzaBuilder.construirSalsa();
        pizzaBuilder.construirRelleno();
    }
}

// Uso del patrón Builder
public class Main {
```

```
    public static void main(String[] args) {
        Cocina cocina = new Cocina();
        PizzaBuilder hawaiPizzaBuilder = new HawaiPizzaBuilder();
        PizzaBuilder picantePizzaBuilder = new PicantePizzaBuilder();

        cocina.setPizzaBuilder(hawaiPizzaBuilder);
        cocina.construirPizza();
        Pizza pizza1 = cocina.getPizza();
        pizza1.mostrar();

        cocina.setPizzaBuilder(picantePizzaBuilder);
        cocina.construirPizza();
        Pizza pizza2 = cocina.getPizza();
        pizza2.mostrar();
    }
}
```

Prototype

Permite crear nuevos objetos clonando una instancia existente. Es útil cuando se necesita crear una copia exacta de un objeto o cuando el costo de crear una instancia es caro.

```
// Interfaz Cloneable
abstract class Forma implements Cloneable {
    private String id;
    protected String tipo;

    abstract void dibujar();

    public String getTipo() {
        return tipo;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
```

```
    public Object clonar() {
        Object clon = null;
        try {
            clon = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clon;
    }
}

// Clases concretas
class Circulo extends Forma {
    public Circulo() {
        tipo = "Círculo";
    }

    public void dibujar() {
        System.out.println("Dibujando un Círculo");
    }
}

class Cuadrado extends Forma {
    public Cuadrado() {
        tipo = "Cuadrado";
    }

    public void dibujar() {
        System.out.println("Dibujando un Cuadrado");
    }
}

// Uso del patrón Prototype
public class Main {
    public static void main(String[] args) {
        Circulo circuloOriginal = new Circulo();
        circuloOriginal.setId("1");

        Circulo circuloClonado = (Circulo) circuloOriginal.clonar();
        System.out.println("Forma : " + circuloClonado.getTipo());
    }
}
```