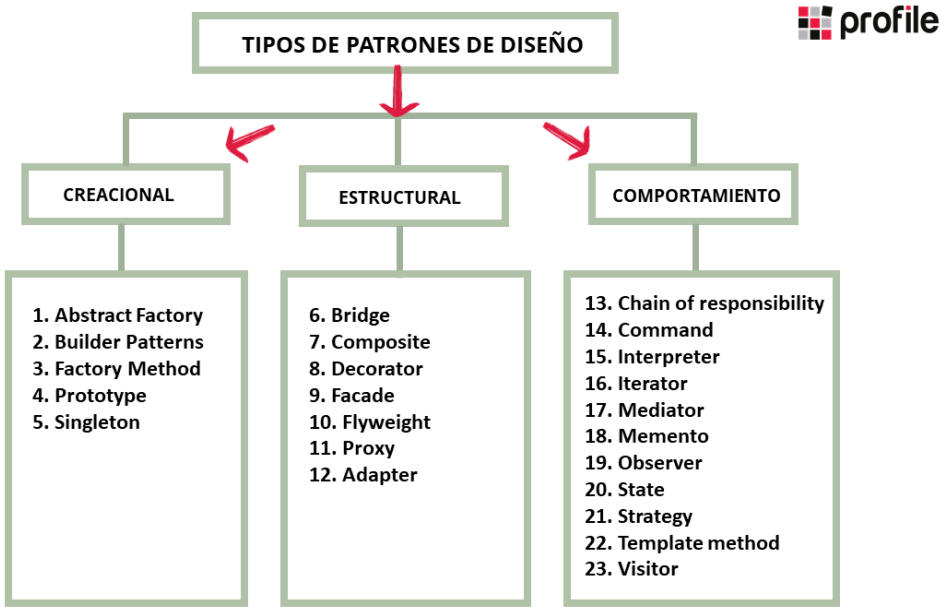
Patrones de diseño:



Los patrones de diseño son soluciones reutilizables a problemas comunes que ocurren en el diseño de software. Estos patrones representan las mejores prácticas que los desarrolladores pueden seguir para resolver problemas específicos en el desarrollo de software. Acá hay algunos ejemplos de los patrones de diseños mas comunes

Patrones Creacionales: Se enfocan en la forma en que se crean los objetos.

Singleton: Asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.

Factory Method: Define una interfaz para crear un objeto, pero permite a las subclases alterar el tipo de objeto que se creará.

Abstract Factory: Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

Builder: Separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción cree diferentes representaciones.

Prototype: Permite crear nuevos objetos copiando un objeto existente, conocido como prototipo.

Patrones Estructurales: Se ocupan de la composición de clases y objetos.

Adapter: Permite que clases con interfaces incompatibles trabajen juntas.

Decorator: Permite agregar responsabilidades adicionales a un objeto de manera dinámica.

Facade: Proporciona una interfaz simplificada a un conjunto de interfaces en un subsistema.

Composite: Permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme.

Proxy: Proporciona un sustituto o marcador de posición para otro objeto para controlar el acceso a él.

Patrones de Comportamiento: Se centran en la comunicación entre objetos.

Observer: Define una dependencia uno a muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

Strategy: Permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables.

Command: Encapsula una solicitud como un objeto, lo que permite parametrizar a los clientes con diferentes solicitudes, encolar o registrar solicitudes y soportar operaciones que se pueden deshacer.

State: Permite que un objeto altere su comportamiento cuando su estado interno cambia.

Template Method: Define el esqueleto de un algoritmo en una operación, diferiendo algunos pasos a las subclases.

Estos patrones ayudan a los desarrolladores a escribir código más flexible, reutilizable y mantenible

Ejemplo de patrones de diseño, en este caso usaremos el Singleton, que pertenece a los creacionales:

```java
// Clase Singleton
public class Singleton {
    // Variable estática privada para almacenar la única instancia
    private static Singleton instanciaUnica;

    // Constructor privado para evitar la creación de instancias desde fuera
    private Singleton() {
        // Inicialización de la instancia
    }

    // Método público estático para obtener la instancia única
    public static Singleton getInstancia() {
        if (instanciaUnica == null) {
            instanciaUnica = new Singleton();
        }
        return instanciaUnica;
    }

    // Otros métodos de la clase
    public void mostrarMensaje() {
        System.out.println("¡Hola desde Singleton!");
    }
}
```

Uso en la clase main

```java
public class Main {
    public static void main(String[] args) {
        // Obtener la única instancia de Singleton
        Singleton singleton = Singleton.getInstancia();

        // Usar la instancia
        singleton.mostrarMensaje();
    }
}
```

Como segundo ejemplo usando Factory Method

```java
// Interfaz Producto
public interface Producto {
    void usar();
}

// Clases concretas que implementan Producto
public class ProductoConcretoA implements Producto {
    @Override
    public void usar() {
        System.out.println("Usando ProductoConcretoA");
    }
}

public class ProductoConcretoB implements Producto {
    @Override
    public void usar() {
        System.out.println("Usando ProductoConcretoB");
    }
}
```

```java
// Clase Creador
public abstract class Creador {
    public abstract Producto crearProducto();
}

// Subclases de Creador
public class CreadorConcretoA extends Creador {
    @Override
    public Producto crearProducto() {
        return new ProductoConcretoA();
    }
}

public class CreadorConcretoB extends Creador {
    @Override
    public Producto crearProducto() {
        return new ProductoConcretoB();
    }
}
```

Uso del patron Factory Method en la clase main:

```
public class Main {
    public static void main(String[] args) {
        Creador creadorA = new CreadorConcretoA();
        Producto productoA = creadorA.crearProducto();
        productoA.usar();

        Creador creadorB = new CreadorConcretoB();
        Producto productoB = creadorB.crearProducto();
        productoB.usar();
    }
}
```

Tercer ejemplo usando patron de diseño Observer

```
// Interfaz Observador
public interface Observador {
    void actualizar(String mensaje);
}

// Clase Sujeto
import java.util.ArrayList;
import java.util.List;

public class Sujeto {
    private List<Observador> observadores = new ArrayList<>();

    public void agregarObservador(Observador observador) {
        observadores.add(observador);
    }

    public void eliminarObservador(Observador observador) {
        observadores.remove(observador);
    }

    public void notificarObservadores(String mensaje) {
        for (Observador observador : observadores) {
            observador.actualizar(mensaje);
```

```
    public void notificarObservadores(String mensaje) {
        for (Observador observador : observadores) {
            observador.actualizar(mensaje);
        }
    }
}

// Clases concretas que implementan Observador
public class ObservadorConcretoA implements Observador {
    @Override
    public void actualizar(String mensaje) {
        System.out.println("ObservadorConcretoA: " + mensaje);
    }
}

public class ObservadorConcretoB implements Observador {
    @Override
    public void actualizar(String mensaje) {
        System.out.println("ObservadorConcretoB: " + mensaje);
    }
}
```

Uso del patron observer en el main:

```
public class Main {
    public static void main(String[] args) {
        Sujeto sujeto = new Sujeto();

        Observador observadorA = new ObservadorConcretoA();
        Observador observadorB = new ObservadorConcretoB();

        sujeto.agregarObservador(observadorA);
        sujeto.agregarObservador(observadorB);

        sujeto.notificarObservadores("Evento ocurrido");
    }
}
```

Cuarto ejemplo usando el patron de diseño Decorator

```
// Interfaz Componente
public interface Componente {
    void operar();
}

// Clase ComponenteConcreto
public class ComponenteConcreto implements Componente {
    @Override
    public void operar() {
        System.out.println("Operación de ComponenteConcreto");
    }
}

// Clase Decorador
public abstract class Decorador implements Componente {
    protected Componente componente;

    public Decorador(Componente componente) {
        this.componente = componente;
    }
```

```
    public Decorador(Componente componente) {
        this.componente = componente;
    }

    @Override
    public void operar() {
        componente.operar();
    }
}

// Clases concretas que extienden Decorador
public class DecoradorConcretoA extends Decorador {
    public DecoradorConcretoA(Componente componente) {
        super(componente);
    }

    @Override
    public void operar() {
        super.operar();
        System.out.println("Operación adicional de DecoradorConcretoA");
    }
}
```

```
public class DecoradorConcretoB extends Decorador {
    public DecoradorConcretoB(Componente componente) {
        super(componente);
    }

    @Override
    public void operar() {
        super.operar();
        System.out.println("Operación adicional de DecoradorConcretoB");
    }
}
```

Uso del patron decorator en la clase main

```
public class Main {
    public static void main(String[] args) {
        Componente componente = new ComponenteConcreto();
        Componente decoradorA = new DecoradorConcretoA(componente);
        Componente decoradorB = new DecoradorConcretoB(decoradorA);

        decoradorB.operar();
    }
}
```

Ejemplo de patron de diseño Strategy

```java
// Interfaz Estrategia
public interface Estrategia {
    void ejecutar();
}

// Clases concretas que implementan Estrategia
public class EstrategiaConcretaA implements Estrategia {
    @Override
    public void ejecutar() {
        System.out.println("EstrategiaConcretaA ejecutada");
    }
}

public class EstrategiaConcretaB implements Estrategia {
    @Override
    public void ejecutar() {
        System.out.println("EstrategiaConcretaB ejecutada");
    }
}
```

```java
// Clase Contexto
public class Contexto {
    private Estrategia estrategia;

    public void setEstrategia(Estrategia estrategia) {
        this.estrategia = estrategia;
    }

    public void ejecutarEstrategia() {
        estrategia.ejecutar();
    }
}
```

Uso del patron Strategy en el main:

```java
public class Main {
    public static void main(String[] args) {
        Contexto contexto = new Contexto();

        contexto.setEstrategia(new EstrategiaConcretaA());
        contexto.ejecutarEstrategia();

        contexto.setEstrategia(new EstrategiaConcretaB());
        contexto.ejecutarEstrategia();
    }
}
```

Diferenciales principales:

Singleton:

Propósito: Asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.

Uso: Cuando se necesita exactamente una instancia de una clase y se debe proporcionar un punto de acceso global a esta instancia.

Ejemplo: Controladores de impresoras, registros de configuración.

Factory Method:

Propósito: Define una interfaz para crear un objeto, pero permite a las subclases alterar el tipo de objeto que se creará.

Uso: Cuando una clase no puede anticipar el tipo de objetos que debe crear.

Ejemplo: Frameworks que permiten a los usuarios extender sus componentes.

Observer:

Propósito: Define una dependencia uno a muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

Uso: Cuando un cambio en un objeto requiere cambiar otros objetos, y no se sabe cuántos objetos necesitan ser cambiados.

Ejemplo: Sistemas de eventos, interfaces de usuario.

Decorator:

Propósito: Permite agregar responsabilidades adicionales a un objeto de manera dinámica.

Uso: Para agregar funcionalidades a objetos individuales de manera flexible y sin afectar a otros objetos de la misma clase.

Ejemplo: Flujos de entrada/salida en Java.

Strategy:

Propósito: Permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables.

Uso: Cuando se tienen múltiples formas de realizar una operación y se quiere que el cliente elija el algoritmo a usar.

Ejemplo: Algoritmos de ordenamiento, métodos de compresión.