Programación Orientada a Objetos
Profesor: Rodolfo Baspineiro

Módulo 3: Patrones de diseño

Sesión 19: Introducción a Patrones de diseño
Noviembre 29 de 2021

Composición y herencia

Herencia:
```
public class Animal{
}
public class Cachorro extends Animal{
}
```

Desventajas:
- Encapsulamiento débil y acoplamiento estrecho, donde el cambio de una superclase puede afectar a todas las subclases.
- La herencia no permite que un objeto adopte una clase diferente.

Composición:
```
public class Sistema {
        Persona persona = new Persona();
        }
public class Persona{
}
```

Desventajas
- El software es dinámico y parametrizado y más difiícil de entender.

¿Cuándo usar Herencia?
En general, se prefiere usar la composición sobre la herencia. Se usa herencia:

- Si una instancia de una clase hija nunca necesitará convertirse en un objeto de otra clase.
- Si la jerarquía representa una relación “es una" en lugar de una relación "tiene una".
- Si se desea o necesita realizar cambios globales en las clases secundarias cambiando una clase principal.
- Cuando la clase secundaria se extiende en lugar de reemplazar total o parcialmente las responsabilidades de la clase principal.

Patrón de diseño
Describe una solución general reutilizable para un problema recurrente en el desarrollo de sistemas de software orientados a objetos. No es un código prefabricado, sino un modelo de cómo resolver un determinado problema. Los patrones de diseño definen las relaciones e interacciones entre clases u objetos, sin especificar los detalles de los involucrados.
Un patrón de diseño debe definir: un nombre, el problema, la solución, cuándo aplicar esa solución y las consecuencias de hacerlo.

Tipos de Patrones de Diseño:

Patrones creacionales:
- Abstraer el proceso de creación de objetos en una aplicación.
- Proporcionan interfaces para crear y copiar objetos.
- Produces diferentes representaciones usando el mismo código.

Patrones estructurales:
- Habilitan la colaboración entre objetos con interfaces incompatibles, agregando nuevos comportamientos.
- Permiten componer objetos en una estructura de árbol.

Patrones de comportamiento:
- Se encargan de las relaciones entre objetos y clases.
- Administran la distribución de responsabilidades en una aplicación.
- Algunos se basan en la herencia y otros en la composición.

Patrón de diseño Singleton
Singleton es un patrón de diseño que garantiza que una clase tenga una sola instancia y define un punto de acceso global para ella.

Ejemplo: Conexión a la base de datos:

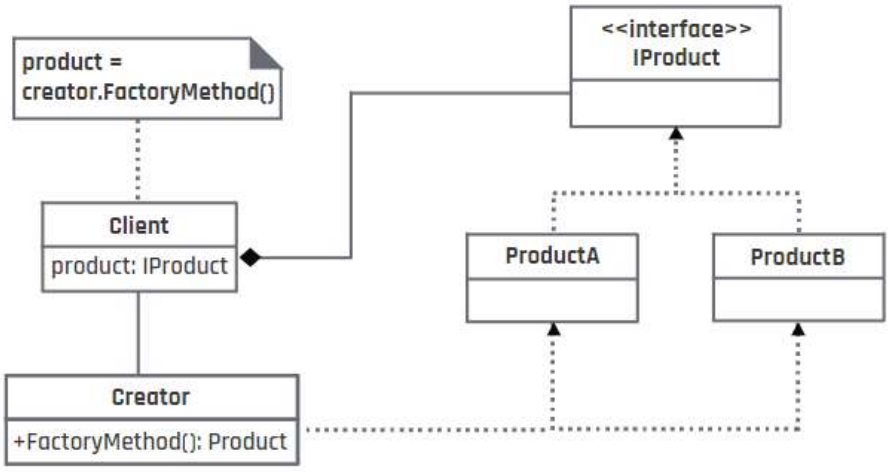
```
public class BaseDeDatos{

	//Atributo con el mismo nombre de la clase
	private static BaseDeDatos instance = new BaseDeDatos();

	//Construtor privado
	private BaseDeDatos(){
		/*Inicializaciones*/
	}

	//Método getInstance() estático
	public static BaseDeDatos getInstance(){
		return instance;
	}
}
```

Implementación alternativa:

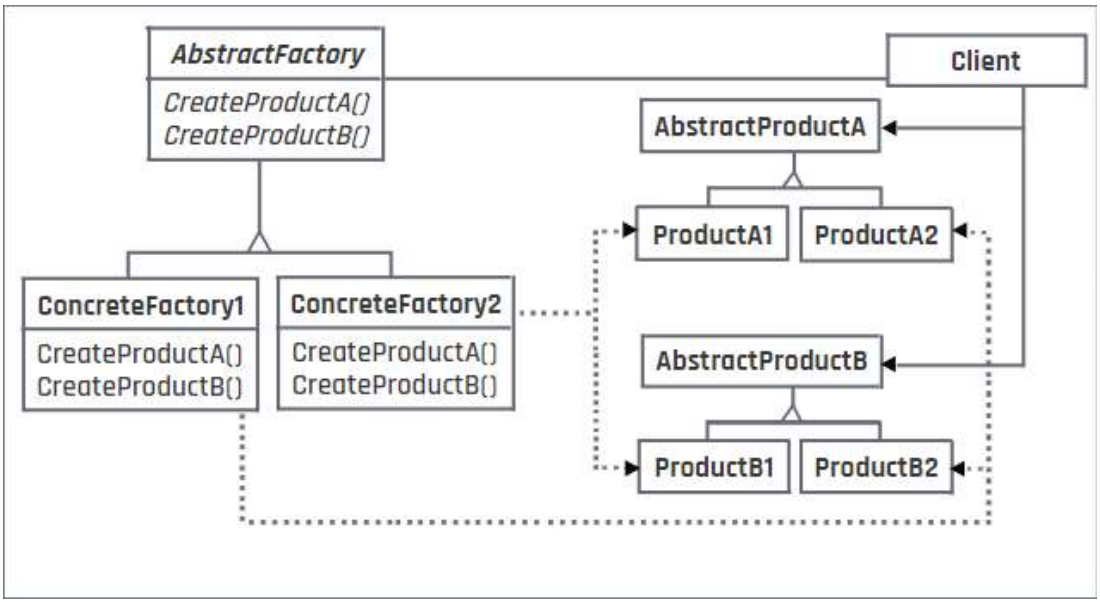
```
public class BaseDeDatos{

	//Atributo con el mismo nombre de la clase
	private static BaseDeDatos instance;

	//Construtor privado
	private BaseDeDatos() {
		/*Inicializaciones*/
	}

	//Método getInstance() estático
	// Inicialización tardía:  hasta que no se invoque al método getInstancia no se crea ningún objeto en memoria.
	public static BaseDeDatos getInstance() {
		if (instance == null)
			instance = new BaseDeDatos();
		return instance;
	}
}
```

Patrón de diseño Factory
Es un patrón de creación. Básicamente, la lógica de creación está encapsulada dentro de la fábrica (FactoryMethod) y se proporciona un método que devuelve un objeto (Metodo Factory predeterminado) o la creación del objeto se delega a una subclase (método Abstract Factory predeterminado). Tiene dos variantes:

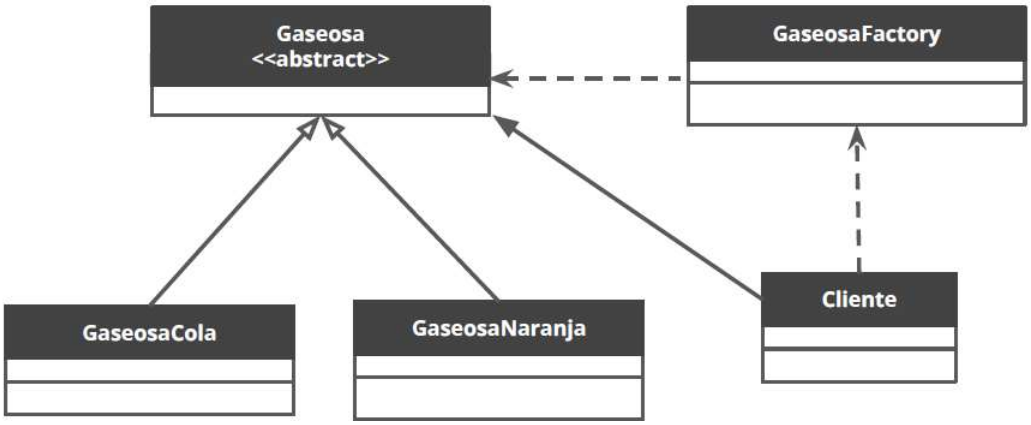
Factory Method
Es un patrón que define una interfaz para crear un objeto, pero permite que las subclases decidan qué clase instanciar.



Abstract Factory
Es un patrón que proporciona una interfaz para crear familias de objetos dependientes o relacionados sin especificar sus clases concretas.



Ejemplo Factory Method:




```
public abstract class Gaseosa {

	private String nombre;

	public String getNombre() {
		return nombre;
	}
	public void abrir() {
		System.out.println("Abriste una refrescante gaseosa de " +
	getNombre());
	}
}
```

```
public class GaseosaCola extends Gaseosa {

	String nombre = "Coca Cool";

	@Override
	public String getNombre() {
		return nombre;
	}
	@Override
	public void abrir() {
		super.abrir();
	}
}


public class GaseosaNaranja extends Gaseosa {

	String nombre = "Naranja dulce";

	@Override
	public String getNombre() {
		return nombre;
	}
	@Override
	public void abrir() {
		super.abrir();
	}
}


public class GaseosaFactory {
	public static Gaseosa construir(String tipo) {
		switch (tipo){
			case "Coca":
				return new GaseosaCola();
			case "Naranja":
				return new GaseosaNaranja();
			default:
				System.out.println("Ups, no encontramos este objeto para
				construir");
				return null;
		}
	}
}
```

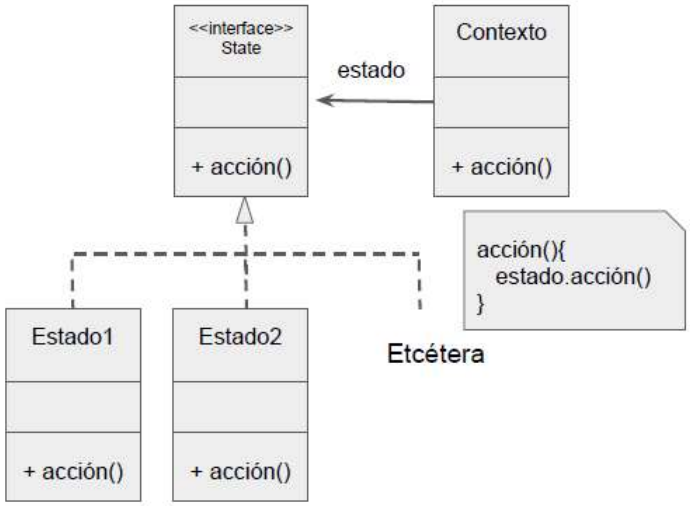
```
public static void main(String[] args) {
	try {
		Gaseosa gs1 = GaseosaFactory.construir("Coca");
		gs1.abrir();
		Gaseosa gs2 = GaseosaFactory.construir("Naranja");
		gs2.abrir();
		Gaseosa gs3 = GaseosaFactory.construir("Uva");
		gs3.abrir();
	}catch (Exception e){
		System.out.println("¡Exception encontrada!: " + e);
	}
}
```

Sincrónico

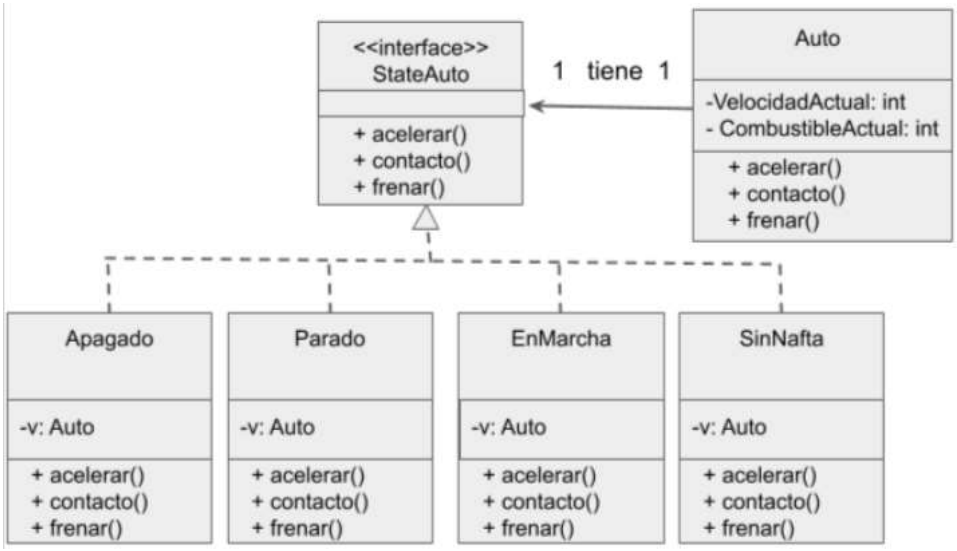
Sesión 20. Patrón State.
Diciembre 1 de 2021.

El Patrón State se utiliza cuando se requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra. El patrón State resuelve esta situación, creando básicamente un objeto por cada estado posible del objeto que lo invoca.
Así, se implementa una clase para cada estado diferente del objeto y cada clase implementará los métodos cuyo comportamiento varía según ese estado.

Ventajas:
- Se localizan fácilmente las responsabilidades de los estados, facilitando el desarrollo y mantenimiento del código.
- Facilita la ampliación de estados
- Permite al objeto cambiar de clase en tiempo de ejecución



Ejemplo de un modelo de auto:



.

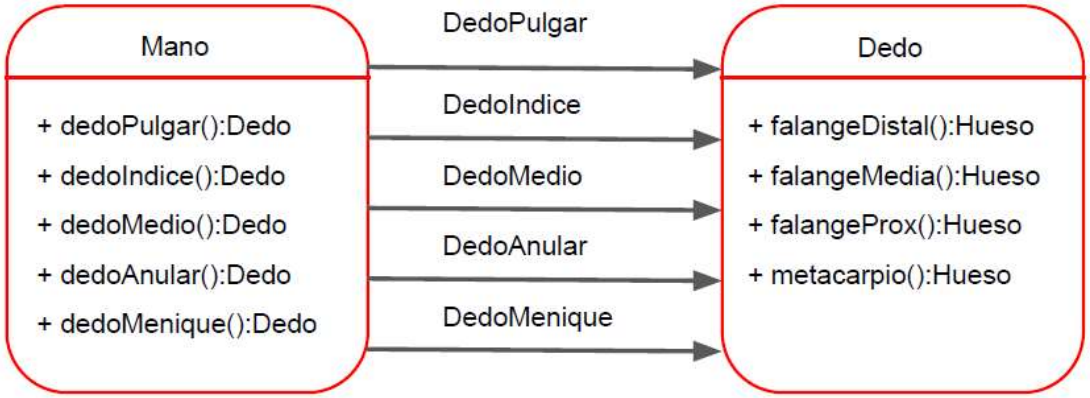
Pendiente: Realizar ejercicio del semáforo

Sincrónico

Sesión 22. Patrón Composite
Diciembre 6 de 2021

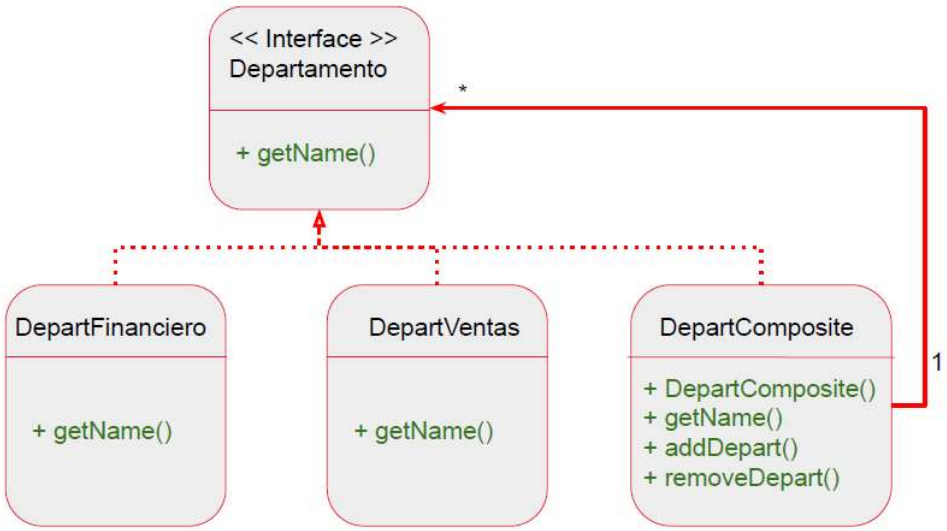
Composite

Patrón estructural que se enfoca en la forma en que los objetos se componen para formar estructuras compuestas.



Ejemplo

```
public interface Departamento {
	void getName();
}

public class DepartFinanciero implements Departamento {
	private int id;
	private String name;

	public void getName(){
		System.out.println(getClass().getSimpleName());
	}
}

public class DepartVentas implements Departamento {
	private int id;
	private String name;

	public void getName(){
		System.out.println(getClass().getSimpleName());
	}
}
```




```
public class DepartComposite implements Departmento {
	private int id;
	private String name;
	private List<Departmento> childDepartments;

	public DepartComposite(int id, String name) {
		this.id = id;
		this.name = name;
		this.childDepartments = new ArrayList<>();
	}
	public void getName() {
		childDepartments.forEach(Departmento::getName);
	}
	public void addDepart(Departmento department) {
		childDepartments.add(department);
	}
	public void removeDepart(Departmento department) {
		childDepartments.remove(department);
	}
}
```

Estructura de clases del patrón Composite

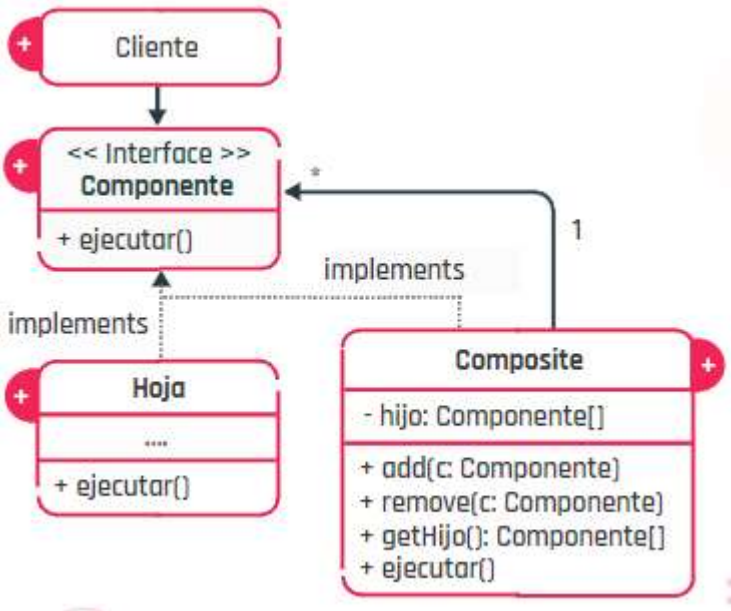
Cliente
Trabaja con todos los elementos a través de la interfaz componente.

Interfaz Componente
Describe las operaciones comunes tanto para elementos simples como para elementos complejos del árbol.

Hoja

Es un elemento básico de un árbol que no tiene sub-elementos.

Composite
El contenedor (o composite) es el elemento que contiene los subelementos hojas u otros contenedores. Un contenedor no sabe la clase concreta de sus hijos: trabaja con todos los subelementos a través de la interfaz componente.



Sincrónico

En el patrón composite, si se require definir atributos, no se utilizaría una Interfaz, sino una clase abstracta.

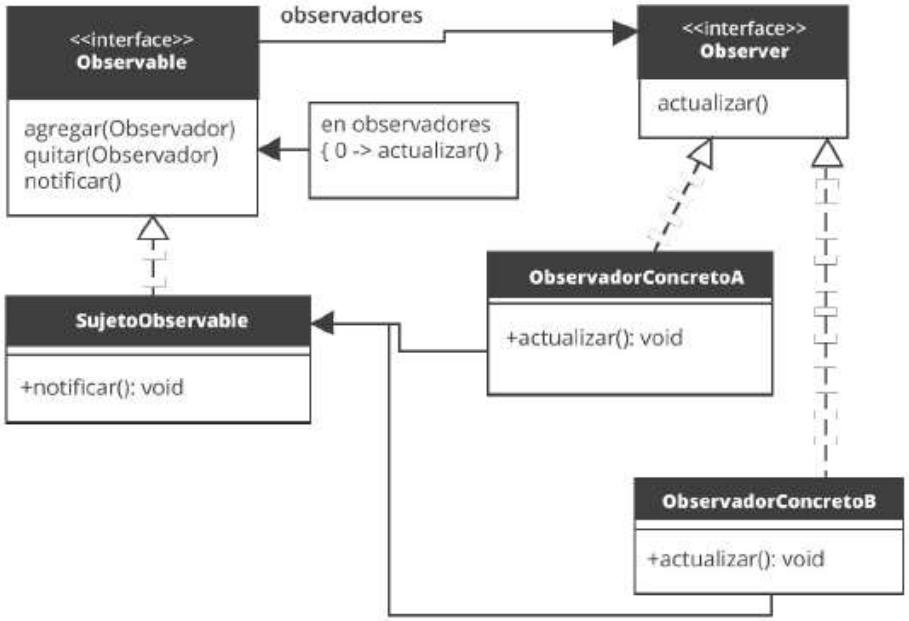
Sesión 23.
Diciembre 9 de 2021.

Relación de uso: línea punteada
No es necesario escribir la palabra “extends” para especificar la relación

Sesión 25. Patrón Observer.
Diciembre 15 de 2021

Patrón Observer

Este patrón establece una relación de Uno a Muchos. Se utiliza cuando se necesita que un objeto notifique a muchos otros objetos cuando se genera algún cambio específico.

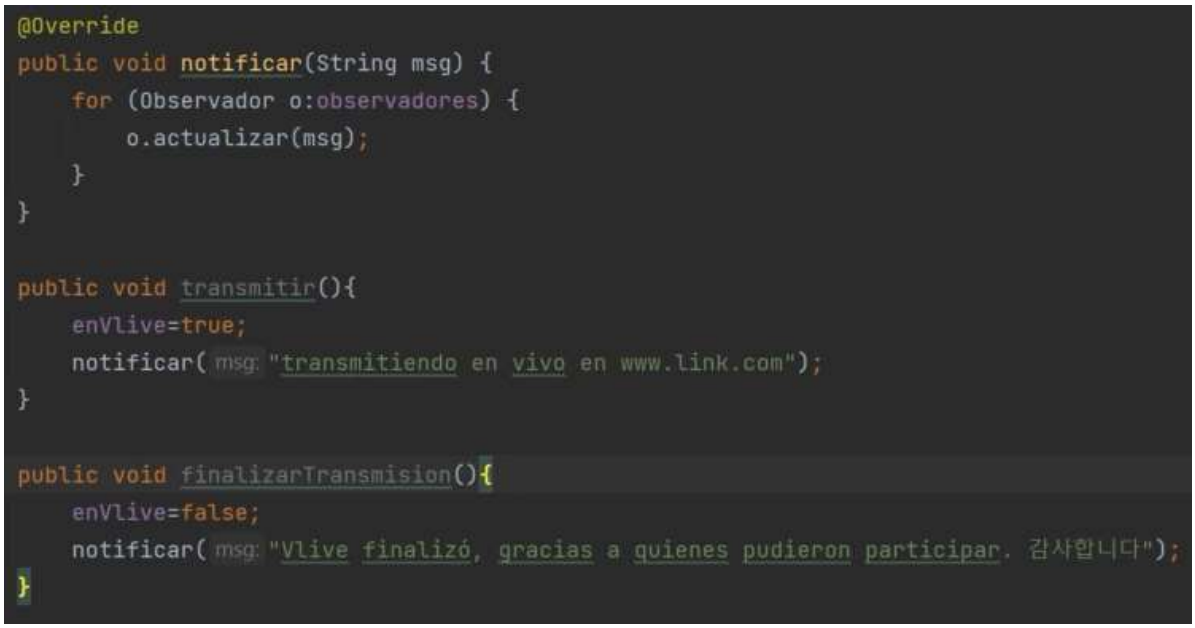


Ejemplo

```
// Interfaz Observable
public interface Observable {
	public void agregar(Observador o);
	public void quitar(Observador o);
	public void notificar(String cambio);
}

// Clase Pizarra
public class Pizarra implements Observable{

	// Cada implementación (sujeto) conoce a sus observadores
	private ArrayList<Observador> observadores = new ArrayList<>();private float
precioActual;

	// El sujetoenvía una notificación a sus observadores cuando cambia su
	estado.
	public void cambiarPrecio(float precio) {
		this.precioActual = precio;
		notificar(" precio actualizado a " + obtenerPrecio());
	}
```

```
	public float obtenerPrecio(){
		return precioActual;
	}
	// Y proporciona una interfaz para agregar o quitar observadores
	@Override
	public void agregar(Observador o) {
		this.observadores.add(o);
	}
	@Override
	public void quitar(Observador o) {
		this.observadores.remove(o);
	}
	@Override
	public void notificar(String cambio) {
		for(Observador o : observadores)
		System.out.println(o.actualizar() + cambio);
	}
}

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

// Clase Oro
public class Oro implements Observador{
	@Override
	public String actualizar() {
		return this + "> Cambio de estado: ";
	}
}

// Main
public class Main {
	public static void main(String[] args) {
		Pizarra pizarra = new Pizarra();
		Observador obs1 = new Oro();
		Observador obs2 = new Oro();

		pizarra.agregar(obs1);
		pizarra.agregar(obs2);

		pizarra.cambiarPrecio(42.5f); // Cada observador (Oro) es notificado y cada uno
		imprime el cambio
		pizarra.cambiarPrecio(44.3f); // Cada observador (Oro)  es notificado y cada uno
		imprime el cambio
	}
}
```

Cuando se requiere enviar diferentes notificaciones, se puede hacer pasar todas ellas por el método notificar. Ejemplo:



```java
@Override
public void notificar(String msg) {
    for (Observador o:observadores) {
        o.actualizar(msg);
    }
}

public void transmitir(){
    enVlive=true;
    notificar( msg: "transmitiendo en vivo en www.link.com");
}

public void finalizarTransmision(){
    enVlive=false;
    notificar( msg: "Vlive finalizó, gracias a quienes pudieron participar. 감사합니다");
}
```

.

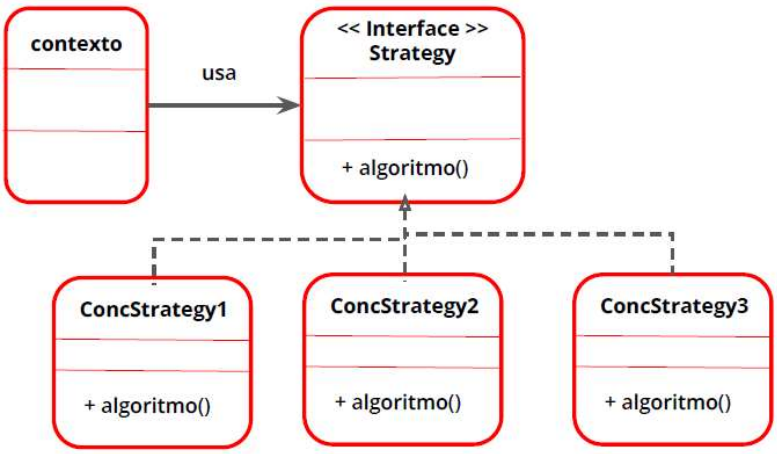
Sesión 26. Patrón strategy
Diciembre 15 de 2021

Patrón Strategy

Permite separar los diferentes comportamientos de una clase y así cambiar la estrategia para cuando sea preciso.
El patrón Strategy hace que los algoritmos varíen independientemente del cliente que los esté usando. Propone una solución simple basada en un objeto que cambia y cuyo comportamiento es el que se adapta a la circunstancia.

Ventajas

- Proporciona una alternativa a la herencias de clases, ya que puede realizarse un cambio dinámico de estrategia.
- Si un algoritmo utiliza información que no deberían conocer los clientes, la utilización del patrón Strategy evita la exposición de dichas estructuras.

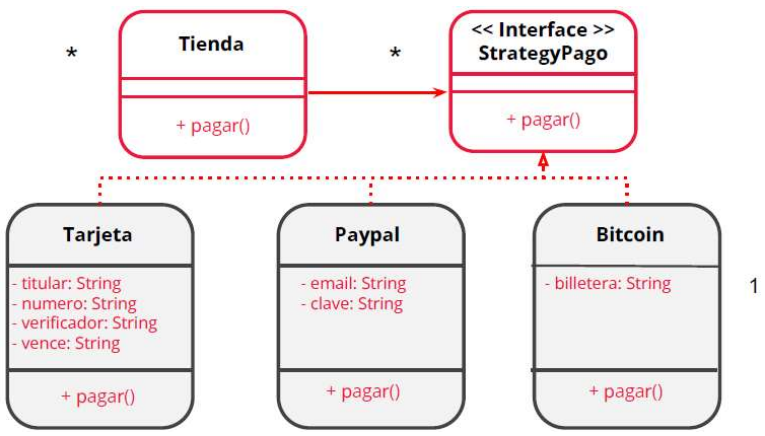


- Contexto: es el elemento que usa los algoritmos. Configura una estrategia concreta mediante una variable que referencia a la estrategia concreta necesaria. La variable que la contiene está definida como Strategy.
- Interface Strategy: declara una interface común para todos los algoritmos soportados.
- ConcStrategy: implementa el algoritmo utilizando la interfaz definida por la estrategia.

Ejemplo

```
// Interface StrategyPago
public interface StrategyPago {
        void pago();
}

// Clase Tarjeta
public class Tarjeta implements StrategyPago {
        private String titular;
        private String numero;
```

```
        private String verificador;
        private String vence;
        public void pago() {
                System.out.println("pagado con tarjeta");
        }
}

// Clase PayPal
public class Paypal implements StrategyPago {
        private String email;
        private String clave;
        public void pago() {
                System.out.println("pagado con paypal");
        }
}

// Clase Bitcoin
public class Bitcoin implements StrategyPago {
        private String billetera;
        public void pago(){
                System.out.println("pagado con bitcoin");
        }
}

// Clase Tienda
public class Tienda {
        private StrategyPago formaPago;
        public void pago() {
                formaPago.pago();
        }
        public void setPago(StrategyPago nuevoPago) {
                formaPago = nuevoPago;
        }
}
```



.