BackEndI
Profesor: Christian Ariel Díaz

Módulo 1. Patrones de Diseño

Sesión 1. Bienvenida
Mayo 16 de 2022

BackEnd
Algunas de las funciones que se gestionan en esta parte son:

- Las peticiones del front end.
- La lógica de negocio.
- Conexión con bases de datos (relacionales y no relacionales).
- Logueo de errores, para encontrar luego, más rápidamente las soluciones.
- Uso de librerías del servidor web, por ejemplo, para implementar temas de caché o para comprimir las imágenes de la web.
- La seguridad de los sitios web que gestiona
- Optimización de los recursos para que las páginas sean performantes.

Tests unitarios vs Test de Integración

Tests unitarios: Pruebas parciales
Los test o pruebas unitarias tiene por objetivo tomar una pequeña parte de software, aislándola del resto del código, para determinar si se funciona de la manera esperada. Cualquier dependencia del módulo bajo prueba debe sustituirse por un mock o un stub, para acotar la prueba específicamente a esa unidad de código.

Para llevar a cabo un correcto test unitario se debe seguir un proceso conocido como 3A:

Arrange: Se definen los requisitos que debe cumplir el código.
Act: Se ejecuta el test que da lugar a los resultados que se deben analizar.
Assert: Se comprueba si los resultados obtenidos son los esperados.

Ventajas de los tests unitarios:
- Facilitan los cambios en el código.
- Permiten encontrar bugs.
- Proveen documentación.
- Mejoran el diseño y la calidad del código. Invitan al desarrollador a pensar en el diseño antes de escribirlo (Test Driven Development – TDD).

Tests de Integración: Pruebas globales
Tiene por objetivo validar la interacción entre los módulos de software.

Anotaciones de JUnit:

@Test
Para generar un test unitario

@ParameterizedTest

@Disable
Para evitar que el test se ejecute

@BeforeEach
Se ejecuta antes de cada test

@AfterEach
Se ejecuta después de cada test

@BeforeAll
Se ejecuta una sola vez antes de todos los test unitarios

@AfterAll
Se ejecuta después de que terminan todos los tests unitarios

Assertions de JUnit
Se encuentran en org.junit.jupiter.api.Assertions

- assertArrayEquals:

- assertEquals:
Permite comparar si dos resultados son iguales

- assertTrue / assertFalse
Permite saber si el resultado es verdadero o falso

- assertNull / assertNotNull

- assertSame / assertNotSame

- assertAll

- assertIterableEquals

- assertThrows

- assertTimeout / assertTimeoutPreemptively

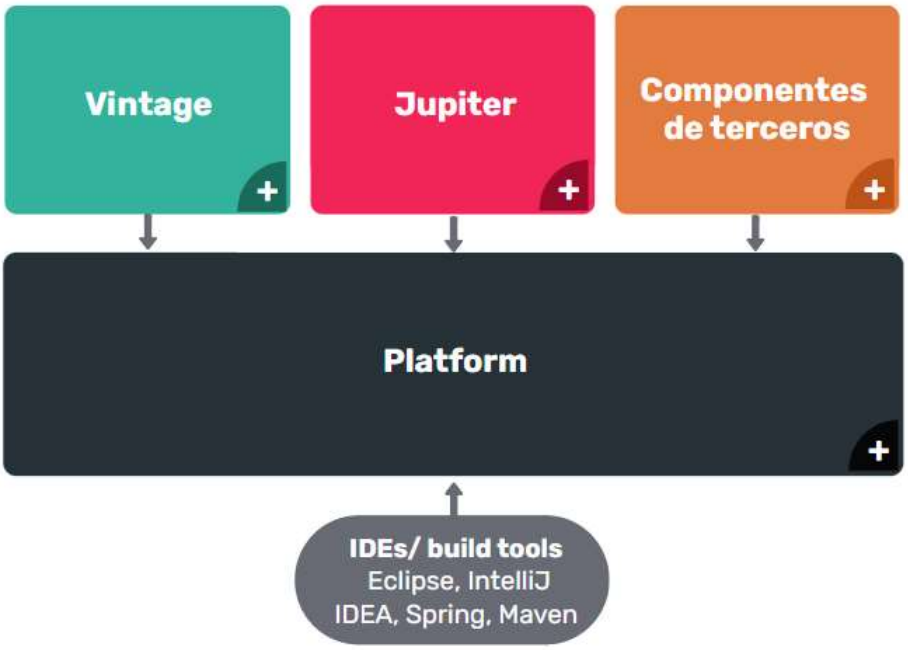
- assertLinesMatch

Principio FIRST
Características que deben tener los test unitarios:

- Fast
- Isolated / Independent
- Repeatable
- Self-validating
- Thorough

JUnit
Es el framework open-source de testing para Java más utilizado. Permite escribir y ejecutar tests automatizados. Es soportado por IDE como Eclipse e IntelliJ, build tolls como Maven o Gradle y por frameworks como Spring.

Arquitectura de JUnit5:

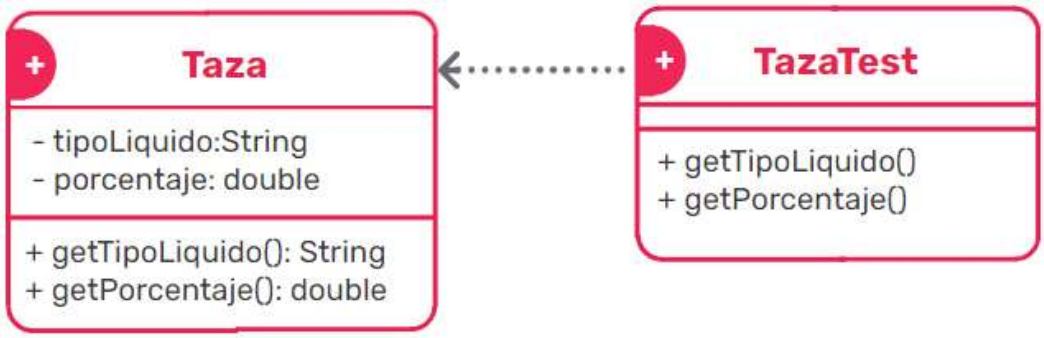


Vintage:
Contiene el motor de JUnit 3 y 4 para correr tests escritos en estas versiones.

Jupiter:
Es el componente que implementa el nuevo modelo d eprogramación y extensión: API para escribir tests y motor para ejecutarlos.

Platform:
Es un componente que actúa de ejecutor genérico de los tests.

Ejemplo:

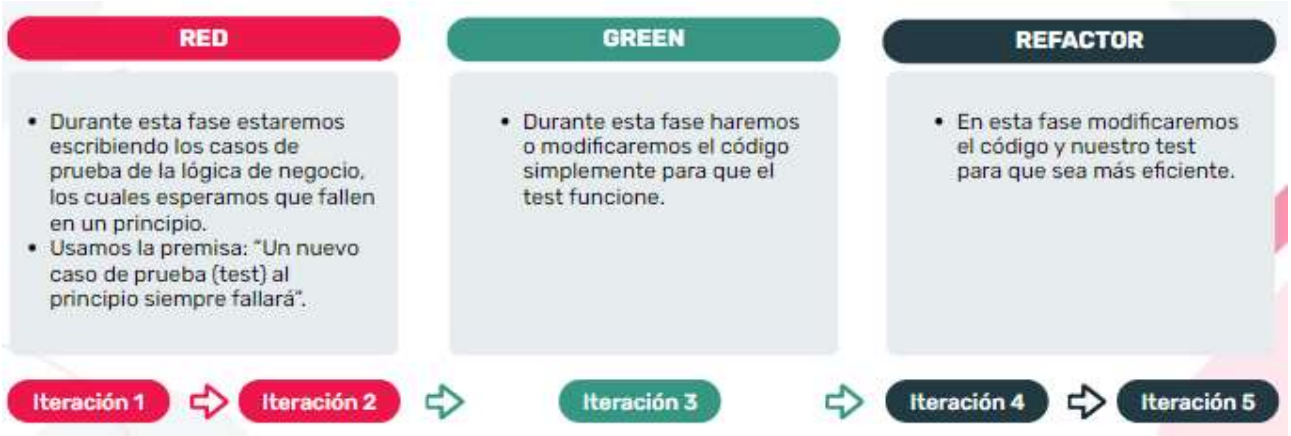



```
// Taza
public class Taza {
   private String tipoLiquido;
   private double porcentaje;
   public Taza(String tipoLiquido, double porcentaje) {
      this.tipoLiquido = tipoLiquido;
      this.porcentaje = porcentaje;
   }
   public String getTipoLiquido() {
      return tipoLiquido;
   }
   public void setTipoLiquido(String tipoLiquido)
      this.tipoLiquido = tipoLiquido;
   }
   public double getPorcentaje() {
      return porcentaje;
   }
   public void setPorcentaje(double porcentaje) {
      this.porcentaje = porcentaje;
   }
 }
```

```
//TazaTest:
import static org.junit.jupiter.api.Assertions.*;
 class TazaTest {
    @Test
    void getTipoLiquido() {
      Taza taza = new Taza("Jugo de Naranja", 70.5);
      assertEquals("Jugo de Naranja", taza.getTipoLiquido());
    }
    @Test
```

```
    void getPorcentaje() {
      Taza taza = new Taza("Jugo de Naranja", 70.5);
      assertEquals(70.5, taza.getPorcentaje());
    }
 }
```

TDD: Test Driven Development
Desarrollo guiado por pruebas. Se usan las pruebas (tests) para orientar o dirigir la forma en la que se escribe el código.



.
Iteración 1:

Código a testear
```
// No existe.
```

Test
```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

Iteración 2:

Código a testear
```
// Se busca que el código al menos compile
public class Validador {
  public static int esPar(int numero) {
```

```
      return 0;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

Iteración 3:

Código a testear

```
// Se hace lo estrictamente necesario para que el código pase el test.
public class Validador {
  public static int esPar(int numero) {
      return 1;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

Iteración 4:

Código a testear

```
public class Validador {
  public static boolean esPar(int numero) {
      return true;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

Iteración 5:

Código a testear

```
public class Validador {
  public static boolean esPar(int numero) {
      return numero%2 == 0;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

Testeo Parametrizado y Test Suite

Test sin parametrización:

```
import org.junit.Assert;
import org.junit.Test;

public class MultiplicarTest {
        @Test
        public void debemosCorroborarMultiplicaciones() {
                Assert.assertEquals(4, 2*2);
                Assert.assertEquals(6, 3*2);
```

```
            Assert.assertEquals(5, 5*1);
            Assert.assertEquals(10, 5*2);
        }
}
```

Tests parametrizados:

```
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import java.util.Arrays;

@RunWith(Parameterized.class)
public class MultiplicarTest {

        @Parameterized.Parameters
        public static Iterable data(){
            return Arrays.asList(new Object[][]{
                    {4,2,2},{6,3,2},{5,5,1},{10,5,2}
            });
        }

        private int multiplierOne;
        private int expected;
        private int multiplierTwo;

        public MultiplicarTest(int expected, int multiplierOne, int multiplierTwo) {
            this.multiplierOne = multiplierOne;
            this.expected = expected;
            this.multiplierTwo = multiplierTwo;
        }

        @Test
        public void debeMultiplicarElResultado(){
            Assert.assertEquals(expected,multiplierOne*multiplierTwo);
        }
}
```

Sincrónico

Base de datos: H2 (tipo SQL).

Sesión 2. Patrón Template Method

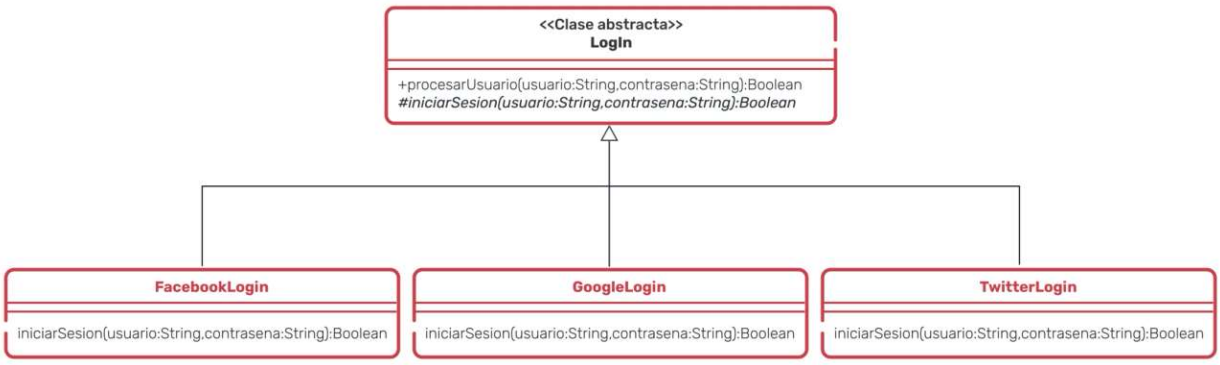
Mayo 16 de 2022

Patrón Template Method

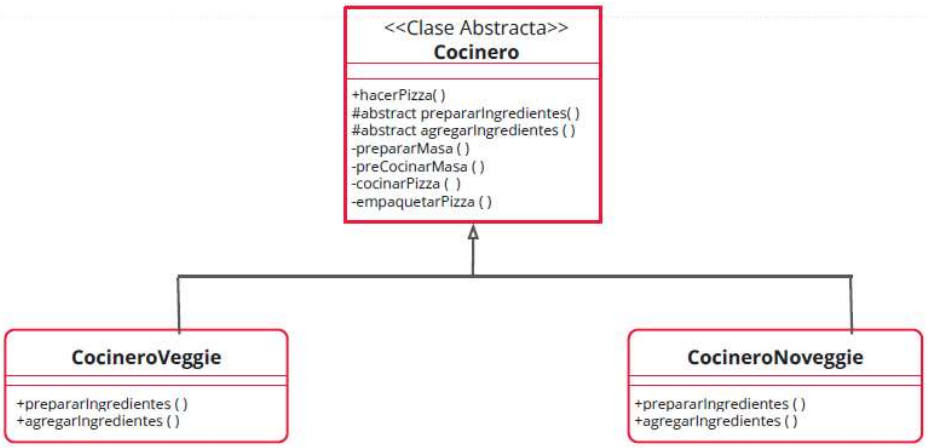
Propósito:
Es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase, pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

Solución:
El método esqueleto está conformado por el código que estas clases tienen en común, permitiendo que algunas partes puedan ser modificadas por la subclase que implemente el mismo, logrando ubicar en un solo lugar el código repetido. Al eliminar el código repetido, el código será más eficiente, legible y mantenible. Esto hará que sea más fácil de extender y mejorar.

Ejemplo:



Ejemplo



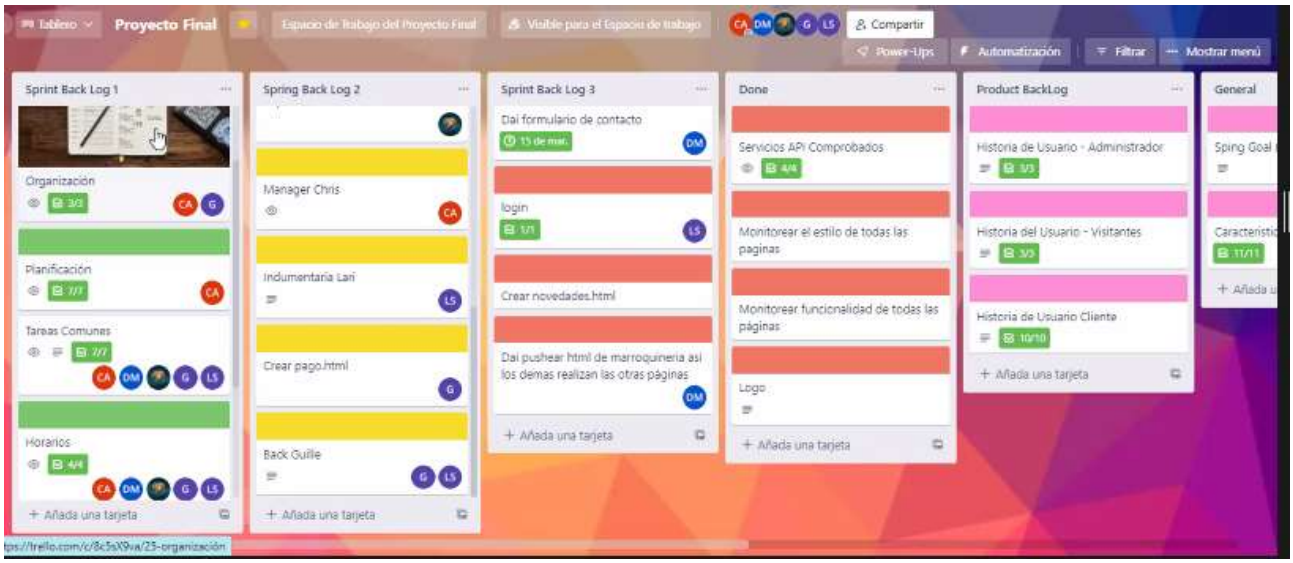
.

```
//Clase abstracta
public abstract class Cocinero {

	public void hacerPizza(){
		prepararMasa();
		preCocinarMasa();
		prepararIngredientes();
		agregarIngredientes();
		cocinarPizza();
		empaquetarPizza();
	}

	protected abstract void prepararIngredientes();
	protected abstract void agregarIngredientes();
	private void prepararMasa(){
		System.out.println("Preparando masa..");
	}
	private void preCocinarMasa(){
		System.out.println("Pre cocinando masa..");
	}
	private void cocinarPizza(){
		System.out.println("Enviando al horno la pizza");
	}
	private void empaquetarPizza(){
		System.out.println("Empaquetando pizza");
	}
}

//Código de la subclase
public class CocineroNoVeggie extends Cocinero {
	@Override
	protected void prepararIngredientes() {
		System.out.println("Preparando queso y jamón,");
	}
	@Override
	protected void agregarIngredientes() {
		System.out.println("Agregando los ingredientes");
	}
}

//Código de la subclase
public class CocineroVeggie extends Cocinero {
	@Override
	protected void prepararIngredientes() {
		System.out.println("Preparando tomate y quesos");
	}
	@Override
	protected void agregarIngredientes() {
		System.out.println("Agregando quesos y tomate");
```

```
	}
}

//Main
public class Main {
	public static void main(String[] args) {

		Cocinero cocineroVeggie = new CocineroVeggie();
		Cocinero cocineroNoVeggie = new CocineroNoVeggie();

		cocineroVeggie.hacerPizza();
		cocineroNoVeggie.hacerPizza();
	}
}
```

Sesión 3. Integradora 1
Mayo 17 de 2022

Sincrónico



Ejercicio
El objetivo de los empleados es conseguir afiliados (conseguirAfiliado) y hacer ventas (vender). Cada uno consigue 10 puntos por cada nuevo afiliado, 5 puntos por cada venta que realice y 5 puntos por cada año de antigüedad que posea.
El objetivo de los afiliados es hacer ventas (vender), pero el cálculo de los puntos es diferente: consiguen 15 puntos por cada nueva venta.
Los vendedores se categorizan de la siguiente manera:
Menos de 20 puntos = novatos.
Entre 20 y 30 puntos = aprendices.
Entre 31 y 40 puntos = buenos.
Más de 40 puntos = maestros.
Se deberá implementar un método mostrarCatetogia que primero deberá calcularPuntos, luego recategorizar de acuerdo al puntaje obtenido en el método anterior y por último mostrar por consola el nombre del vendedor el total de puntos y la categoría.
Realizar los casos de prueba necesarios para garantizar la calidad del método mostrarCategoría()
Si llegaste hasta acá, ¡felicitaciones!. Si aún te queda tiempo, te proponemos el siguiente desafío para seguir practicando.

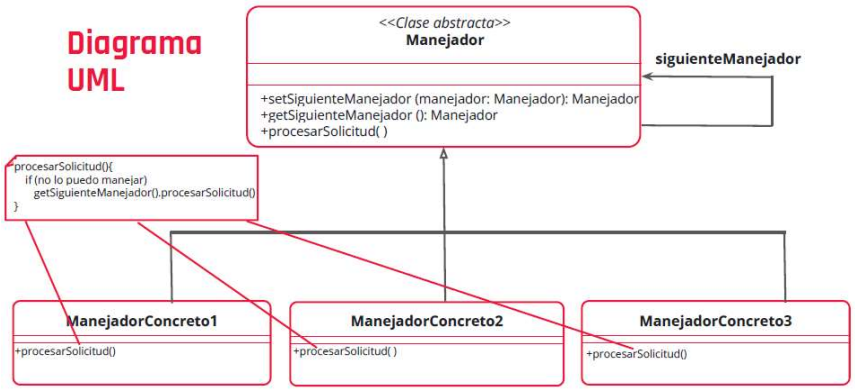
Sesión 4. Patrón cadena de responsabilidad
Mayo 18 de 2022

Patrón Cadena de Responsabilidad

Propósito

Es un patrón de diseño de comportamiento que permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada uno de ellos, decide si la procesa o la pasa al siguiente manejador.

Solución
Crear una cadena con las clases manejadores para que procesen la solicitud del cliente. Cada uno tiene un campo para almacenar una referencia al siguiente manejador de la cadena.
La solicitud viaja por la misma hasta que todos los manejadores hayan tenido la oportunidad de procesarla (los manejadores pueden decidir no pasar la solicitud y detener el procedimiento).



Ventajas:
- Mayor flexibilidad, Menor acoplamiento

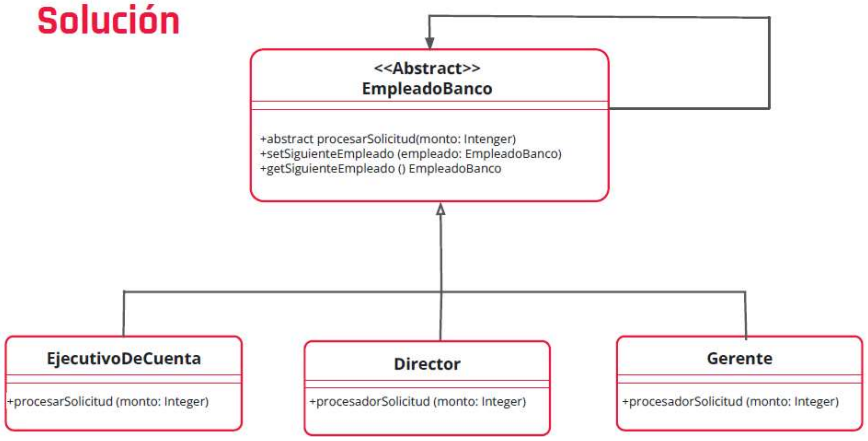
Desventajas:
- Puede ser complejo implementar la cadena.

En conclusión, es beneficioso utilizar el patrón cadena de responsabilidad cuando se espera que la aplicación procese diferentes tipos de solicitudes y responda con diferentes resultados, pero no se conoce de antemano cuáles son estas solicitudes.

Ejemplo

```
// Clase Manejadora
public abstract class EmpleadoBanco {

	private EmpleadoBanco sigEmpleadoBanco;
	public abstract void procesarSolicitud(Integer monto);

	public void setSigEmpleadoBanco(EmpleadoBanc emp) {
		sigEmpleadoBanco = emp;
	}
	public EmpleadoBanco getSigEmpleadoBanco() {
		return sigEmpleadoBanco;
	}
}
```

Solución




```
// Clase Director
public class Director extends EmpleadoBanco {

	@Override
	public void procesarSolicitud(Integer monto) {
		if (monto > 200000)
			System.out.println("Yo me encargo de gestionarlo.
			Director");
		else if (getSigEmpleadoBanco() != null)
			getSigEmpleadoBanco().procesarSolicitud(monto);
	}
}

// Clase EjecutivoCuenta
public class EjecutivoCuenta extends EmpleadoBanco {

	@Override
	public void procesarSolicitud(Integer monto) {
		if (monto < 60000)
			System.out.println("Yo me encargo de gestionarlo. Ejecutivo
			de cuenta");
		else if (getSigEmpleadoBanco() != null)
			getSigEmpleadoBanco().procesarSolicitud(monto);
	}
}

// Clase Gerente
public class Gerente extends EmpleadoBanco {

	@Override
	public void procesarSolicitud(Integer monto) {
		if (monto >= 60000 && monto <= 200000)
			System.out.println("Yo me encargo de gestionarlo.
			Gerente");
		else if (getSigEmpleadoBanco() != null)
			getSigEmpleadoBanco().procesarSolicitud(monto);
```

```
        }
}

//Main
public static void main(String[] args) {

        EmpleadoBanco empleado1 = new EjecutivoCuenta();
        EmpleadoBanco empleado2 = new Gerente();
        EmpleadoBanco empleado3 = new Director();

        empleado2.setSigEmpleadoBanco(empleado3);
        empleado1.setSigEmpleadoBanco(empleado2);
        empleado1.procesarSolicitud(78000);
}
```

Ejercicio

Pensemos en un gobierno que está compuesto por tres tipos de jerarquías: diputados, ministros y presidente. Queremos desarrollar un sistema de mensajería interno, en el que podamos enviarle documentos a los miembros gubernamentales. Además, queremos que los documentos recibidos, solo puedan ser leídos si están autorizados.

Un documento está compuesto por el contenido (String) y un tipo (Integer). Los posibles tipos de documento son valores numéricos: 1, 2 o 3. Cada número representa un nivel de acceso:

1 = Reservado
2 = Secreto
3 = Muy secreto

Los documentos de categoría “Reservado”, los leerán los diputados, los clasificados como “Secreto”, los ministros y por último, los categorizados como “Muy secreto” los leerá el presidente. También tengamos en cuenta que cada uno podrá leer los documentos que se encuentren clasificados en un orden menor al suyo. Por ejemplo, el presidente podrá leer todos los documentos, pero los diputados solo podrán leer los de categoría “Reservado”.

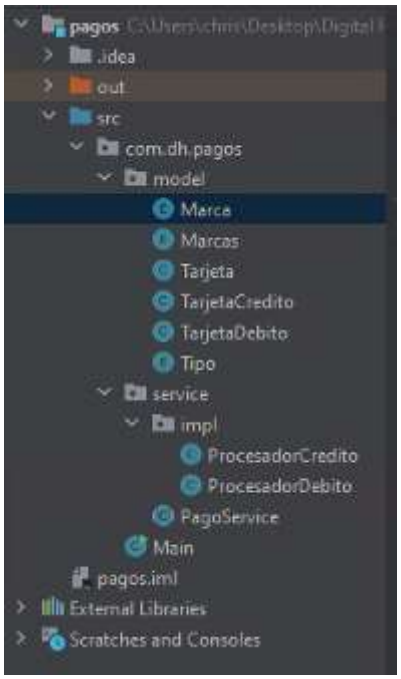
Pensemos a los diputados, ministros y presidentes como usuarios del sistema. Queremos que al momento en el que cualquiera de los tres tipos de usuario intente leer un documento (tenga o no acceso), lo envíe a otro tipo de usuario y así sucesivamente hasta que a todos los usuarios les llegue el documento. Los tres tipos de usuario deberán tener un método para leer el documento y un atributo que indicara que tipo de acceso tiene (1, 2 o 3). Este método recibirá un documento y según el tipo de jerarca que sea, evaluará si lo puede leer, en caso de que pueda, lo hará y además lo enviará a otro jerarca. En el caso de que no tenga acceso para leerlo, también lo enviará a otro tipo de jerarca. Te proponemos representar la solución en un diagrama UML e implementarlo en JAVA.

Sincrónico

Carpetas
- configurations
- controllers
- dtos
- models
- repository

Tipo de archivo: enum

public enum Tipo {
DEBITO,
CREDITO
}



Crear un enum para cada cosa.

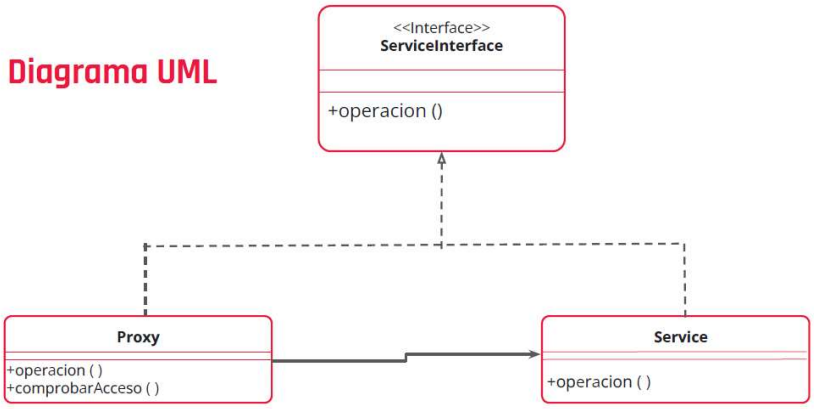
private Tipo Tipo;
(Lo que se está haciendo es crear un tipo de variable. Esto se hace en vez de usar un atributo de tipo String).

Package Services: Para ser consumidos por el FrontEnd
Package Repository: Para acceder a los datos

Sesión 5. Patrón Proxy
Mayo 19 de 2022

Propósito
Tiene por objetivo desarrollar la función de ser un intermediario que agrega funcionalidad a una clase, sin tocar la misma.

Solución:
Definir una clase Proxy con la misma interfaz que el objeto de servicio original. Después, se debe actualizar nuestra aplicación para que los clientes se comuniquen con el proxy y no con el servicio destino. Al recibir la solicitud de un cliente, el proxy le enviará al servicio, pero como intermediario podremos realizar operaciones antes o después de enviarle la solicitud.



Ventajas:
- El proxy funciona incluso si el objeto de servicio no está listo o no está disponible.
- Principio de abierto / cerrado: Se pueden introducir nuevos proxies sin cambiar el servicio o los clientes.

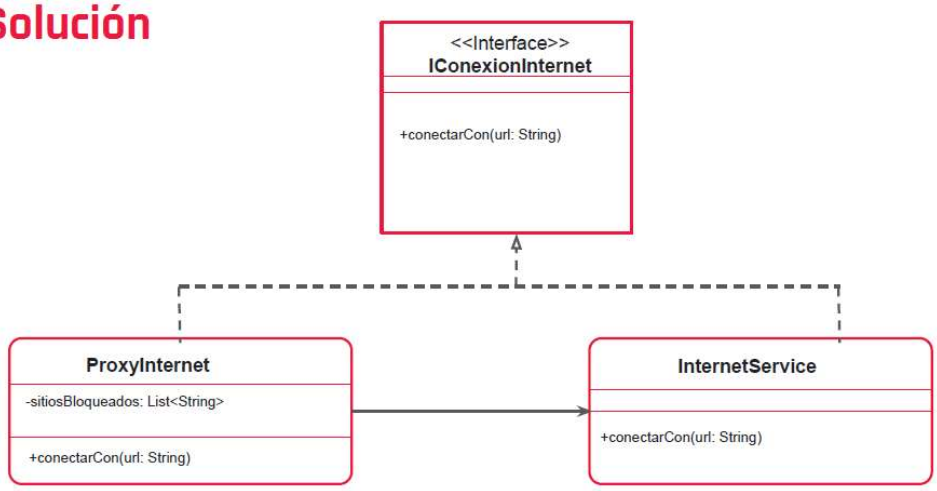
Desventajas:
- Al agregar una capa más entre el cliente y el servicio real, la respuesta puede retrasarse.

Ejemplo

```
public interface IConexionInternet {
	public void conectarCon(String url);
}

public class InternetService implements IConexionInternet {
	@Override
	public void conectarCon(String url) {
		System.out.println("Conectando con "+url);
	}
}
```

```
public class ProxyInternet implements IConexionInternet {

	private InternetService internetService;
	private List<String> sitiosBloqueados;

	public ProxyInternet(List<String> sitiosBloqueados, InternetService
	internetService) {
		this.sitiosBloqueados = sitiosBloqueados;
		this.internetService = internetService;
	}

	@Override
	public void conectarCon(String url) {
		if(!this.sitiosBloqueados.contains(url))
			this.internetService.conectarCon(url);
		else
			System.out.println("Acceso denegado");
	}
}
```

Solución




```
public class Main {

	public static void main(String[] args) {
		List<String> sitiosBloqueados = new ArrayList<>();
		sitiosBloqueados.add("www.youtube.com");
		sitiosBloqueados.add("www.facebook.com");
		IConexionInternet proxy;
		proxy = new ProxyInternet(sitiosBloqueados, new
		InternetService());
		proxy.conectarCon("www.google.com");
		proxy.conectarCon("www.youtube.com");
	}
}
```

Ejercicio
Pensemos en una aplicación al estilo Google Drive, un servicio que nos trae documentos. Para acceder al mismo, debemos enviarle una url y un email. Los documentos están compuestos por una id, una url, un contenido y una lista de usuarios autorizados a verlo. Queremos registrar quiénes acceden a los documentos. ¿Cómo resolverías este problema aplicando el patrón proxy?

Sesión 6.
Mayo 20 de 2022

Sincrónico

Sesión 7. Patrón flyweight
Mayo 23 de 2022

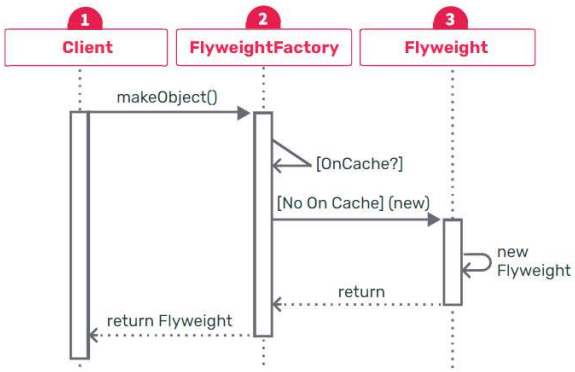
Propósito
Abstrae las partes reutilizables de un objeto y, en lugar de crear objetos cada vez que sea requerido, reutiliza objetos creados por otras instancias. Esto permite reducir la capacidad de memoria requerida por la aplicación.

Solución
Este patrón cuenta con varios componentes:

Cliente:
Objeto que dispara la ejecución.

FlyweightFactory:
Fábrica utilizada para crear los objetos flyweight u objetos ligeros.

Flyweight:
objetos que se desea reutilizar para que sean más ligeros.

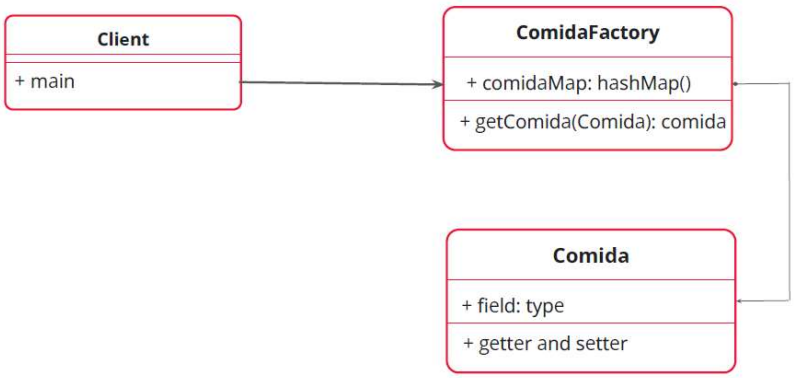


Este patrón es utilizado cuando la optimización de los recursos es algo primordial, ya que elimina la redundancia de objetos con propiedades idénticas.

Ejemplo

```
import java.util.HashMap ;

public class ComidaFactory {
        private static final HashMap<String, Comida> comidaMap = new
        HashMap() ;
```

```
        public static Comida getComida (String tipoComida) {
                Comida comida = (Comida) comidaMap.get(tipoComida);

                if (comida == null) {
                        comida = new Comida(tipoComida);
                        comidaMap.put(tipoComida, comida);
                        System.out.println( "Creando objecto comida de tipo: "+
                        tipoComida) ;
                }
                return comida;
        }
}

public class Comida {
        private String tipoComida ;
        private int precio;
        private boolean tieneLechuga ;

        public Comida(String tipoComida) {
                this.tipoComida = tipoComida;
        }
        public String getTipoComida () {
                return tipoComida;
        }
        public int getPrecio() {
                return precio;
        }
        public void setPrecio(int precio) {
                this.precio = precio;
        }
        public boolean isTieneLechuga() {
                return tieneLechuga;
        }
        public void setTieneLechuga(boolean tieneLechuga) {
                this.tieneLechuga = tieneLechuga;
        }
        public void descripcionDeLaComida (){
                System. out.println( "Es un/una " + getTipoComida() + " que sale:
                " + getPrecio()) ;
        }
}
```



Ejercicio
Pensemos en un sistema para agregar canciones a una lista de reproducción.

Una canción está compuesta por:
- Nombre.
- Artista.
- Género.

Una lista de reproducción está compuesta por:
- Nombre.

Para agregar una canción es necesario crear la lista de reproducción. Se podrá eliminar y obtener canciones.

CancionFactory será el lugar donde se almacenarán las canciones. Permitirá, antes de crear el objeto, validar si ya existe uno idéntico al que se le está solicitando. De ser así, regresa el objeto existente; de no existir, crea el nuevo objeto y lo almacena en caché para ser reutilizado más adelante.

También nos gustaría poder ver por consola las listas de reproducción.

Te proponemos realizar diagrama UML e implementación en JAVA.

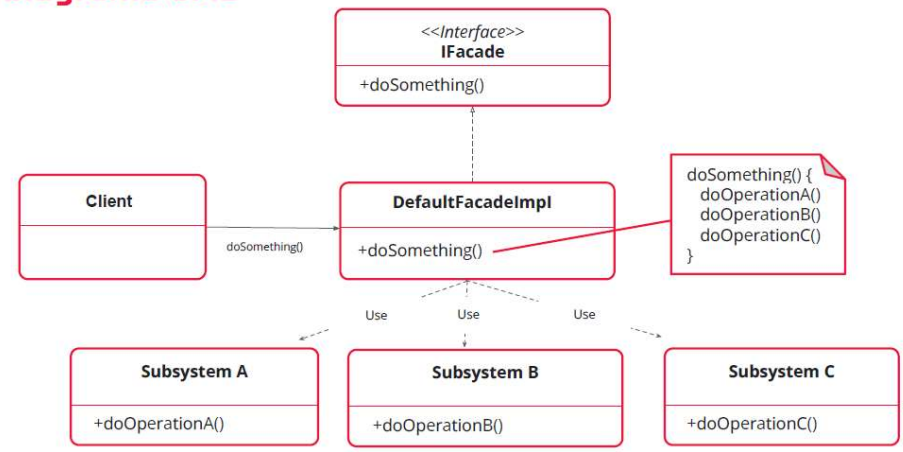
Sincrónico

Sesión 8. Patrón Facade
Mayo 23 de 2022

Propósito
Facade es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

Solución
Se dispone de una interfaz que define cómo el cliente se va a comunicar con el sistema. La clase que implementa esta interfaz recibe las peticiones y será la encargada de enviarle la petición del cliente a la clase que corresponda (subsistemas).

Diagrama UML

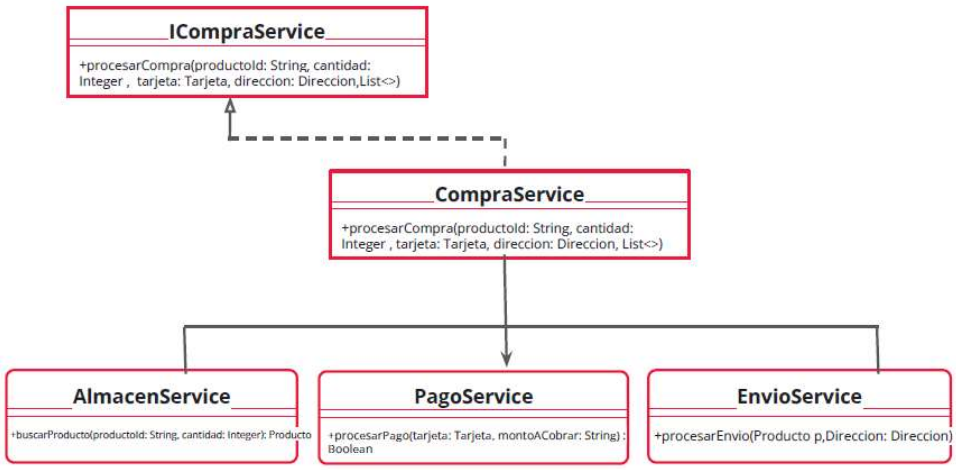


Ventajas:
- El software se vuelve más flexible y fácil de expandir.
- Se reduce el uso de objetos que tratan directamente con el cliente.
- Se reduce el acoplamiento entre el cliente y los subsistemas.

Desventajas:
- Alto grado de dependencia en la interfaz de la fachada.

El patrón facade ayuda a reducir la complejidad de interactuar con un conjunto de subsistemas, actuando de intermediario entre el cliente y los subsistemas.

Ejemplo
Supóngase que se requiere que diseñar un sistema para un e-commerce. El cliente requiere que al momento de efectuar la venta del producto, el sistema debería realizar una serie de pasos, por ejemplo: pedir el producto al almacén, acreditar el pago y enviar el pedido.




```
//Interface ICompraService
public interface ICompraService {
	public void procesarCompra(String productoId, Integer cantidad, Tarjeta
	tarjeta, Direccion direccion, List<Producto> productos);
}

//Clase CompraService, implementa la interfaz
public class CompraService implements ICompraService {
	private AlmacenService almacenService;
	private PagoService pagoService;
	private EnvioService envioService;

	public CompraService() {
		this.almacenService = new AlmacenService();
		this.pagoService = new PagoService();
		this.envioService = new EnvioService();
	}

	public void procesarCompra(String productoId, Integer cantidad, Tarjeta
	tarjeta, Direccion direccion, List<Producto> productos) {
		Producto prod;
		almacenService.setProductos(productos);
		prod = almacenService.buscarProducto(productoId, cantidad);
		if(prod != null){
			double montoCobrar = producto.getValor() * cantidad;
			if(pagoService.procesarPago(tarjeta, montoCobrar)){
				envioService.procesarEnvio(producto, direccion);
			}
		}
	}

// Clase AlmacenService
public class AlmacenService {
	private List<Producto> productos;
	public void setProductos(List<Producto> productos) {
```

```
            this.productos = productos;
        }
        public Producto buscarProducto(String productoId, Integer
        cantidad) {
            Producto producto = null;
            for (Producto p : this.productos) {
                if (p.getProductoId().equals(productoId) && p.getCantidad()
                >= cantidad){
                    producto = p;
                    p.setCantidad(p.getCantidad() - cantidad);
                    producto.setCantidad(cantidad);
                }
            }
            return producto;
        }
}

// Clase PagoService
Public class PagoService {
        public Boolean procesarPago(Tarjeta tarjeta, double montoACobrar){
            Boolean pagoRealizado = Boolean.FALSE;
            if(tarjeta != null && tarjeta.getNumerosFrente() != null &&
            tarjeta.getCodSeguridad() != null)
            System.out.println("Procesando el pago por "+ montoACobrar);
            pagoRealizado = TRUE;
            return pagoRealizado;
        }
}

// Clase EnvioService
public class EnvioService {
        public void procesarEnvio(Producto producto, Direccion direccion){
            System.out.println("Enviando producto a " + direccion.getCalle()
            +" "+ direccion.getNro() +","+ direccion.getBarrio());
        }
}

// Main
public class Prueba {
        public static void main(String[] args) {
            List<Producto> productos = new ArrayList<>();
            Producto productoUno = new Producto("1", 5, 1000, "Mouse");
            Producto productoDos = new Producto("2", 5, 3000, "Teclado");
            productos.add(productoUno);
            productos.add(productoDos);
            Tarjeta tarjeta = new Tarjeta("1111222233334444", "012",
            "2025/07/09");
            Direccion direccion = new Direccion("Av Monroe", "860", "1428",
            "CABA", "Capital federal");
```

```
            ICompraService compraService = new CompraService();
            compraService.procesarCompra("1", 2, tarjeta, direccion,
            productos);
        }
}
```

Ejercicio
Imaginate que tenés que aplicar el patrón facade en un sistema para retirar dinero en el cajero. Contás con tres servicios (subsistemas) disponibles para usar: uno se encargará de la autenticación (AutenticationService), otro de traer los datos de la cuenta bancaria (CuentaService) y el último de solicitar al banco el efectivo (CajaService). Queremos que el cliente pueda retirar dinero enviando al sistema el número de identificación, contraseña y saldo a retirar, sin tener que utilizar directamente estos servicios.

El servicio de autenticación, cuenta con un método llamado validarUsuarioYContrasena, que recibirá un identificador del usuario (DNI) y contraseña.

El servicio que nos trae la información de la cuenta tiene un método llamado getCuenta, que recibirá solamente el identificador del usuario (DNI). Este servicio lo deberíamos usar solamente luego de utilizar el servicio de autenticación. La cuenta está compuesta por un campo de identificación del usuario y otro de saldo actual.

Y por último tenemos el servicio de CAJA, que es el encargado de retirar dinero del banco. Este servicio tiene un método llamado entregarDinero, que recibirá el monto a retirar del banco.

Luego de verificar que el usuario tiene el dinero que quiere retirar, deberíamos usar el servicio de caja. Te proponemos realizar diagrama de clases e implementación en JAVA. ¡Éxitos!

Sesión 9. Integradora III
Mayo 24 de 2022