# Semana 8 - Patrones Adapter y Bridge

## 1. Introducción a los Patrones Estructurales

Los patrones estructurales se centran en cómo las clases y objetos se componen para formar estructuras más grandes. Esta semana nos enfocamos en:

* **Adapter:** Permite que interfaces incompatibles trabajen juntas.
* **Bridge:** Separa la abstracción de su implementación, permitiendo que ambas evolucionen de forma independiente.

## 2. Patrón Adapter

### a) Definición

El patrón **Adapter** convierte la interfaz de una clase en otra que el cliente espera. Permite la colaboración entre clases que, de otra forma, serían incompatibles.

### b) ¿Cuándo utilizarlo?

* Cuando se quiere reutilizar una clase existente cuya interfaz no coincide con la esperada.
* Para integrar librerías o sistemas antiguos en nuevas aplicaciones.

### c) Antipatrones del Adapter

* Usar Adapter para encubrir un mal diseño inicial.
* Adaptar interfaces sin validar su compatibilidad lógica (puede producir errores de comportamiento).

## 3. Patrón Bridge

### a) Definición

El patrón **Bridge** desacopla una abstracción de su implementación, de modo que ambas puedan variar independientemente. Se usa principalmente para evitar una explosión de clases debido a combinaciones de herencia.

### b) ¿Cuándo utilizarlo?

* Cuando se tienen múltiples variantes de una misma jerarquía.
* Cuando se desea cambiar las implementaciones en tiempo de ejecución.

### c) Antipatrones del Bridge

* Sobreingeniería: usar Bridge cuando no hay variantes que lo justifiquen.
* Duplicación de jerarquías innecesarias.

## 4. Ejercicios con Historias de Usuario

### Ejercicio 1: Adapter en una Plataforma de Pagos

#### Historia de Usuario:

Como desarrollador de una tienda en línea, quiero integrar un nuevo sistema de pagos externo cuya interfaz es diferente a la utilizada actualmente, para no modificar todo el sistema existente.

#### Código Java:

// Interfaz esperada

interface ProcesadorPago {

void pagar(double monto);

}

// Clase existente de terceros

class SistemaExternoPago {

public void realizarPago(double cantidad) {

System.out.println("Pago realizado por: $" + cantidad);

}

}

// Adapter

class AdaptadorPago implements ProcesadorPago {

private SistemaExternoPago sistema;

public AdaptadorPago(SistemaExternoPago sistema) {

this.sistema = sistema;

}

public void pagar(double monto) {

sistema.realizarPago(monto);

}

}

// Cliente

public class Main {

public static void main(String[] args) {

ProcesadorPago pago = new AdaptadorPago(new SistemaExternoPago());

pago.pagar(150.0);

}

}

### Ejercicio 2: Bridge para Gestión de Notificaciones

#### Historia de Usuario:

Como desarrollador de una plataforma de servicios, necesito enviar notificaciones por diferentes canales (Email, SMS) desde diferentes servicios (orden, facturación), sin duplicar código.

#### Código Java:

// Implementación

interface CanalNotificacion {

void enviar(String mensaje);

}

class Email implements CanalNotificacion {

public void enviar(String mensaje) {

System.out.println("Email enviado: " + mensaje);

}

}

class SMS implements CanalNotificacion {

public void enviar(String mensaje) {

System.out.println("SMS enviado: " + mensaje);

}

}

// Abstracción

abstract class Notificador {

protected CanalNotificacion canal;

public Notificador(CanalNotificacion canal) {

this.canal = canal;

}

public abstract void notificar(String mensaje);

}

class NotificadorOrden extends Notificador {

public NotificadorOrden(CanalNotificacion canal) {

super(canal);

}

public void notificar(String mensaje) {

canal.enviar("Orden: " + mensaje);

}

}

class NotificadorFactura extends Notificador {

public NotificadorFactura(CanalNotificacion canal) {

super(canal);

}

public void notificar(String mensaje) {

canal.enviar("Factura: " + mensaje);

}

}

// Cliente

public class Main {

public static void main(String[] args) {

Notificador noti1 = new NotificadorOrden(new Email());

Notificador noti2 = new NotificadorFactura(new SMS());

noti1.notificar("Tu pedido ha sido enviado.");

noti2.notificar("Tu factura está lista.");

}

}

### Ejercicio 3: Adapter para Sistema de Envío de Correos Legacy

#### Historia de Usuario:

Como ingeniero de soporte, necesito que la nueva plataforma de marketing utilice el sistema antiguo de envío de correos, que tiene una interfaz distinta a la nueva.

#### Código Java:

interface EnviadorCorreo {

void enviarCorreo(String destino, String contenido);

}

class CorreoLegacy {

public void enviarMensaje(String texto, String email) {

System.out.println("Enviando a " + email + ": " + texto);

}

}

class AdaptadorCorreo implements EnviadorCorreo {

private CorreoLegacy correo;

public AdaptadorCorreo(CorreoLegacy correo) {

this.correo = correo;

}

public void enviarCorreo(String destino, String contenido) {

correo.enviarMensaje(contenido, destino);

}

}

public class Main {

public static void main(String[] args) {

EnviadorCorreo servicio = new AdaptadorCorreo(new CorreoLegacy());

servicio.enviarCorreo("cliente@ejemplo.com", "Bienvenido a la plataforma");

}

}

### Ejercicio 4: Bridge para Renderizado de Figuras

#### Historia de Usuario:

Como desarrollador de un software gráfico, necesito representar diferentes figuras (círculo, rectángulo) que puedan ser renderizadas con distintas tecnologías (OpenGL, DirectX), sin generar múltiples combinaciones de clases.

#### Código Java:

interface Renderizador {

void renderizar(String figura);

}

class OpenGL implements Renderizador {

public void renderizar(String figura) {

System.out.println("Renderizando " + figura + " con OpenGL");

}

}

class DirectX implements Renderizador {

public void renderizar(String figura) {

System.out.println("Renderizando " + figura + " con DirectX");

}

}

abstract class Figura {

protected Renderizador renderizador;

public Figura(Renderizador renderizador) {

this.renderizador = renderizador;

}

public abstract void dibujar();

}

class Circulo extends Figura {

public Circulo(Renderizador renderizador) {

super(renderizador);

}

public void dibujar() {

renderizador.renderizar("Círculo");

}

}

class Rectangulo extends Figura {

public Rectangulo(Renderizador renderizador) {

super(renderizador);

}

public void dibujar() {

renderizador.renderizar("Rectángulo");

}

}

public class Main {

public static void main(String[] args) {

Figura f1 = new Circulo(new OpenGL());

Figura f2 = new Rectangulo(new DirectX());

f1.dibujar();

f2.dibujar();

}

}

## 5. Conclusión

Los patrones **Adapter** y **Bridge** permiten flexibilidad y reutilización en sistemas heterogéneos. Adapter resuelve incompatibilidades entre interfaces existentes, mientras que Bridge ayuda a manejar jerarquías extensas sin crear una explosión de clases.

Implementarlos adecuadamente mejora la mantenibilidad del código y permite que diferentes módulos evolucionen de forma independiente.

# Taller del Patrón Adapter y Bridge en Sectores Productivos

## 1. Patrón Adapter

### Ejercicio 1: Sector Facturación - Integración con sistema contable antiguo

#### Historia de Usuario:

Como analista financiero, necesito que el nuevo sistema de facturación pueda generar reportes compatibles con el sistema contable antiguo de la empresa, sin cambiar la lógica principal del sistema nuevo.

#### Código Java:

interface ReporteNuevo {

void generar(String datos);

}

class SistemaContableAntiguo {

public void exportar(String contenido) {

System.out.println("Exportando a sistema contable antiguo: " + contenido);

}

}

class AdaptadorReporte implements ReporteNuevo {

private SistemaContableAntiguo sistema;

public AdaptadorReporte(SistemaContableAntiguo sistema) {

this.sistema = sistema;

}

public void generar(String datos) {

sistema.exportar(datos);

}

}

public class Main {

public static void main(String[] args) {

ReporteNuevo reporte = new AdaptadorReporte(new SistemaContableAntiguo());

reporte.generar("Factura #123 - Total: $50000");

}

}

### Ejercicio 2: Sector Transporte - Adaptador de GPS entre camiones y plataforma web

#### Historia de Usuario:

Como administrador de flotas, necesito integrar el sistema GPS de los camiones, que usa un protocolo diferente, con la plataforma web de monitoreo de rutas en tiempo real.

#### Código Java:

interface LocalizadorWeb {

void mostrarUbicacion(String coordenadas);

}

class GPSCamion {

public String obtenerUbicacion() {

return "Lat: 4.60971, Lon: -74.08175";

}

}

class AdaptadorGPS implements LocalizadorWeb {

private GPSCamion gps;

public AdaptadorGPS(GPSCamion gps) {

this.gps = gps;

}

public void mostrarUbicacion(String coordenadas) {

String ubicacion = gps.obtenerUbicacion();

System.out.println("Ubicación del camión: " + ubicacion);

}

}

public class Main {

public static void main(String[] args) {

LocalizadorWeb localizador = new AdaptadorGPS(new GPSCamion());

localizador.mostrarUbicacion("");

}

}

## 2. Patrón Bridge

### Ejercicio 3: Sector Facturación - Emisión de facturas en distintos formatos

#### Historia de Usuario:

Como desarrollador de un sistema de facturación electrónica, necesito que el sistema pueda emitir facturas tanto en formato PDF como en XML, dependiendo del cliente.

#### Código Java:

interface FormatoFactura {

void generarFactura(String datos);

}

class FormatoPDF implements FormatoFactura {

public void generarFactura(String datos) {

System.out.println("Generando factura en PDF: " + datos);

}

}

class FormatoXML implements FormatoFactura {

public void generarFactura(String datos) {

System.out.println("Generando factura en XML: " + datos);

}

}

abstract class Factura {

protected FormatoFactura formato;

public Factura(FormatoFactura formato) {

this.formato = formato;

}

public abstract void emitir(String datos);

}

class FacturaElectronica extends Factura {

public FacturaElectronica(FormatoFactura formato) {

super(formato);

}

public void emitir(String datos) {

formato.generarFactura(datos);

}

}

public class Main {

public static void main(String[] args) {

Factura factura1 = new FacturaElectronica(new FormatoPDF());

factura1.emitir("Factura #789 - Total: $1.200.000");

}

}

### Ejercicio 4: Sector Transporte - Reportes de estado de vehículos

#### Historia de Usuario:

Como supervisor de mantenimiento, quiero que el sistema de monitoreo pueda generar reportes del estado de los vehículos con distintas salidas: en pantalla, archivo de texto o correo electrónico, sin modificar la lógica de captura de datos.

#### Código Java:

interface ModoReporte {

void imprimir(String contenido);

}

class Consola implements ModoReporte {

public void imprimir(String contenido) {

System.out.println("Reporte en consola: " + contenido);

}

}

class ArchivoTexto implements ModoReporte {

public void imprimir(String contenido) {

System.out.println("Guardando en archivo: " + contenido);

}

}

abstract class ReporteVehiculo {

protected ModoReporte salida;

public ReporteVehiculo(ModoReporte salida) {

this.salida = salida;

}

public abstract void generar(String estado);

}

class ReporteEstado extends ReporteVehiculo {

public ReporteEstado(ModoReporte salida) {

super(salida);

}

public void generar(String estado) {

salida.imprimir("Estado del vehículo: " + estado);

}

}

public class Main {

public static void main(String[] args) {

ReporteVehiculo reporte = new ReporteEstado(new Consola());

reporte.generate("Motor funcionando, llantas en buen estado");

}

}

Nota:

Estos ejercicios aplican los patrones Adapter y Bridge a situaciones reales en sectores clave del país, facilitando integración, flexibilidad y extensibilidad de los sistemas empresariales.