Tecnológico Nacional De México

Instituto Tecnológico de Oaxaca

Ingeniería En Sistemas Computacionales

Diseño e Implementación de Software con Patrones.

7 am – 8 am

Unidad 2

“Patrón de Diseño Bridge”

Presenta:

Copy Max

Nombres	Numero de Control
Bautista Fabian Max	**C19160532**
Celis Delgado Jorge Eduardo	**21160599**
Flores Guzmán Alan Ismael	**20161193**
García Osorio Bolívar	**20161819**
Pérez Barrios Diego	**21160750**
Perez Martínez Edith Esmeralda	**21160752**
Sixto Morales Ángel	**21160797**

Periodo Escolar:

Febrero – Julio

Grupo:

7SB

Maestro:

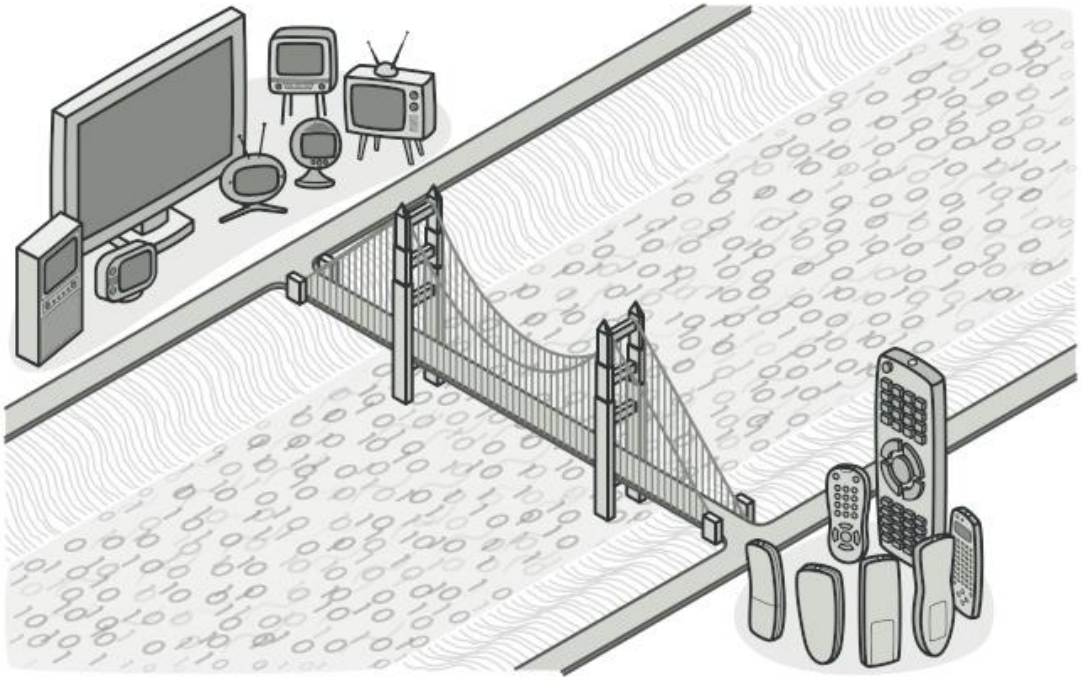
Espinoza Pérez Jacob

Oaxaca de Juárez, Oaxaca **Marzo, 2025**

INDICE

Patrón de Diseño Bridge

Bridge es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



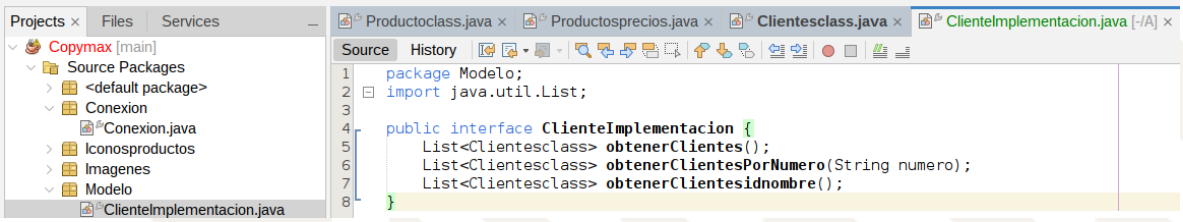
Para aplicar el patrón de Bridge vamos a trabajar sobre la clase de Clientesclass.java donde vamos a modificar la obtención de los clientes.

El patrón de bridge vamos a realizarlo en dos partes, primero la Implementación y la abstracción.

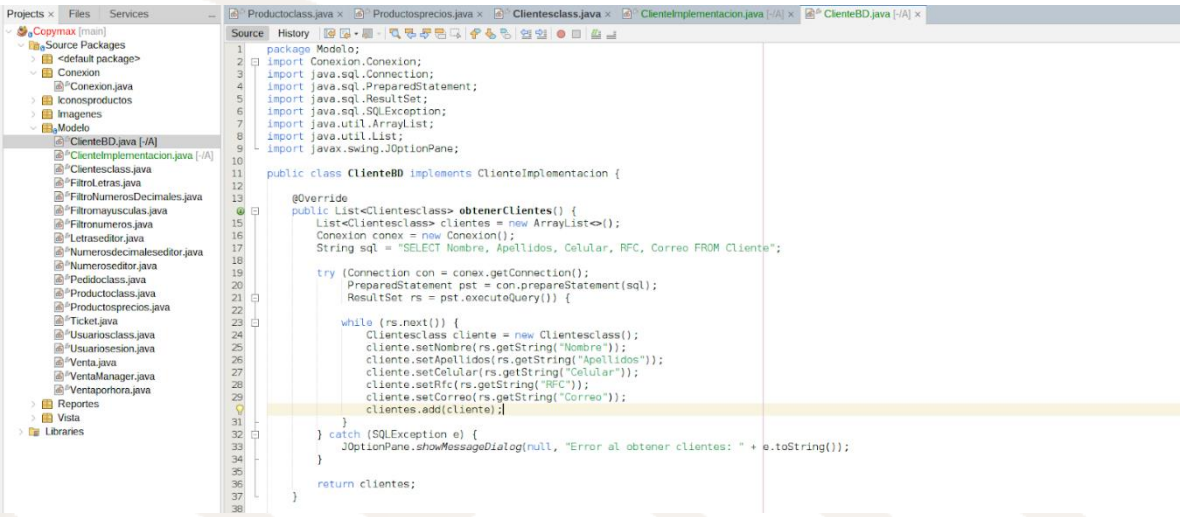
Implementación:

Vamos a realizar una interfaz llamada ClienteImplementacion que quedaría de la siguiente forma definimos que debe hacer, pero aún no, el cómo debe hacerlo:

```
package Modelo;

import java.util.List;


public interface ClienteImplementacion {
    List<Clientesclass> obtenerClientes();
    List<Clientesclass> obtenerClientesPorNumero(String numero);
    List<Clientesclass> obtenerClientesidnombre();
}
```



Crearemos nuestra implementación de la clase que se ocupara de las consultas a nuestra BDD de forma concreta, únicamente hará esa acción dejaremos la clase de Clientesclass.java sin consultas únicamente tendrá los parámetros y sus getters/setters.

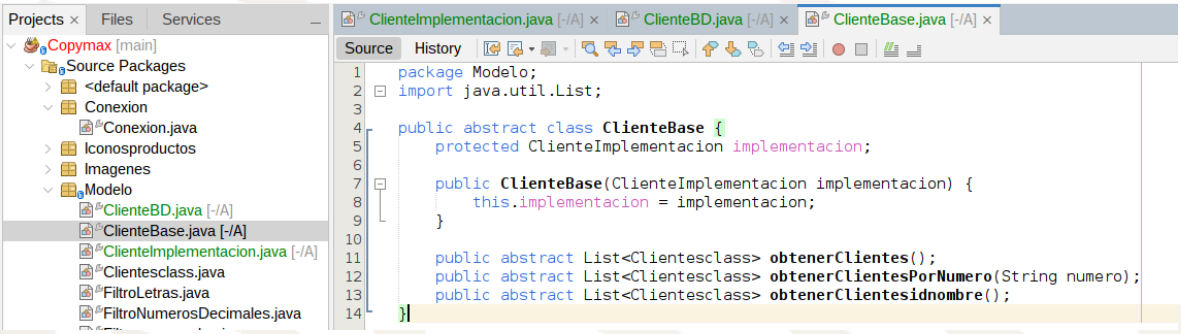


Ya que tengamos la implementación del patrón, ahora vamos a realizar la abstracción del patrón.

Primero realizaremos nuestra clase abstracta, sería nuestra clase base que define la funcionalidad genral, esta clase podría usar cualquiera de las implementaciones que hicimos anteriormente.

```
package Modelo;

import java.util.List;


public abstract class ClienteBase {

    protected ClienteImplementacion implementacion;


    public ClienteBase(ClienteImplementacion implementacion) {

        this.implementacion = implementacion;

    }


    public abstract List<Clientesclass> obtenerClientes();
```

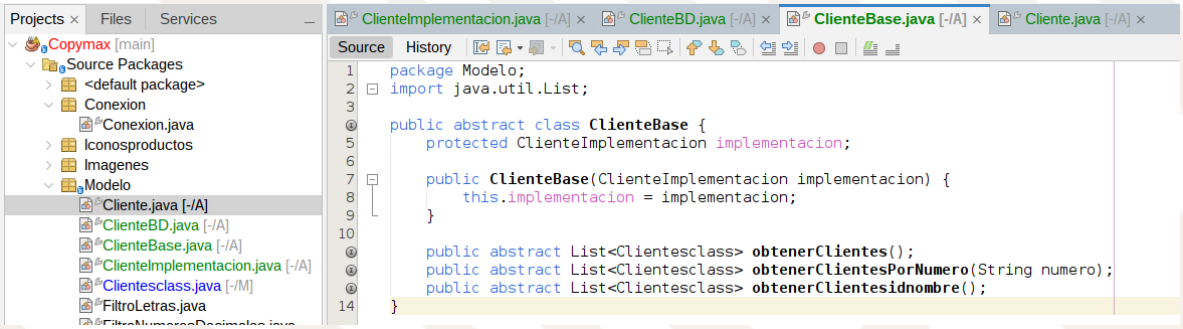
```
    public abstract List<Clientesclass> obtenerClientesPorNumero(String numero);
    public abstract List<Clientesclass> obtenerClientesidnombre();
}
```



Ahora debemos crear nuestra clase de Cliente que usara la implementación sin preocuparse por los detalles que heredan de nuestra clase abstracta.

```
package Modelo;

import java.util.List;


public class Cliente extends ClienteBase {
    public Cliente(ClienteImplementacion implementacion) {
        super(implementacion);
    }

    @Override
    public List<Clientesclass> obtenerClientes() {
        return implementacion.obtenerClientes();
```

```
    }

    @Override
    public List<Clientesclass> obtenerClientesPorNumero(String numero) {
        return implementacion.obtenerClientesPorNumero(numero);
    }

    @Override
    public List<Clientesclass> obtenerClientesidnombre() {
        return implementacion.obtenerClientesidnombre();
    }
}
```



Con eso quedaría nuestro patrón, debemos cambiar los métodos que se usaban en Clientesclass.java por los nuevos de nuestro ClienteBase que ahora ocuparemos.

Porque creamos dos clases casi del mismo tipo, ejemplo nuestra clase de ClienteBase es de forma abstracta, así que define las operaciones generales, se encarga de **definir la funcionalidad.**

Mientras que la clase de Cliente ees una implementación de ClienteBase, se encarga de **implementar la funcionalidad** con una implementación específica.

Por ejemplo, si en un futuro se debe de agregar otra clase de clientes, ya no debemos modificar nuestras clases de ClienteBase con que creemos otra clase y heredemos de nuestra clase ClienteBase.

Ahora cambiamos los métodos de la clase Clientesclass.java por los nuevos:

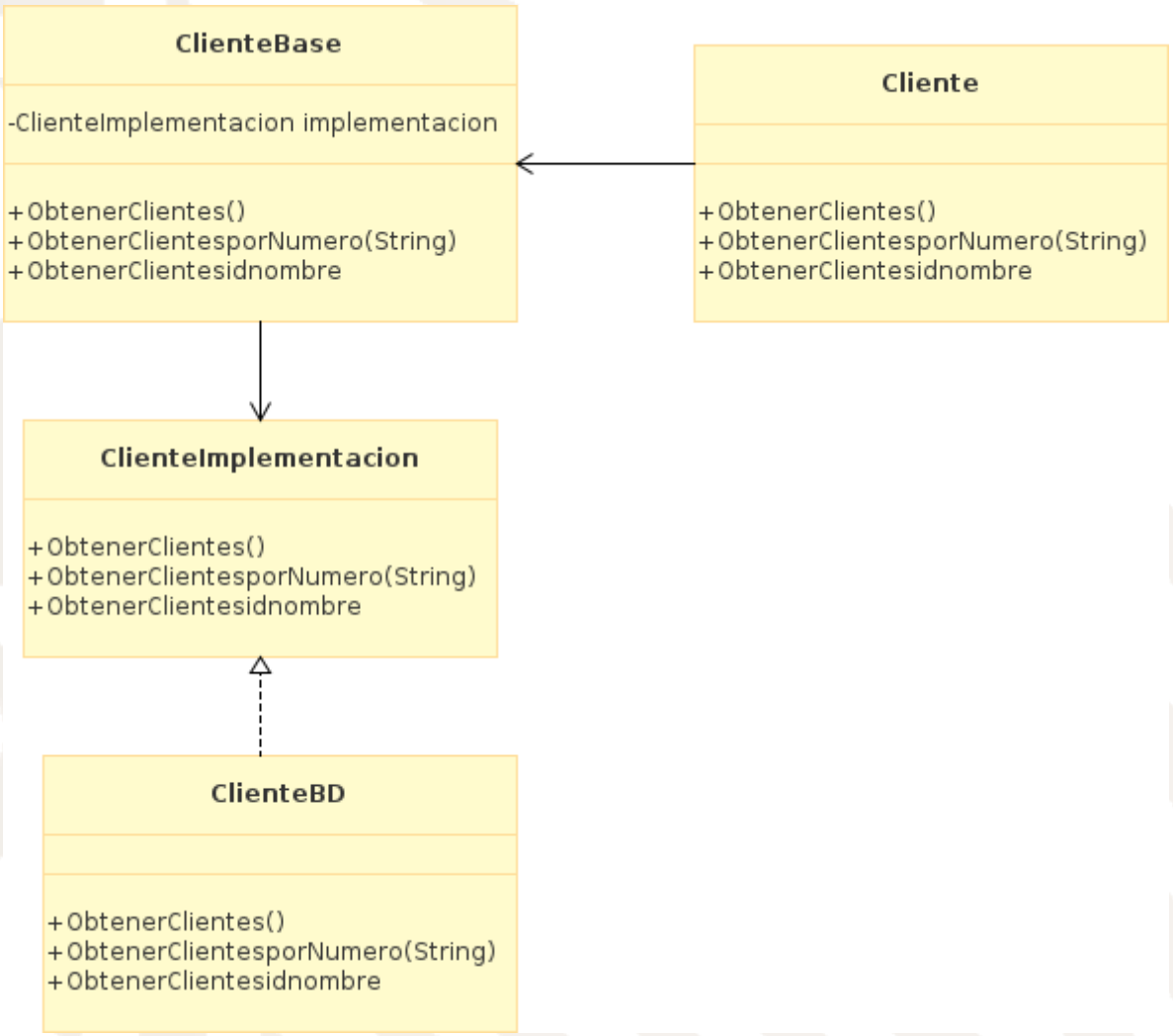
Antes:

```
private void llenarTabla() {
    Clientesclass clientee = new Clientesclass();
    List<Clientesclass> clientes = clientee.obtenerClientes();

    for (Clientesclass cliente : clientes) {
        Object[] fila = new Object[5];
        fila[0] = cliente.getNombre();
        fila[1] = cliente.getApellidos();
        fila[2] = cliente.getCelular();
        fila[3] = cliente.getRfc();
        fila[4] = cliente.getCorreo();
        modelo.addRow(fila);
    }
}
```

Despues:

```
private void llenarTabla() {
    ClienteImplementacion clienteBD = new ClienteBD();
    Cliente cliente = new Cliente(clienteBD);
    List<Clientesclass> clientes = cliente.obtenerClientes();

    for (Clientesclass c : clientes) {
        Object[] fila = new Object[5];
        fila[0] = c.getNombre();
        fila[1] = c.getApellidos();
        fila[2] = c.getCelular();
        fila[3] = c.getRfc();
        fila[4] = c.getCorreo();
        modelo.addRow(fila);
    }
}
```

Estructura UML



Ventajas y Desventajas

Ventajas	Desventajas
Puedes crear clases y aplicaciones independientes de plataforma.	Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.
El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.	
Principio de abierto/cerrado. Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.	
Principio de responsabilidad única. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.	

Conclusión

El patrón Bridge separa la abstracción de su implementación, permitiendo que ambas evolucionen de manera independiente. Este enfoque reduce el acoplamiento, mejora la extensibilidad y facilita el mantenimiento del software. Es útil cuando se espera que una jerarquía de clases deba soportar múltiples dimensiones de variabilidad.