



UNIVERSIDAD TECNOLÓGICA DEL PERÚ
Facultad de Ingeniería

Programación Orientada a Objetos

Sesión 14: Introducción a Patrones GoF

Recordando:

¿Que vimos la clase pasada?



Logro de aprendizaje



Al finalizar la sesión, el estudiante soluciona problemas aplicando Patrones GoF usando Java en la resolución de ejercicios.

Utilidad

Los Patrones GoF son herramientas poderosas que no solo facilitan la implementación de diseños robustos y eficientes, sino que también mejoran la comunicación y la colaboración entre desarrolladores, promoviendo la creación de software de alta calidad y mantenible.

Saberes Previos

- Conceptos básicos de programación orientada a objetos.
- Familiaridad con el lenguaje de programación (preferiblemente Java).
- Comprender la importancia de la reutilización de código y la mantenibilidad del software.

Agenda

- Tipos de Patrones GoF



Tipos de Patrones GoF-Patrones Creacionales

**Singleton
(Singleton
Pattern)**

**Factory Method
(Método de
Fábrica)**

Builder

Tipos de Patrones GoF-Patrones Creacionales

- **Singleton (Singleton Pattern):** Este patrón garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. Es útil cuando se necesita compartir una única instancia de una clase en todo el sistema, como un registro central o una conexión a una base de datos.
- **Factory Method (Método de Fábrica):** El patrón Factory Method delega la creación de objetos a sus subclases, permitiendo que una clase principal sea independiente de las clases concretas que crea. Esto es valioso cuando se necesita crear objetos sin especificar la clase exacta de objeto que se creará.
- **Builder:** El patrón permite construir objetos complejos paso a paso. Se utiliza para construir un objeto compuesto asegurando que el proceso de construcción sea independiente de las partes que componen el objeto y de su representación.

Tipos de Patrones GoF-Patrones Estructurales



**Adapter
(Adaptador)**

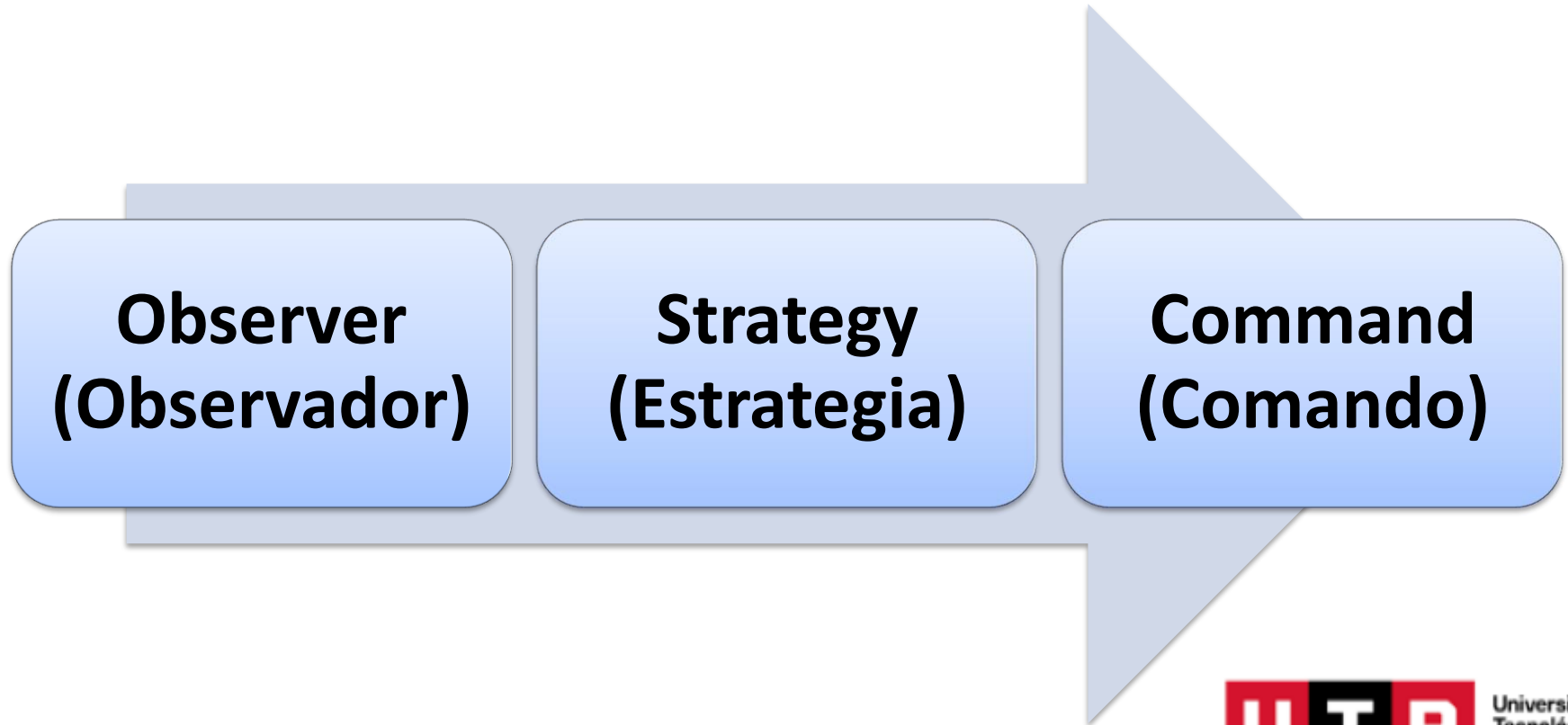
**Composite
(Composición)**

Bridge

Tipos de Patrones GoF-Patrones Estructurales

- **Adapter (Adaptador):** Este patrón permite que objetos con interfaces incompatibles trabajen juntos. Se utiliza cuando se necesita conectar dos sistemas que no pueden interactuar directamente debido a diferencias en sus interfaces.
- **Composite (Composición):** El patrón Composite se utiliza para componer objetos en estructuras de árbol para representar jerarquías de objetos. Es útil cuando se necesita tratar tanto objetos individuales como composiciones de objetos de manera uniforme.
- **Bridge:** El patrón desacopla una abstracción de su implementación, de modo que ambas pueden variar independientemente. Es especialmente útil cuando se necesita extender clases en varias dimensiones.

Tipos de Patrones GoF-Patrones Comportamiento

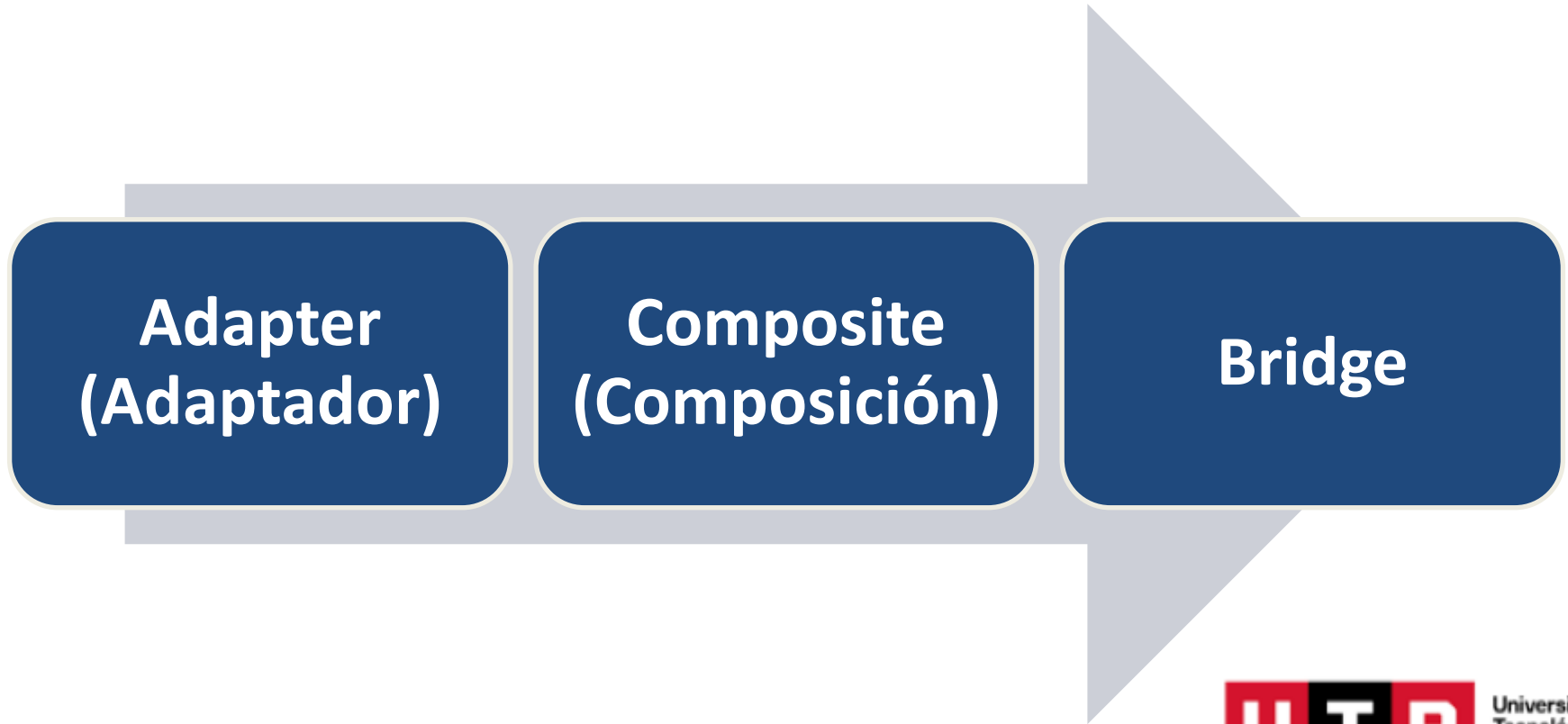


Tipos de Patrones GoF-Patrones Comportamiento

- **Observer (Observador):** Este patrón establece una relación uno a muchos entre objetos, de modo que cuando un objeto cambia su estado, todos los objetos que dependen de él sean notificados y actualizados automáticamente.
- **Strategy (Estrategia):** El patrón Strategy define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Esto permite que el cliente elija el algoritmo que se utilizará en tiempo de ejecución.
- **Command (Comando):** El patrón Command encapsula una solicitud como un objeto, lo que permite parametrizar clientes con operaciones, encolar solicitudes y respaldar operaciones deshacer. Se utiliza para desacoplar a quien emite una solicitud de quien la procesa.

Detalle de cada uno de los Patrones GoF

Patrones GoF-Patrones Estructurales



Patrón Adapter:

El patrón Adapter es un patrón estructural que permite que objetos con interfaces incompatibles trabajen juntos. Actúa como un puente entre dos interfaces incompatibles.

Propósito: Convertir la interfaz de una clase en otra interfaz que el cliente espera. El Adapter permite que clases que de otra manera serían incompatibles trabajen juntas.

Uso común: Se utiliza cuando se quiere que dos clases diferentes trabajen juntas, pero sus interfaces son incompatibles.

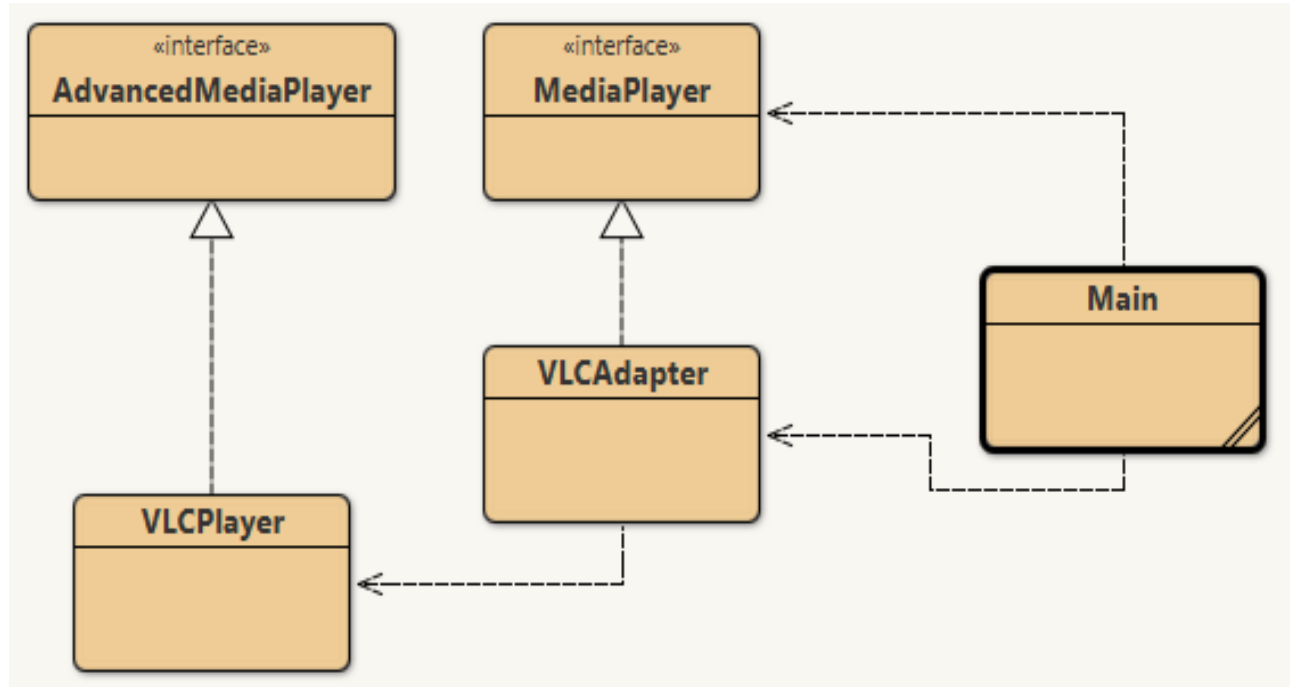
Patrón Adapter:



https://www.youtube.com/watch?v=_mU6YAAACcl

Patrón Adapter:

Supongamos que tenemos una interfaz MediaPlayer que puede reproducir archivos de audio y una interfaz avanzada AdvancedMediaPlayer que puede reproducir archivos de audio y video.



Patrón Adapter:

Implementación básica en Java:

Supongamos que tenemos una interfaz MediaPlayer que puede reproducir archivos de audio y una interfaz avanzada AdvancedMediaPlayer que puede reproducir archivos de audio y video.

```
// Interfaz MediaPlayer
public interface MediaPlayer {
    void playAudio(String fileName);
}
```

```
// Interfaz AdvancedMediaPlayer
public interface AdvancedMediaPlayer {
    void playAudio(String fileName);
    void playVideo(String fileName);
}
```

```
// Clase VLC que implementa AdvancedMediaPlayer
public class VLCPlayer implements AdvancedMediaPlayer {
    @Override
    public void playAudio(String fileName) {
        System.out.println("Playing audio with VLC: " + fileName);
    }
    @Override
    public void playVideo(String fileName) {
        System.out.println("Playing video with VLC: " + fileName);
    }
}
```

Patrón Adapter:

Ahora, queremos que VLCPlayer sea compatible con la interfaz MediaPlayer. Para ello, creamos un Adapter:

```
public class VLCAdapter implements
MediaPlayer {
    private VLCPlayer vlcPlayer;
    public VLCAdapter(VLCPlayer vlcPlayer) {
        this.vlcPlayer = vlcPlayer;
    }
    @Override
    public void playAudio(String fileName) {
        vlcPlayer.playAudio(fileName);
    }
}
```

Patrón Adapter:

Cómo funciona:

La interfaz MediaPlayer tiene un método para reproducir audio, mientras que AdvancedMediaPlayer tiene métodos para reproducir audio y video.

- VLCPlayer es una clase que implementa AdvancedMediaPlayer.
- VLCAdapter es la clase Adapter que hace que VLCPlayer sea compatible con la interfaz MediaPlayer.
- En el método main, se utiliza el Adapter para reproducir un archivo de audio a través de VLCPlayer usando la interfaz MediaPlayer.

```
public class Main {  
    public static void main(String[] args) {  
        VLCPlayer vlc = new VLCPlayer();  
        MediaPlayer player = new  
VLCAdapter(vlc);  
        player.playAudio("song.mp3");  
        // Salida: Playing audio with VLC:  
song.mp3  
    }  
}
```

Patrón Adapter:

El patrón Adapter es útil para integrar sistemas o bibliotecas que tienen interfaces diferentes, permitiendo que trabajen juntas sin modificar el código original.

Patrón Bridge:

El patrón Bridge es un patrón estructural que desacopla una abstracción de su implementación, de modo que ambas pueden variar independientemente. Es especialmente útil cuando se necesita extender clases en varias dimensiones.

Propósito: Separar una abstracción de su implementación para que ambas puedan ser modificadas independientemente sin afectarse mutuamente.

Uso común: Se utiliza cuando se quiere evitar un enlace permanente entre una abstracción y su implementación, especialmente si ambas deben extenderse en diferentes direcciones.

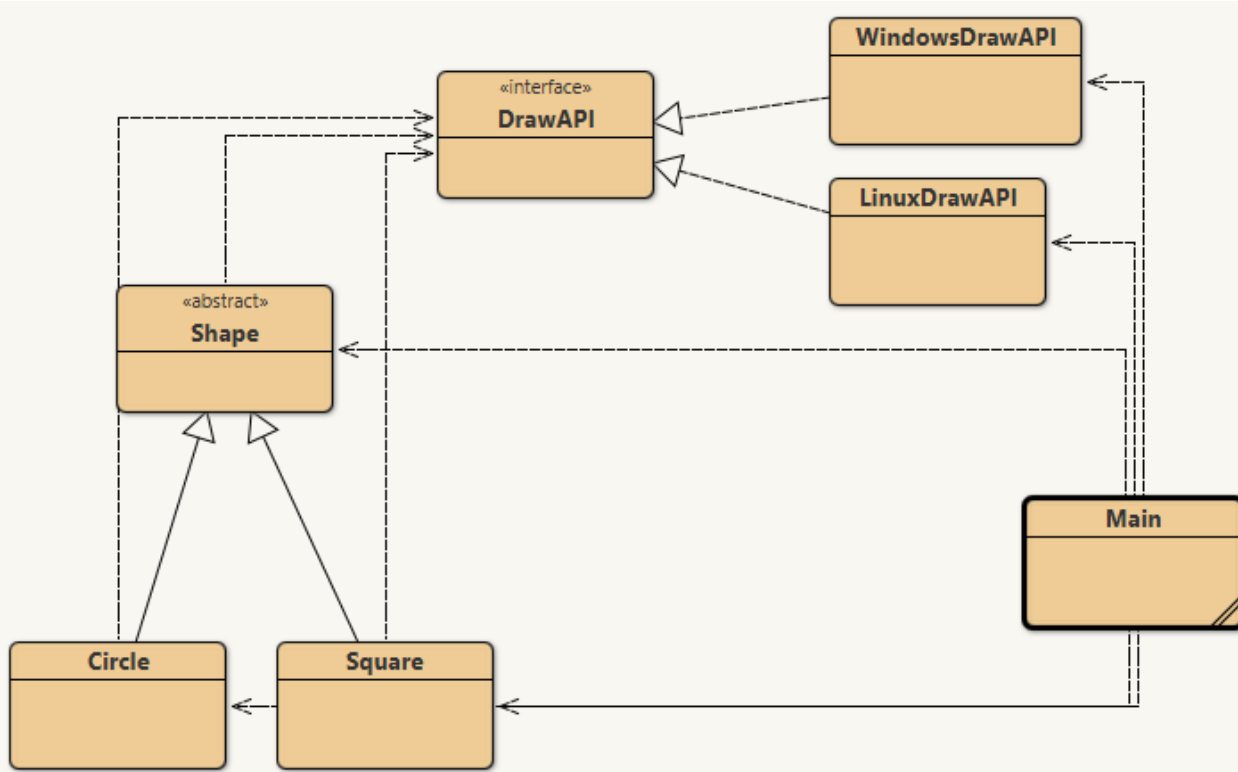
Patrón Bridge:



<https://www.youtube.com/watch?v=uY6uEvvc578>

Patrón Bridge:

Supongamos que tenemos un sistema de dibujo con formas como círculos y cuadrados, y queremos dibujar estas formas en diferentes plataformas (por ejemplo, Windows y Linux).



Patrón Bridge:

Implementación básica en Java:

Supongamos que tenemos un sistema de dibujo con formas como círculos y cuadrados, y queremos dibujar estas formas en diferentes plataformas (por ejemplo, Windows y Linux).

Primero, definimos la "implementación" (la plataforma en la que se dibujará la forma):

```
// Implementor
public interface DrawAPI {
    void drawShape(String shapeType);
}

// ConcreteImplementorA
public class WindowsDrawAPI implements DrawAPI {
    @Override
    public void drawShape(String shapeType) {
        System.out.println("Drawing " + shapeType + " on Windows");
    }
}

// ConcreteImplementorB
public class LinuxDrawAPI implements DrawAPI {
    @Override
    public void drawShape(String shapeType) {
        System.out.println("Drawing " + shapeType + " on Linux");
    }
}
```

Patrón Bridge:

Luego, definimos la "abstracción" (las formas) y la "abstracción refinada" (implementaciones específicas de las formas):

```
// Abstraction
public abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI) {
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}

// RefinedAbstractionA
public class Circle extends Shape {
    public Circle(DrawAPI drawAPI) {
        super(drawAPI);
    }
}
```

```
@Override
    public void draw() {
        drawAPI.drawShape("Circle");
    }
}

// RefinedAbstractionB
public class Square extends Shape {
    public Square(DrawAPI drawAPI) {
        super(drawAPI);
    }
    @Override
    public void draw() {
        drawAPI.drawShape("Square");
    }
}
```

Patrón Bridge:

Cómo funciona:

- DrawAPI es la interfaz que define cómo se dibujará la forma (la "implementación").
- WindowsDrawAPI y LinuxDrawAPI son implementaciones concretas de DrawAPI.
- Shape es una abstracción que representa una forma y tiene una referencia a DrawAPI.
- Circle y Square son abstracciones refinadas que extienden Shape y definen cómo se dibujarán las formas específicas.
- En el método main, se crean formas específicas para plataformas específicas utilizando el patrón Bridge.

```
public class Main {  
    public static void main(String[] args) {  
        Shape circleOnWindows = new  
        Circle(new WindowsDrawAPI());  
        circleOnWindows.draw(); //  
        Salida: Drawing Circle on Windows  
        Shape squareOnLinux = new  
        Square(new LinuxDrawAPI());  
        squareOnLinux.draw(); // Salida:  
        Drawing Square on Linux  
    }  
}
```

Patrón Bridge:

El patrón Bridge es útil para desacoplar abstracciones e implementaciones, permitiendo que ambas jerarquías se extiendan independientemente.

Patrón Composite:

Propósito:

Componer objetos en estructuras de árbol para representar jerarquías parte-todo. El patrón Composite permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme.

Uso común:

Se utiliza cuando se quiere tratar tanto objetos individuales como grupos de objetos de la misma manera.

Patrón Composite:



<https://www.youtube.com/watch?v=4GJytYyol5g>

Patrón Composite:

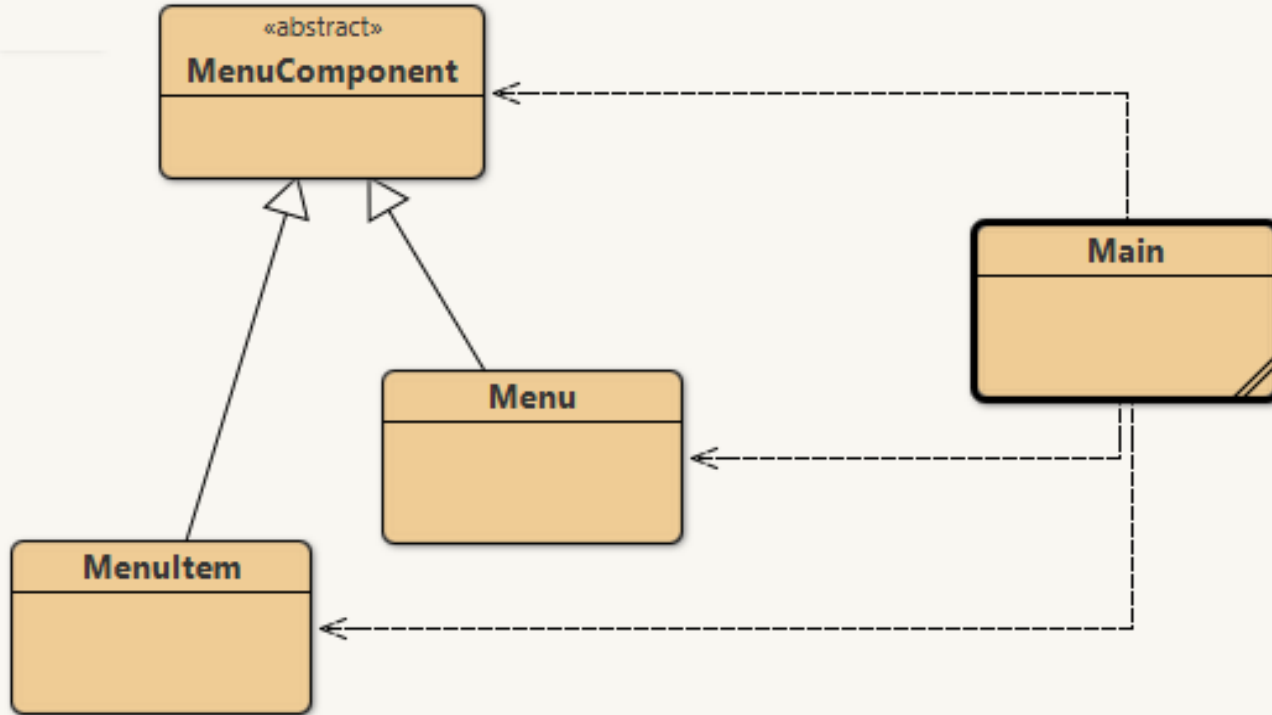
Implementación básica en Java:

Supongamos que estamos diseñando un sistema para representar un menú de elementos. Un elemento del menú puede ser un ítem individual o un submenú que contiene otros ítems.

```
public abstract class MenuComponent {  
    public void add(MenuComponent menuComponent)  
    {  
        throw new UnsupportedOperationException();  
    }  
    public void remove(MenuComponent  
menuComponent) {  
        throw new UnsupportedOperationException();  
    }  
    public String getName() {  
        throw new UnsupportedOperationException();  
    }  
    public void print() {  
        throw new UnsupportedOperationException();  
    }  
}
```

Patrón Composite:

Supongamos que estamos diseñando un sistema para representar un menú de elementos. Un elemento del menú puede ser un ítem individual o un submenú que contiene otros ítems.



Patrón Composite:

Implementación de un ítem individual (Leaf):

```
public class MenuItem extends MenuComponent {  
    private String name;  
    public MenuItem(String name) {  
        this.name = name;  
    }  
    @Override  
    public String getName() {  
        return name;  
    }  
    @Override  
    public void print() {  
        System.out.println("Item: " + getName());  
    }  
}
```

Patrón Composite: Implementación del submenú (Composite)

```
import java.util.ArrayList;
import java.util.List;
public class Menu extends MenuComponent {
    private List<MenuComponent> menuComponents = new ArrayList<>();
    private String name;
    public Menu(String name) {
        this.name = name;
    }
    @Override
    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }
}
```

Patrón Composite: Implementación del submenú

```
@Override
public void remove(MenuComponent menuComponent) {
    menuComponents.remove(menuComponent);
}
@Override
public String getName() {
    return name;
}
@Override
public void print() {
    System.out.println("\nSubmenu: " + getName());
    for (MenuComponent menuComponent : menuComponents) {
        menuComponent.print();
    }
}
}
```

Patrón Composite:

```
public class Main {  
    public static void main(String[] args) {  
        MenuComponent breakfastItem = new MenuItem("Pancake");  
        MenuComponent lunchItem = new MenuItem("Burger");  
        Menu breakfastMenu = new Menu("Breakfast Menu");  
        Menu lunchMenu = new Menu("Lunch Menu");  
        breakfastMenu.add(breakfastItem);  
        lunchMenu.add(lunchItem);  
        Menu allMenus = new Menu("All Menus");  
        allMenus.add(breakfastMenu);  
        allMenus.add(lunchMenu);  
        allMenus.print();  
    }  
}
```

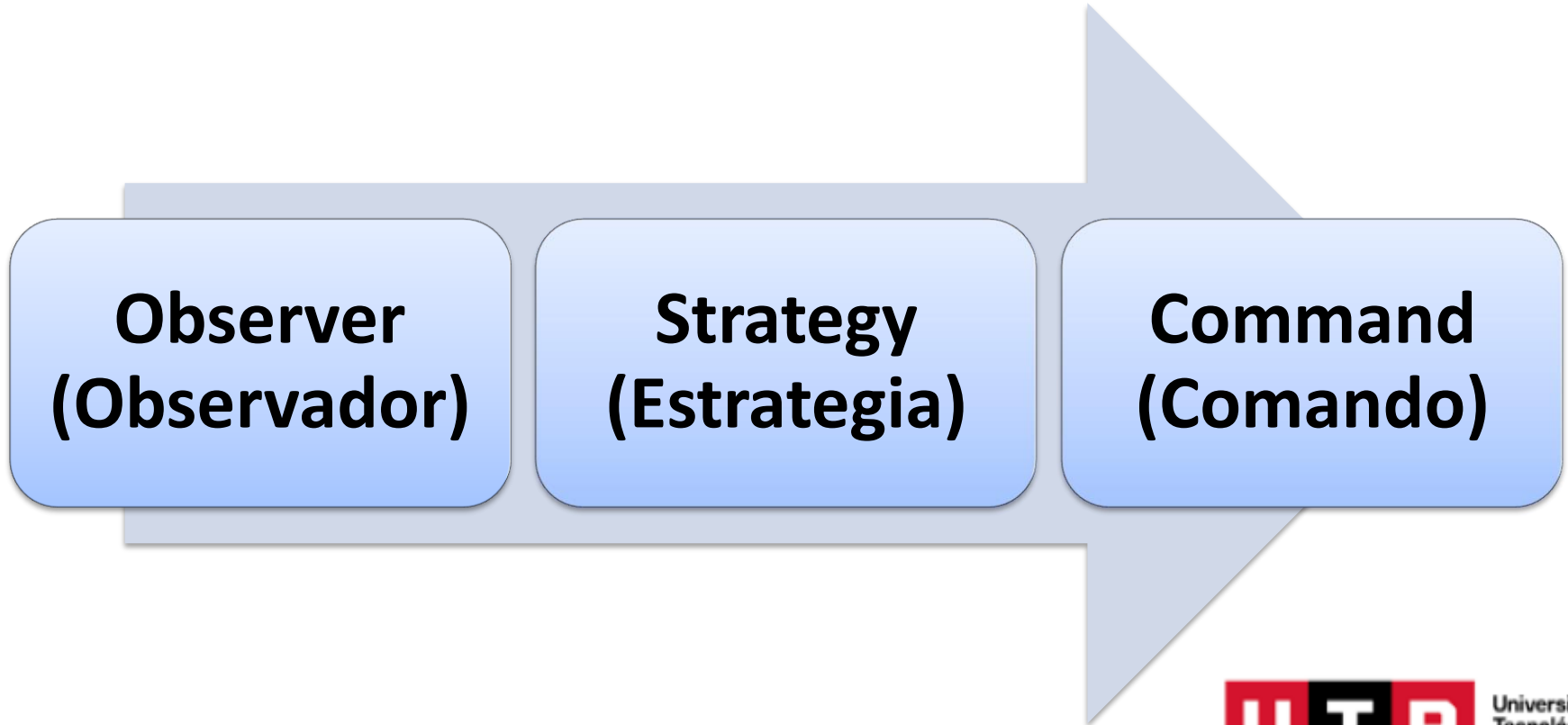
Patrón Composite:

El patrón Composite es útil para representar estructuras jerárquicas y permite tratar objetos individuales y grupos de objetos de manera uniforme.

Cómo funciona:

- MenuComponent es la interfaz base que define operaciones comunes para objetos simples y compuestos.
- MenuItem representa un ítem individual y es una hoja en la estructura del árbol.
- Menu representa un grupo de MenuComponent y puede contener tanto ítems individuales (MenuItem) como otros submenús (Menu).
- En el método main, se crean diferentes menús y submenús utilizando el patrón Composite.

Patrones GoF-Patrones Comportamiento



Patrón Observer:

El patrón Observer es un patrón de diseño de comportamiento que 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.

Propósito: Establecer una relación uno-a-muchos entre objetos para que, cuando un objeto cambie de estado, todos sus dependientes sean notificados y actualizados automáticamente.

Uso común: Se utiliza en situaciones donde un objeto (el sujeto) necesita mantener informados a otros objetos (observadores) sobre cambios en su estado.

Patrón Observer:

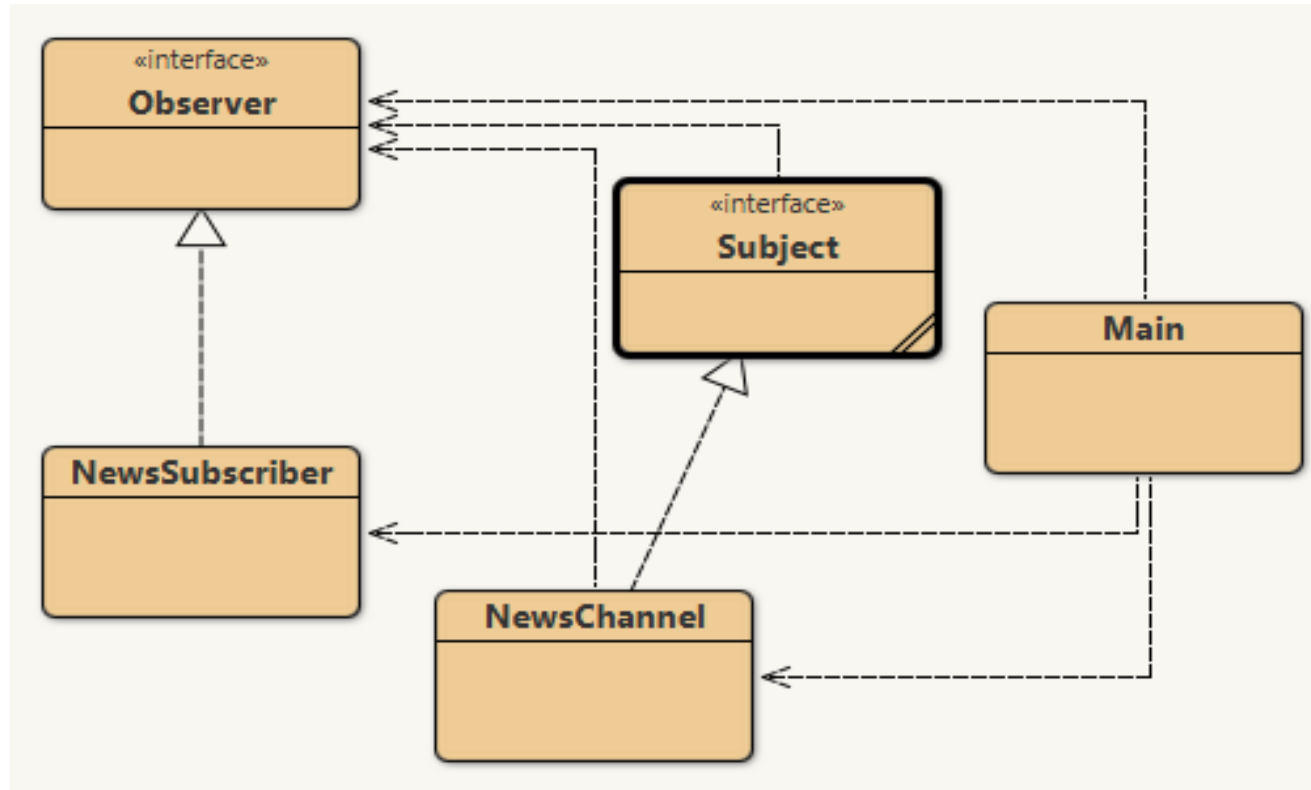
PATRON de DISEÑO OBSERVER

```
observer1.observar()  
observer2.observar()  
observable.notificar(  
observer1,observer2)
```



https://www.youtube.com/watch?v=JIN--0m_V7Q&list=PL1YXwF4Lvrn1c9yrsoG3i0aKuYmIf0FjU&index=6

Patrón Observer:



Patrón Observer: Implementación básica en Java:

Definir la interfaz Observer:

```
public interface Observer {  
    void update(String  
message);  
}
```

Definir la interfaz Subject:

```
public interface Subject {  
    void registerObserver(Observer  
observer);  
    void removeObserver(Observer  
observer);  
    void notifyObservers();  
}
```

Patrón Observer: Implementación concreta del Subject (por ejemplo, un canal de noticias)

```
import java.util.ArrayList;
import java.util.List;
public class NewsChannel implements Subject {
    private List<Observer> observers = new
    ArrayList<>();
    private String news;
    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
}
```

```
@Override
public void removeObserver(Observer observer) {
    observers.remove(observer);
}
@Override
public void notifyObservers() {
    for (Observer observer : observers) {
        observer.update(news);
    }
}
public void setNews(String news) {
    this.news = news;
    notifyObservers();
}
}
```

Patrón Observer: Implementación concreta del Observer (por ejemplo, un suscriptor de noticias):

```
public class NewsSubscriber implements Observer {  
    private String name;  
    public NewsSubscriber(String name) {  
        this.name = name;  
    }  
    @Override  
    public void update(String message) {  
        System.out.println(name + " received news: " +  
message);  
    }  
}
```

Patrón Observer:

```
public class Main {  
    public static void main(String[] args) {  
        NewsChannel channel = new NewsChannel();  
        Observer subscriber1 = new NewsSubscriber("Alice");  
        Observer subscriber2 = new NewsSubscriber("Bob");  
        channel.registerObserver(subscriber1);  
        channel.registerObserver(subscriber2);  
        channel.setNews("Breaking News: Observer Pattern is  
awesome!");  
        channel.removeObserver(subscriber1);  
        channel.setNews("Update: Patterns are essential in  
software design.");  
    }  
}
```

Patrón Observer:

El patrón Observer es útil para crear sistemas desacoplados donde múltiples objetos deben ser informados sobre cambios en otro objeto.

Cómo funciona:

- Observer y Subject son las interfaces principales que definen los métodos esenciales para el patrón.
- NewsChannel es un sujeto concreto que mantiene una lista de observadores y notifica a todos ellos cuando hay una actualización.
- NewsSubscriber es un observador concreto que recibe y procesa las actualizaciones del sujeto.
- En el método main, se crean un canal y dos suscriptores. Cuando el canal actualiza sus noticias, notifica a todos los suscriptores registrados.

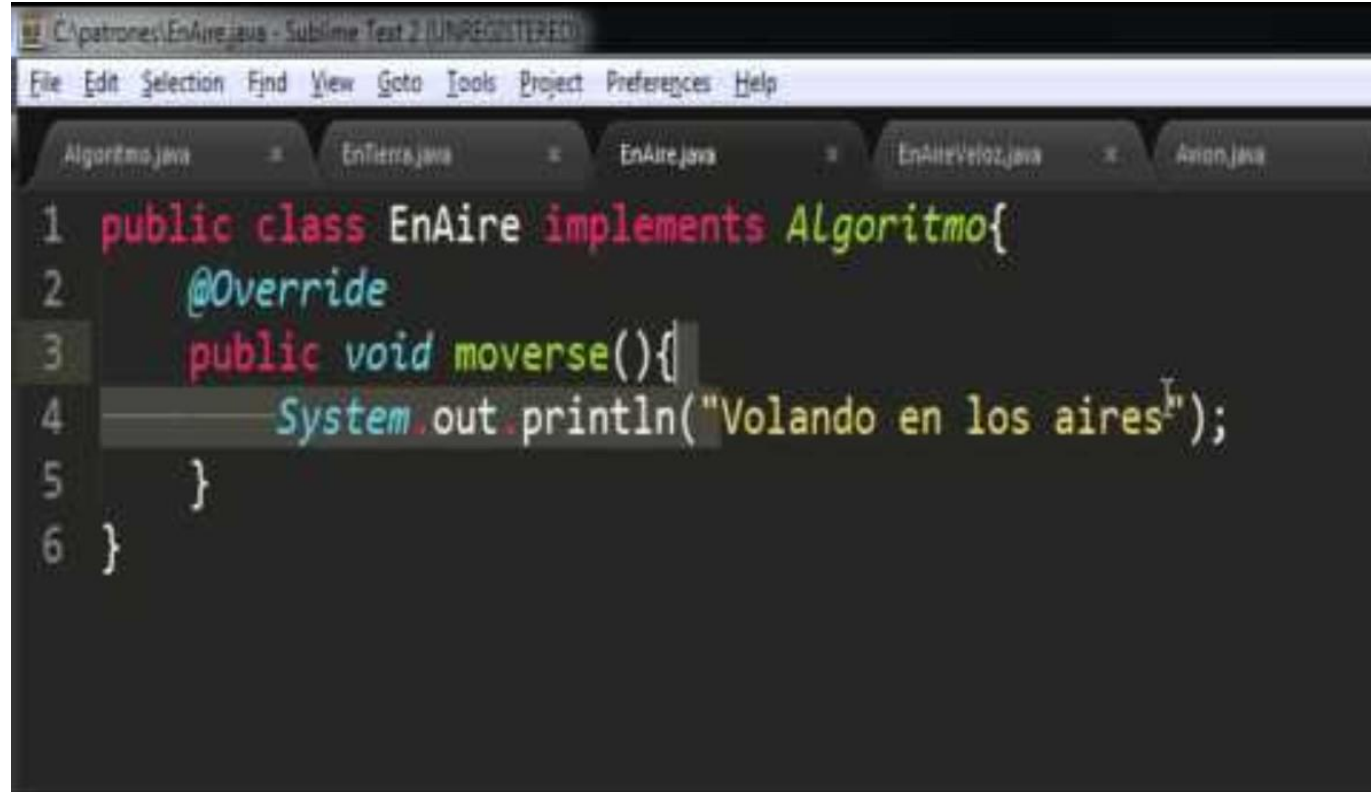
Patrón Strategy:

El patrón Strategy es un patrón de diseño de comportamiento que permite seleccionar un algoritmo de implementación en tiempo de ejecución.

Esencialmente, define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables.

- **Propósito:** Definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. El patrón Strategy permite que el algoritmo varíe independientemente de los clientes que lo utilizan.
- **Uso común:** Se utiliza cuando hay múltiples formas de hacer algo (algoritmos) y se quiere poder cambiar entre ellos dinámicamente.

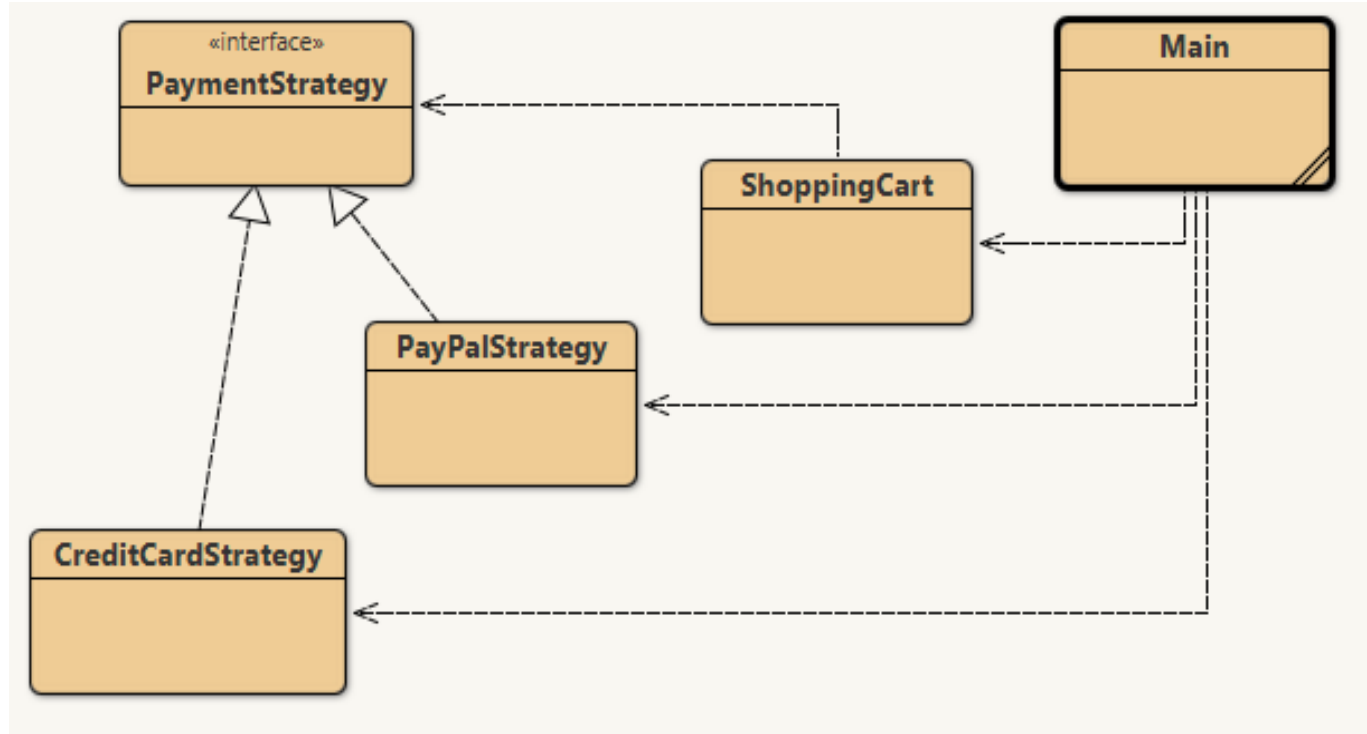
Patrón Strategy:



The screenshot shows a code editor window titled "C:\patrones\EnAire.java - Sublime Text 2 (UNREGISTERED)". The editor has a menu bar with "File", "Edit", "Selection", "Find", "View", "Goto", "Tools", "Project", "Preferences", and "Help". Below the menu bar, there are several tabs: "Algoritmo.java", "EnTierra.java", "EnAire.java", "EnAireVeloz.java", and "Avion.java". The "EnAire.java" tab is active, showing the following code:

```
1 public class EnAire implements Algoritmo{
2     @Override
3     public void moverse(){
4         System.out.println("Volando en los aires");
5     }
6 }
```


Patrón Strategy:



Patrón Strategy:

```
//Definir la interfaz Strategy:  
public interface PaymentStrategy {  
    void pay(int amount);  
}
```

```
public class CreditCardStrategy implements  
PaymentStrategy {  
    private String cardNumber;  
    private String name;  
    public CreditCardStrategy(String name, String  
cardNumber) {  
        this.name = name;  
        this.cardNumber = cardNumber;  
    }  
    @Override  
    public void pay(int amount) {  
        System.out.println("Pagado S/. " + amount + " utilizando tarjeta de crédito.");  
    }  
}
```

Patrón Strategy:

```
public class PayPalStrategy implements PaymentStrategy {  
    private String email;  
    public PayPalStrategy(String email) {  
        this.email = email;  
    }  
    @Override  
    public void pay(int amount) {  
        System.out.println("Pagado S/. " + amount + " usando PayPal.");  
    }  
}
```

Patrón Strategy:

```
public class ShoppingCart {  
    private PaymentStrategy paymentStrategy;  
    public ShoppingCart(PaymentStrategy paymentStrategy) {  
        this.paymentStrategy = paymentStrategy;  
    }  
    public void checkout(int amount) {  
        paymentStrategy.pay(amount);  
    }  
    public void setPaymentStrategy(PaymentStrategy  
paymentStrategy) {  
        this.paymentStrategy = paymentStrategy;  
    }  
}
```

Patrón Strategy:

```
public class Main {  
    public static void main(String[] args) {  
        ShoppingCart cart = new ShoppingCart(new CreditCardStrategy("Alice", "1234-5678-9012-3456"));  
        cart.checkout(100); // Salida: Paid 100 using Credit Card.  
        cart.setPaymentStrategy(new PayPalStrategy("alice@example.com"));  
        cart.checkout(50); // Salida: Paid 50 using PayPal.  
    }  
}
```

Patrón Strategy:

Cómo funciona:

- `PaymentStrategy` es la interfaz principal que define una operación que varía según la estrategia.
- `CreditCardStrategy` y `PayPalStrategy` son implementaciones concretas de diferentes estrategias de pago.
- `ShoppingCart` es el contexto que utiliza una estrategia de pago. Puede cambiar su estrategia de pago en tiempo de ejecución mediante el método `setPaymentStrategy`.
- En el método `main`, se crea un carrito de compras y se realiza el pago utilizando diferentes estrategias.

El **patrón Strategy** es útil para desacoplar algoritmos de las clases que los utilizan, permitiendo cambiar algoritmos en tiempo de ejecución sin modificar el código.

Patrón Command:

El patrón Command es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene información sobre la solicitud. Esto permite desacoplar clases que invocan operaciones de clases que realizan la operación.

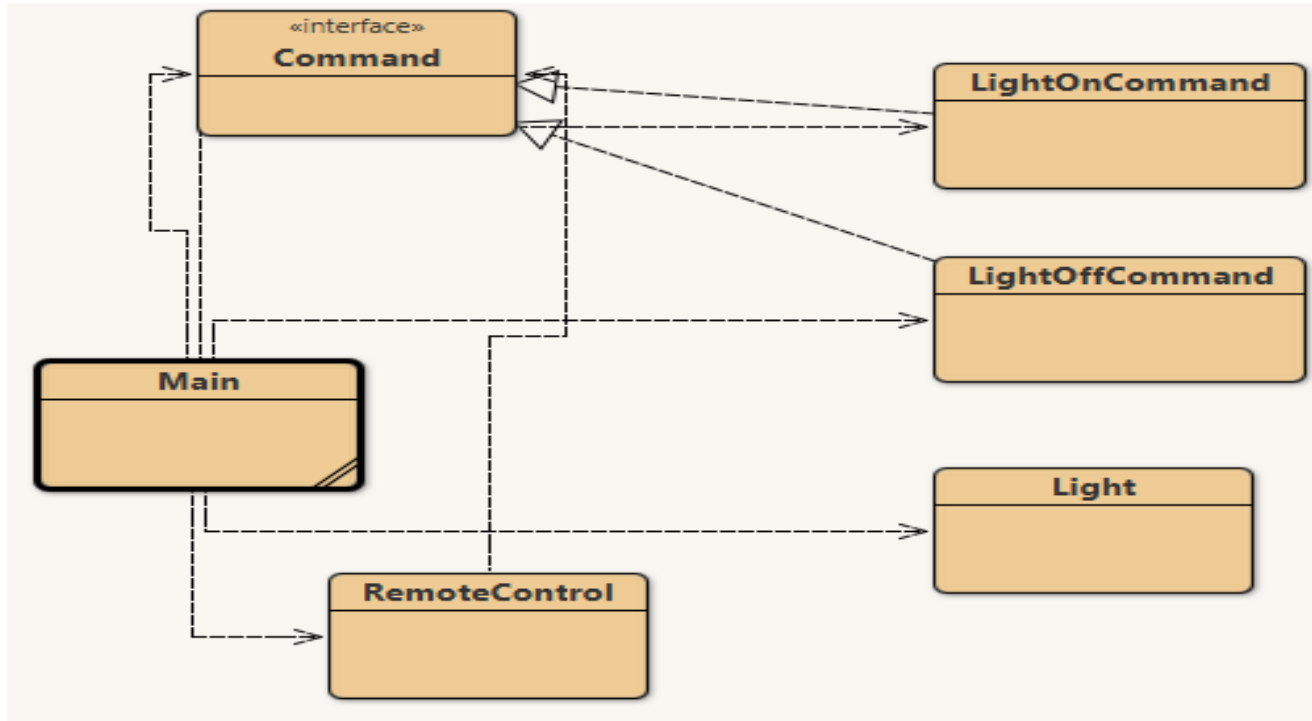
- **Propósito:** Encapsular una solicitud como un objeto, permitiendo así a los usuarios parametrizar clientes con diferentes solicitudes, encolar solicitudes y soportar operaciones que no se pueden realizar.
- **Uso común:** Se utiliza cuando se quiere desacoplar el objeto que invoca una operación del objeto que sabe cómo realizarla, o cuando se necesita soportar operaciones de deshacer/rehacer.

Patrón Command:



<https://www.youtube.com/watch?v=hDBOfyzFKEU>

Patrón Command:



Implementaciones concretas de Command (encender y apagar una luz):

```
public class LightOnCommand implements Command {  
    private Light light;  
    public LightOnCommand(Light light) {  
        this.light = light;  
    }  
    @Override  
    public void execute() {  
        light.turnOn();  
    }  
}
```

```
public class LightOffCommand implements Command {  
    private Light light;  
    public LightOffCommand(Light light) {  
        this.light = light;  
    }  
    @Override  
    public void execute() {  
        light.turnOff();  
    }  
}
```

Patrón Command:

Definir la interfaz Command:

```
public interface Command {  
    void execute();  
}
```

Patrón Command:

Clase Receiver (la luz en este caso):

```
public class Light {  
    public void turnOn() {  
        System.out.println("La luz está encendida");  
    }  
    public void turnOff() {  
        System.out.println("La luz está apagada");  
    }  
}
```

Invoker (control remoto que puede ejecutar comandos):

```
public class RemoteControl {  
    private Command command;  
    public void setCommand(Command command) {  
        this.command = command;  
    }  
    public void pressButton() {  
        command.execute();  
    }  
}
```

Patrón Command:

```
public class Main {  
    public static void main(String[] args) {  
        Light light = new Light();  
        Command lightOn = new LightOnCommand(light);  
        Command lightOff = new LightOffCommand(light);  
        RemoteControl remote = new RemoteControl();  
        remote.setCommand(lightOn);  
        remote.pressButton(); // Salida: Light is ON  
        remote.setCommand(lightOff);  
        remote.pressButton(); // Salida: Light is OFF  
    }  
}
```

Patrón Command:

Cómo funciona:

- Command es la interfaz principal que define una operación.
- LightOnCommand y LightOffCommand son implementaciones concretas de comandos que invocan operaciones en el objeto Light.
- Light es el receptor que realiza la operación real.
- RemoteControl es el invocador que solicita una operación. Puede cambiar su comando en tiempo de ejecución mediante el método setCommand.
- En el método main, se crea un control remoto y se controla una luz utilizando diferentes comandos.

El patrón Command es útil para desacoplar el objeto que invoca una operación del objeto que realiza la operación, y para soportar operaciones como deshacer/rehacer, encolar solicitudes, etc.

Practica

Implementa los ejercicios vistos.

Conclusiones:

- Cada uno de estos patrones GoF aborda problemas de diseño específicos y proporciona soluciones probadas.
- Los desarrolladores pueden seleccionar y aplicar estos patrones según las necesidades de sus proyectos, lo que contribuye a un diseño de software más flexible, mantenible y escalable.



Conclusiones:

- Al comprender y aplicar estos patrones, los desarrolladores pueden tomar decisiones informadas sobre cómo abordar los problemas de diseño en sus proyectos y mejorar la calidad de su código.
- Estos patrones han resistido la prueba del tiempo y se han convertido en herramientas esenciales en el kit de herramientas de cualquier desarrollador de software.



Cierre

¿Qué hemos aprendido hoy?

Bibliografía

- MORENO PÉREZ, J. “Programación orientada a objetos”. RA-MA Editorial.
<https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=31933>
- Vélez Serrano, José. “Diseñar y programar, todo es empezar: una introducción a la Programación Orientada a Objetos usando UML y Java”.
Dykinson. <https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=36368>