Técnicas de programación avanzada

Curso 2023-2024

Tema 3.3 Patrones Estructurales

Tema 1: Objetos y memoria.

1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

1.2 Sentencias de control. Secuencia, selección e iteración.

1.3 Abstracción. Clases, objetos, métodos y atributos.

1.4 Sobrecarga de métodos y encapsulamiento.

Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos

2.1 Herencia. Interfaces y clases abstractas. Agregación.

2.2 Polimorfismo.

2.3 Gestión de Excepciones.

2.4 Genericidad y plantillas.

2.5 Utilidades. Entrada y Salida.

2.6 Anotaciones.

Tema 3. Patrones de Diseño.

3.1 Concepto de Patrones de Diseño.

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

Tema 4. Programación de Interfaces.

4.1 Interfaces Gráficas de Usuario.

4.2 Gestión de eventos.

Tema 5. Temas Avanzados.

5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

3.3 Patrones estructurales

Introducción

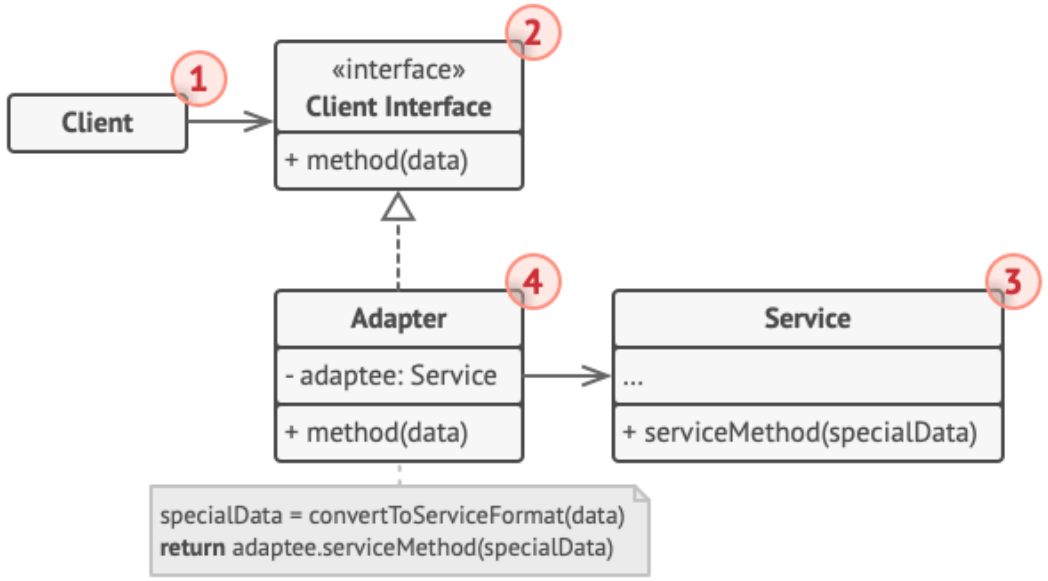
Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.

Patrón	Finalidad
Adapter	Permite la colaboración entre objetos con interfaces incompatibles.
Bridge	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.
Composite	Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.
Decorator	Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.
Facade	Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.
Flyweight	Permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.
Proxy	Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

3.3 Patrones Estructurales

ADAPTER

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.



Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar.

Partes:

(1) La clase ***Client*** contiene la lógica de negocio existente del programa.

(2) ***Client Interface*** describe un protocolo que otras clases deben seguir para poder colaborar con el código *Client*.

(3) ***Service*** es alguna clase útil (normalmente de una tercera parte o heredada). *Client* no puede utilizar directamente esta clase porque tiene una interfaz incompatible.

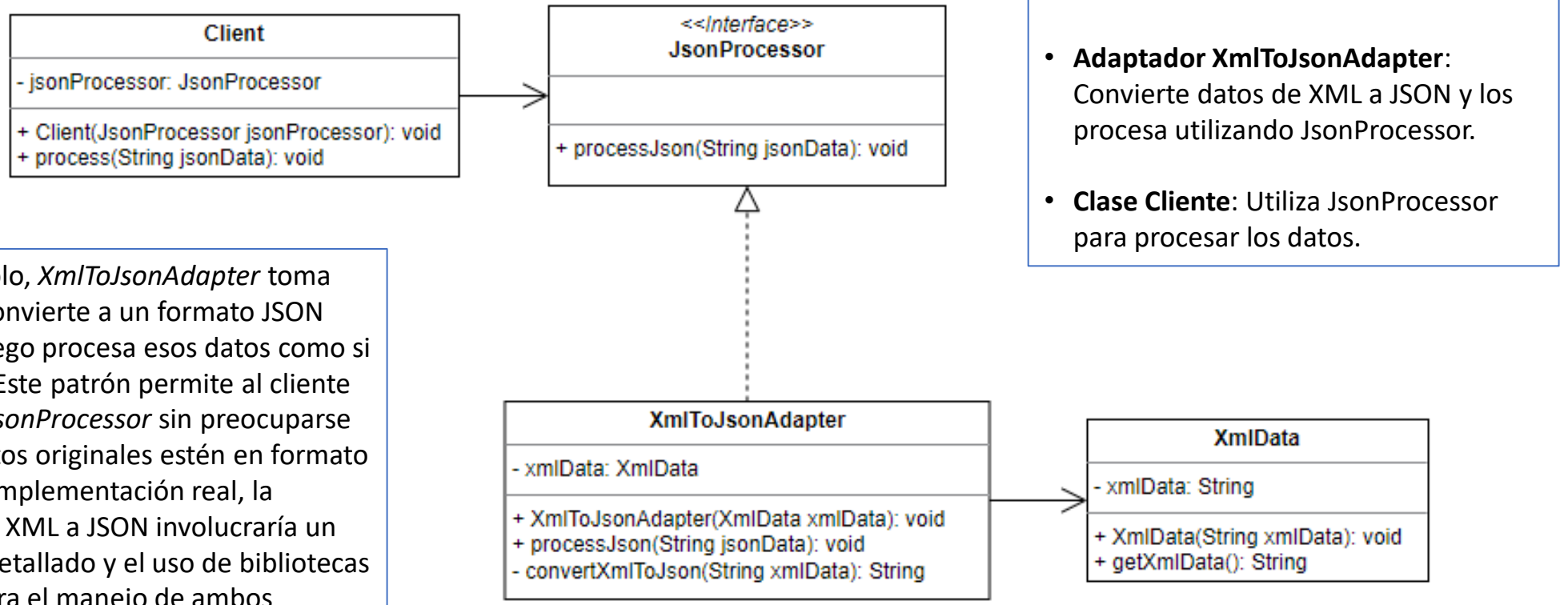
(4) La clase ***Adapter*** es capaz de trabajar tanto con la clase *Client* como con la clase *Service*: implementa la interfaz con el cliente (*Client Interface*), mientras envuelve el objeto de la clase *Service*. La clase *Adapter* recibe llamadas de *Client* a través de *Client Interface* y las traduce en llamadas al objeto envuelto de la clase *Service*, pero en un formato que pueda comprender.

https://refactoring.guru/es/design-patterns/adapter

3.3 Patrones Estructurales

ADAPTER

Ejemplo de conversión de XML a JSON



- **Interfaz JsonProcessor**: Define cómo procesar datos en formato JSON.
- **Clase XmlData**: Representa los datos en formato XML.
- **Adaptador XmlToJsonAdapter**: Convierte datos de XML a JSON y los procesa utilizando JsonProcessor.
- **Clase Cliente**: Utiliza JsonProcessor para procesar los datos.

En este ejemplo, *XmlToJsonAdapter* toma *XmlData*, la convierte a un formato JSON simulado y luego procesa esos datos como si fueran JSON. Este patrón permite al cliente trabajar con *JsonProcessor* sin preocuparse de que los datos originales estén en formato XML. En una implementación real, la conversión de XML a JSON involucraría un análisis más detallado y el uso de bibliotecas adecuadas para el manejo de ambos formatos.

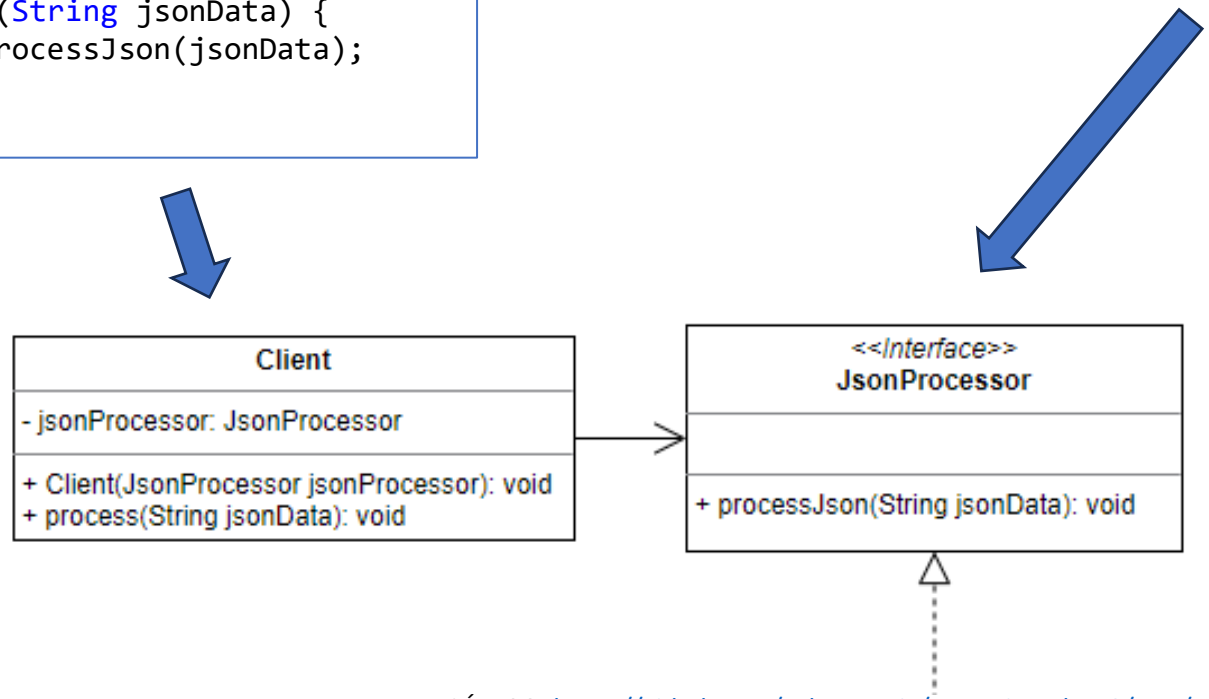
CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/f_Patron_Adapter_XMLtoJASON

3.3 Patrones Estructurales

ADAPTER

```java
// Clase cliente que utiliza JsonProcessor
class Client {
    private JsonProcessor jsonProcessor;

    public Client(JsonProcessor jsonProcessor) {
        this.jsonProcessor = jsonProcessor;
    }

    public void process(String jsonData) {
        jsonProcessor.processJson(jsonData);
    }
}
```

```java
public interface JsonProcessor {
    void processJson(String jsonData);
}
```



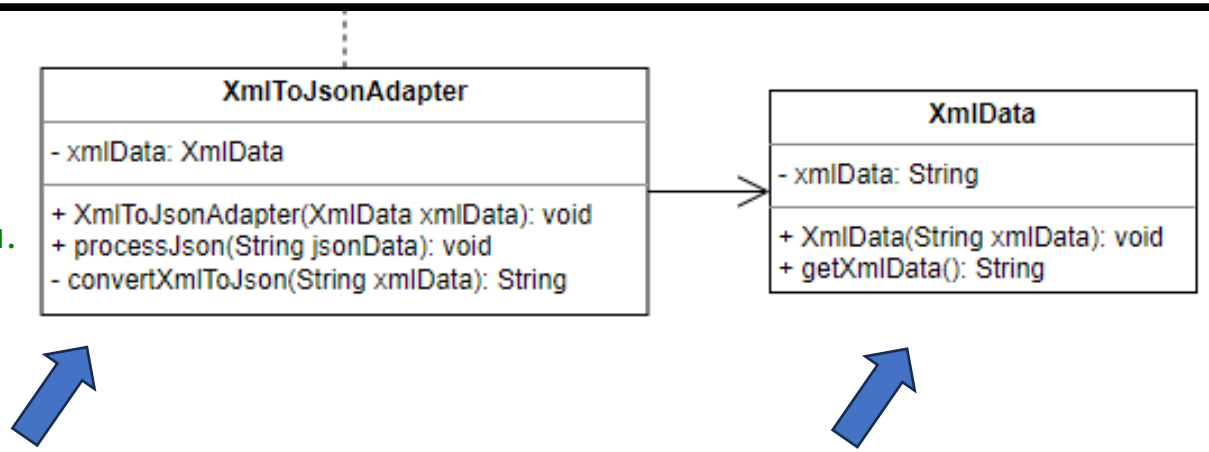
CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/f_Patron_Adapter_XMLtoJASON

3.3 Patrones Estructurales

ADAPTER

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/f_Patron_Adapter_XMLtoJASON

```java
class XmlToJsonAdapter implements JsonProcessor {
    private XmlData xmlData;

    /**
     * Constructor del adaptador que toma XmlData como entrada.
     * @param xmlData Datos XML que necesitan ser convertidos a JSON.
     */
    public XmlToJsonAdapter(XmlData xmlData) {
        this.xmlData = xmlData;
    }

    /**
     * Método que adapta el procesamiento de XML a JSON.
     * @param jsonData String que representa datos en formato JSON.
     */
    @Override
    public void processJson(String jsonData) {
        // Aquí se convertiría XML a JSON (se necesita una implementación
real para la conversión)
        String convertedJson = convertXmlToJson(xmlData.getXmlData());
        // Procesar el JSON convertido
        System.out.println("Procesando JSON: " + convertedJson);
    }

    /**
     * Método simulado para la conversión de XML a JSON
     * @param xml String que representa datos en formato XML.
     * @return String que representa datos convertidos en formato JSON.
     */
    private String convertXmlToJson(String xmlData) {
       // Implementación de la conversión (simulada)
       return "{\"jsonEquivalent\": \"" + xmlData + "\"}";
    }
}
```




```java
class XmlData {
    private String xmlData;

    /**
     * Constructor que inicializa los datos XML.
     * @param xmlData datos en formato XML.
     */
    public XmlData(String xmlData) {
        this.xmlData = xmlData;
    }

    /**
     * Obtiene los datos XML.
     * @return datos en formato XML.
     */
    public String getXmlData() {
        return xmlData;
    }
}
```

3.3 Patrones Estructurales

ADAPTER

```java
/**
 * Demostración del uso del patrón Adapter.
 */
public class MainClass {
    public static void main(String[] args) {
        // Creamos una instancia de XmlData con datos XML de ejemplo.
        XmlData xmlData = new XmlData("<data>Hello XML</data>");

        // Creamos un adaptador que convertirá XML a JSON.
        XmlToJsonAdapter adapter = new XmlToJsonAdapter(xmlData);

        // Utilizamos el adaptador para procesar los datos JSON convertidos.
        Client client = new Client(adapter);
        client.process(xmlData.getXmlData()); // Aquí se pasaría el XML, y el adaptador lo convierte a JSON
    }
}
```

Resultado ejecución → Procesando JSON: {"jsonEquivalent": "<data>Hello XML</data>"}

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/f_Patron_Adapter_XMLtoJASON

Usos comunes

El patrón Adapter es ampliamente utilizado en el desarrollo de software para solucionar problemas de incompatibilidad entre interfaces y facilitar la comunicación entre sistemas o componentes que de otro modo no podrían trabajar juntos. Algunas aplicaciones reales de este patrón incluyen:

- **Integración de Sistemas Externos o de Terceros**: Cuando se integran bibliotecas externas o APIs de terceros en una aplicación existente, a menudo estas no coinciden con las interfaces esperadas por la aplicación. Un adaptador puede convertir la interfaz de la biblioteca externa a una que sea compatible con la aplicación.

- **Migración y Compatibilidad con Sistemas Antiguos**: Durante la migración de sistemas antiguos a tecnologías más modernas, se pueden utilizar adaptadores para garantizar la compatibilidad con el código antiguo, permitiendo que el sistema actualizado interactúe con los componentes heredados.

- **Interfaces de Usuario para Diferentes Dispositivos**: En el desarrollo de interfaces de usuario, especialmente en aplicaciones multiplataforma, los adaptadores pueden ser utilizados para transformar datos o llamadas a métodos de una forma que sea compatible con diferentes dispositivos o sistemas operativos.

- **Conversión de Datos**: Los adaptadores son útiles para convertir datos de un formato a otro, permitiendo que diferentes sistemas o componentes que esperan diferentes formatos de datos puedan trabajar juntos.

3.3 Patrones Estructurales

DECORATOR

Decorator es un patrón de diseño estructural que te permite añadir funcionalidades sin necesidad de usar herencia (apila comportamientos)

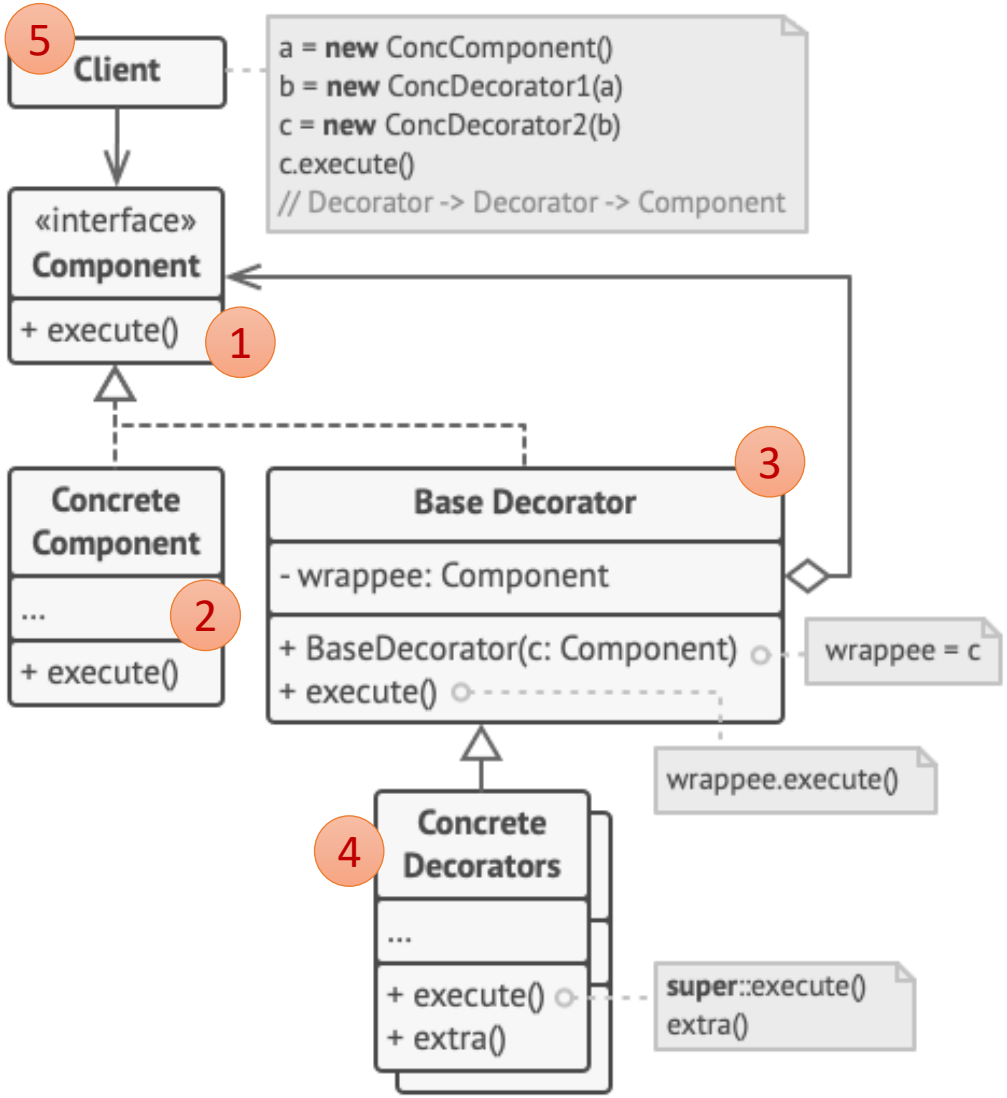
(1) ***Component*** declara la interfaz común tanto para contenedores como para objetos envueltos.

(2) ***Concrete Component*** es una clase de objetos que son "envueltos". Define el comportamiento básico, que puede ser alterado por decoradores.

(3) La clase ***Base Decorator*** tiene un campo para hacer referencia a un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto componentes concretos como decoradores. El decorador base delega todas las operaciones al objeto envuelto.

(4) ***Concrete Decorators*** definen comportamientos adicionales que se pueden agregar a los componentes de manera dinámica. Los *Concrete Decorators* anulan los métodos del *Base Decorator* y ejecutan su comportamiento antes o después de llamar al método principal.

(5) ***Client*** puede envolver componentes en múltiples capas de decoradores, siempre que funcione con todos los objetos a través de la interfaz de componentes.



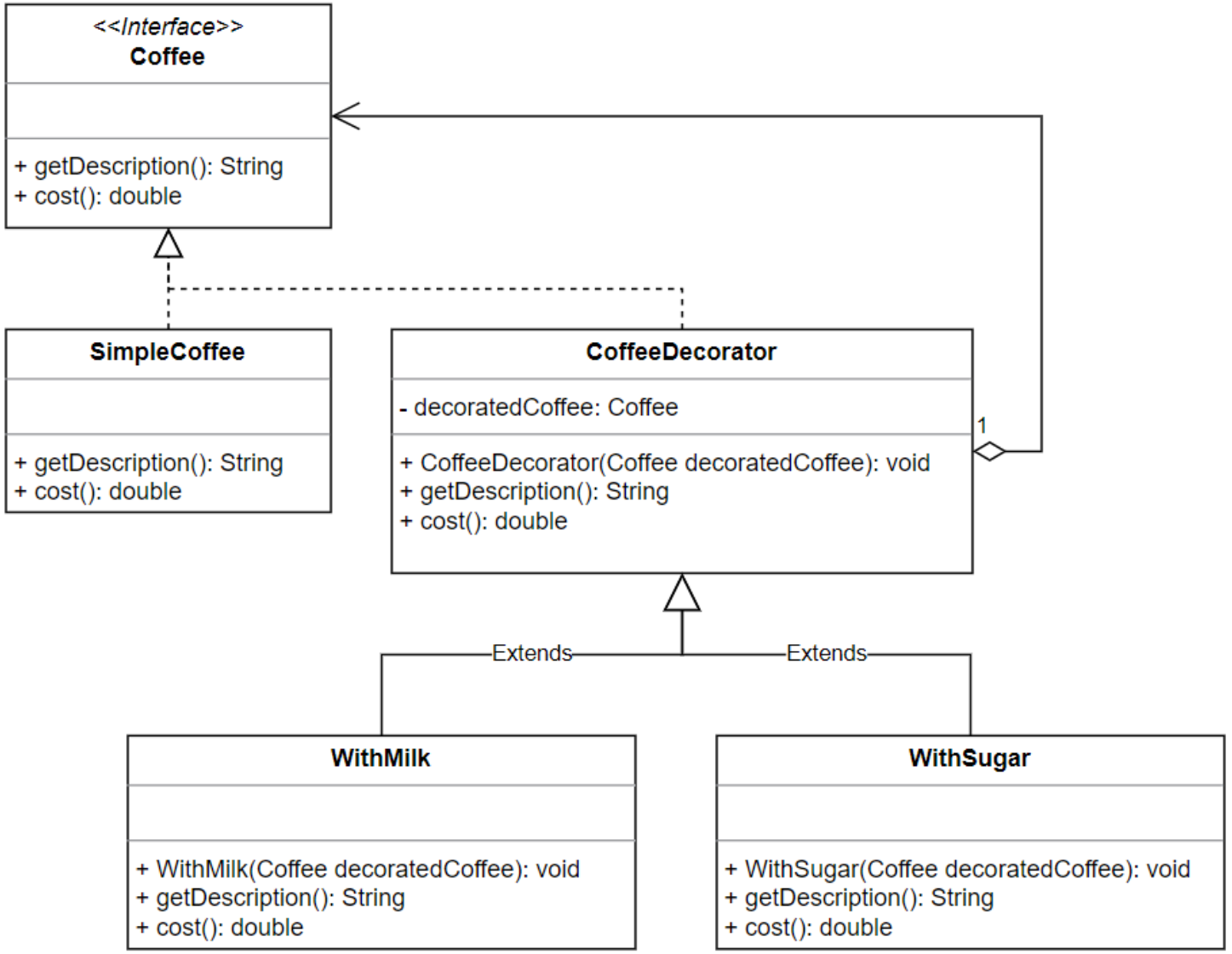
https://refactoring.guru/es/design-patterns/decorator

3.3 Patrones Estructurales

DECORATOR

Ejemplo de cafetería

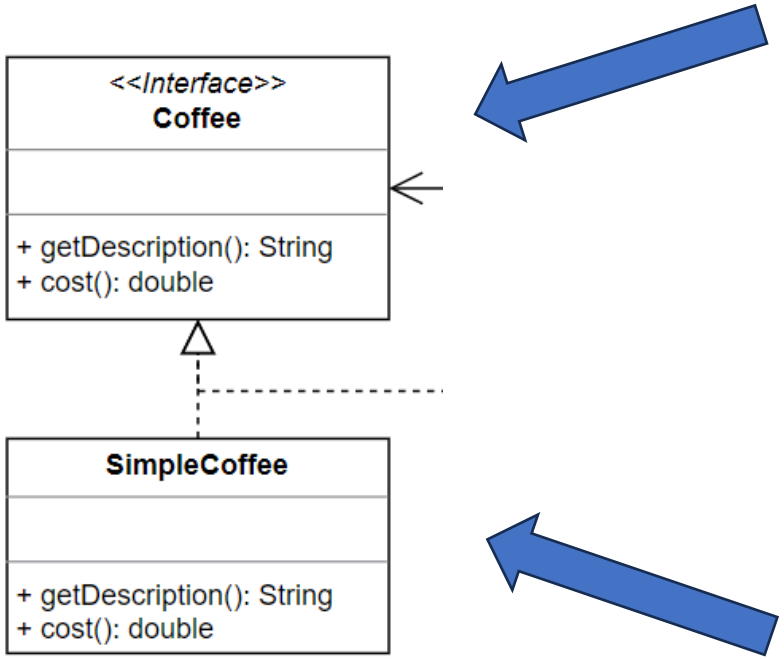
El escenario que vamos a utilizar es el de una cafetería que ofrece diferentes combinaciones de café con varios ingredientes (decoradores), como leche, azúcar, etc.



CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/g_Patron_Decorator_Coffee

3.3 Patrones Estructurales

DECORATOR



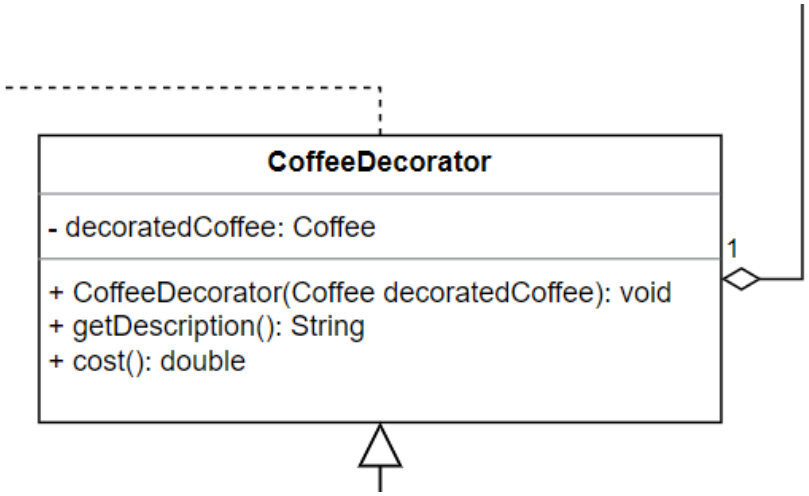

```java
/**
 * La interfaz 'Coffee' define las operaciones de un objeto de café.
 */
interface Coffee {
    public String getDescription();
    public double cost();
}
```

```java
/**
 * Implementación concreta de un objeto 'Coffee' básico.
 */
class SimpleCoffee implements Coffee {

    /**
     * Devuelve una descripción del café simple.
     */
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    /**
     * Devuelve el costo del café simple.
     */
    @Override
    public double cost() {
        return 1.0;
    }
}
```

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/g_Patron_Decorator_Coffee

3.3 Patrones Estructurales

DECORATOR

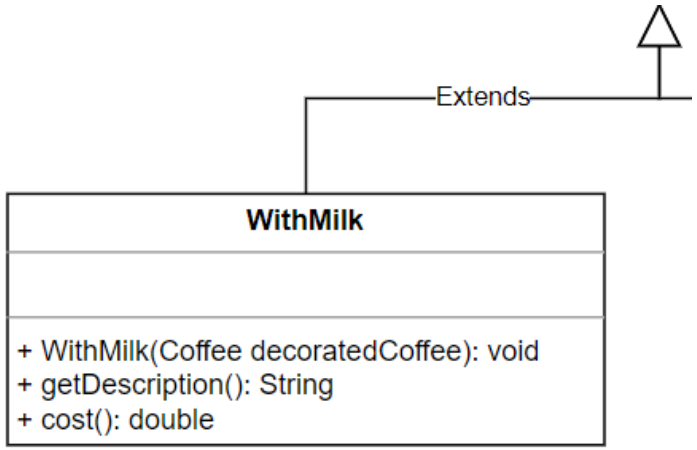



```java
/**
 * Clase abstracta que actúa como la clase base para todos los decoradores.
 */
abstract class CoffeeDecorator implements Coffee {
    private Coffee decoratedCoffee;

    /**
     * Constructor para el decorador de café.
     * @param decoratedCoffee El objeto 'Coffee' que está siendo decorado.
     */
    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    /**
     * Implementa la operación 'getDescription' del componente 'Coffee'.
     */
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    /**
     * Implementa la operación 'cost' del componente 'Coffee'.
     */
    public double cost() {
        return decoratedCoffee.cost();
    }
}
```

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/g_Patron_Decorator_Coffee

3.3 Patrones Estructurales

DECORATOR

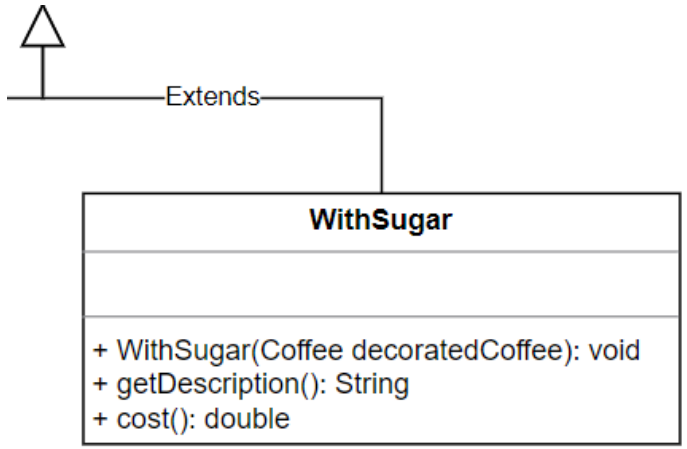



```java
/**
 * Un decorador concreto que añade leche al café.
 */
class WithMilk extends CoffeeDecorator {

    /**
     * Constructor para el decorador 'WithMilk'.
     * @param decoratedCoffee El objeto 'Coffee' que está siendo decorado con leche.
     */
    public WithMilk(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    /**
     * Añade la descripción de la leche al café.
     */
    @Override
    public String getDescription() {
        return super.getDescription() + ", Milk";
    }

    /**
     * Añade el costo de la leche al costo total del café.
     */
    @Override
    public double cost() {
        return super.cost() + 0.5;
    }
}
```

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/g_Patron_Decorator_Coffee

3.3 Patrones Estructurales

DECORATOR




```java
/**
 * Un decorador concreto que añade azúcar al café.
 */
class WithSugar extends CoffeeDecorator {

    /**
     * Constructor para el decorador 'WithSugar'.
     * @param decoratedCoffee El objeto 'Coffee' que está siendo decorado con azúcar.
     */
    public WithSugar(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    /**
     * Añade la descripción del azúcar al café.
     */
    @Override
    public String getDescription() {
        return super.getDescription() + ", Sugar";
    }

    /**
     * Añade el costo del azúcar al costo total del café.
     */
    @Override
    public double cost() {
        return super.cost() + 0.2;
    }
}
```

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/g_Patron_Decorator_Coffee

3.3 Patrones Estructurales

DECORATOR

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/g_Patron_Decorator_Coffee

```java
/**
 * Clase de demostración para mostrar el patrón Decorator en acción.
 */
public class MainClass {

    /**
     * Punto de entrada principal del programa.
     */
    public static void main(String[] args) {
        // Preparar un simple café
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " Cost: $" + coffee.cost());

        // Decorar el café con leche
        Coffee milkCoffee = new WithMilk(coffee);
        System.out.println(milkCoffee.getDescription() + " Cost: $" + milkCoffee.cost());

        // Decorar el café con leche y azúcar
        Coffee milkSugarCoffee = new WithSugar(milkCoffee);
        System.out.println(milkSugarCoffee.getDescription() + " Cost: $" + milkSugarCoffee.cost());
    }
}
```

Resultado ejecución →

```
Simple Coffee Cost: $1.0
Simple Coffee, Milk Cost: $1.5
Simple Coffee, Milk, Sugar Cost: $1.7
```

3.3 Patrones Estructurales

DECORATOR

El patrón Decorator se utiliza en este ejemplo para añadir responsabilidades adicionales a los objetos de manera dinámica. El programa simula una cafetería que puede servir café, con la posibilidad de añadir ingredientes adicionales.

Componentes del patrón

- **Componente Concreto (SimpleCoffee)**: Es la implementación básica de la interfaz **Coffee**. Define el objeto a decorar. En nuestro caso, es un café simple con una descripción y un costo base.
- **Decorador Base (CoffeeDecorator)**: Es una clase abstracta que implementa la interfaz **Coffee** y contiene una referencia a un objeto **Coffee**. Sirve como la clase base para todos los decoradores concretos.
- **Decoradores Concretos (WithMilk, WithSugar)**: Son las implementaciones que extienden el decorador base. Estos decoradores añaden funcionalidades (como añadir leche o azúcar) al objeto **Coffee** que envuelven, modificando su descripción y costo.
- **Cliente (DecoratorPatternDemo)**: Es la clase que demuestra el uso del patrón. Crea un **SimpleCoffee** y luego lo decora con **WithMilk** y **WithSugar**, mostrando cómo se pueden añadir dinámicamente características adicionales a los objetos.

Cómo funciona

- Se crea un **SimpleCoffee**, que es nuestro componente concreto con su propio costo y descripción.
- Luego, se "envuelve" este **SimpleCoffee** en un **WithMilk**, que es un decorador. Esto añade la descripción de la leche y su costo al café.
- A continuación, se envuelve el **WithMilk** (que ya contiene un **SimpleCoffee**) en un **WithSugar**, agregando otra capa de decoración. Esto añade la descripción del azúcar y su costo adicional.
- El cliente (en el método **main**) solo interactúa con la interfaz **Coffee**, pero gracias a los decoradores, se pueden combinar múltiples comportamientos.

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/g_Patron_Decorator_Coffee

Usos comunes

El patrón Decorator es un patrón de diseño estructural que permite añadir funcionalidades a objetos de manera dinámica, sin alterar su estructura. Este patrón es ampliamente utilizado en aplicaciones reales, y algunas de sus aplicaciones más comunes incluyen:

- **Interfaces de Usuario (UI)**: En el desarrollo de interfaces gráficas, el patrón Decorator se utiliza para añadir dinámicamente comportamientos o estilos a elementos de la UI, como botones o campos de texto, sin modificar el código de los elementos en sí.

- **Manipulación de Streams de Entrada/Salida (I/O)**: En Java y otros lenguajes de programación, el patrón Decorator se usa extensamente en las bibliotecas de I/O para añadir funcionalidades como el buffer, la compresión, la codificación y la encriptación a los streams de datos.

- **Desarrollo Web**: En el diseño de sitios web, se pueden utilizar decoradores para modificar o añadir funcionalidades a componentes web, como agregar eventos de seguimiento o mejorar la seguridad de formularios y solicitudes.

- **Desarrollo de Juegos**: En el desarrollo de videojuegos, el patrón Decorator permite modificar las propiedades de los personajes o elementos del juego, como añadir habilidades, efectos o modificar su apariencia, sin cambiar la clase base del personaje o elemento.

Ejercicios

3.3 Patrones Estructurales

Ejercicios

Enunciado del Ejercicio: **El Guerrero Decorado**

Implementar el patrón de diseño Decorator para simular la equipación de un personaje de juego con varios ítems de inventario. El personaje comenzará con una equipación básica, y podrás añadirle dinámicamente ítems como una espada, una armadura y un escudo.

Pasos a seguir:

1. Crear el Componente Base: Define una interfaz Character con un método equip() que imprimirá la equipación actual del personaje.
2. Implementar el Componente Concreto: Crea una clase BasicCharacter que implemente Character, representando un personaje con equipación básica.
3. Crear el Decorador Base: Diseña una clase abstracta CharacterDecorator que implemente Character y contenga una referencia a un objeto Character.
4. Implementar Decoradores Concretos: Desarrolla clases decoradoras concretas (SwordDecorator, ArmorDecorator, ShieldDecorator) que extiendan CharacterDecorator y añadan funcionalidades específicas al método equip().
5. Demostración: En la clase principal, crea una instancia de BasicCharacter y decórala sucesivamente con los decoradores creados, imprimiendo la descripción final del personaje después de cada decoración.

Bibliografía

- Gamma, E., Johnson, R., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

- **Sumérgete en los patrones de diseño.** V2021-1.7. Alexander Shvets.

  https://refactoring.guru/es/design-patterns/book

  Versión online: https://refactoring.guru/es/design-patterns/catalog

- Freeman, E., Robson, E., Bates, B., & Sierra, K. (2008). *Head first design patterns*. " O'Reilly Media, Inc.".

- Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, *1*(34), 597.

Técnicas de Programación Avanzada

```
exit(); //Gracias!
```