PATRONES DE DISEÑO

Saúl Crespo Saldaña

2º DAM CEINMARK
Acceso a datos

INTRODUCCIÓN:

Los patrones de diseño son soluciones probadas a problemas que ocurren con frecuencia en el proceso de desarrollo de software. A lo largo de los años, al enfrentarse a problemas similares, ingenieros y desarrolladores de software han ido creando soluciones similares. Estas soluciones se han ido solidificando en arquetipos y estructuras (patrones) que se han convertido en un lenguaje común en el diseño de sistemas, lo que ha posibilitado una mejor comunicación entre ingenieros de diferentes empresas, ramas o escuelas de desarrollo de software.

Conocer estos patrones permite encontrar más rápidamente soluciones a los problemas que surgen cuando el tamaño de las aplicaciones crezcan y poder anticiparse a los problemas que vayan apareciendo. El objetivo de este trabajo es dar una visión general de los patrones de diseño que se utilizan en la Programación Orientada a Objetos: su clasificación, y el catálogo de los mismos. También se examinarán de cerca tres de estos patrones (uno de cada familia): los patrones Builder, Chain Of Responsability y Adapter.

PATRONES CREACIONALES:

Los patrones de diseño creacionales son los que se ocupan de resolver problemas en cuanto a cómo se crean los objetos. Estos patrones pueden determinar la manera en la que se crean, la cantidad de instancias que puede haber de una clase… A continuación se citan los patrones creacionales más conocidos:

- SINGLETON: Una clase de la cual sólo puede haber una instancia en toda la aplicación.
- PROTOTYPE: Permite copiar objetos existentes sin que el código dependa de sus clases.
- FACTORY METHOD: Proporciona la interfaz para crear objetos en una superclase, dejando que sean las subclases las que determinen los objetos que se crean en función del entorno.
- ABSTRACT FACTORY: Permite crear familias de objetos relacionados sin necesidad de especificar sus clases concretas.
- BUILDER: Permite crear objetos paso a paso.

Elegir el patrón creacional correcto para nuestras aplicaciones nos ayudará a delegar las responsabilidades de la creación de diferentes objetos y a simplificar la jerarquía de clases evitando árboles de herencia complejos donde la refactorización resulte demasiado compleja.

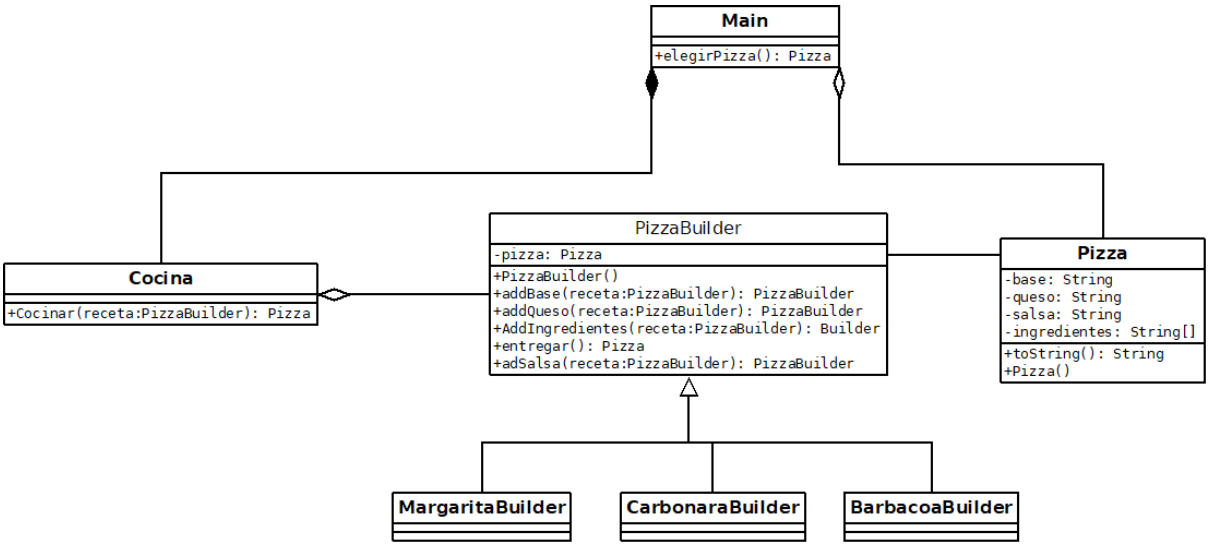
BUILDER:

A diferencia del Factory, que crea objetos predefinidos en función del entorno de la aplicación, el patrón builder nos permite crear objetos paso por paso, (como si fuera un constructor que va construyendo una casa poco a poco). Esto nos permite tener una sóla clase para un tipo de objeto, en lugar de una jerarquía muy compleja que luego va a ser muy difícil de refactorizar si necesitamos hacer cambios, y dejarle toda la variedad a

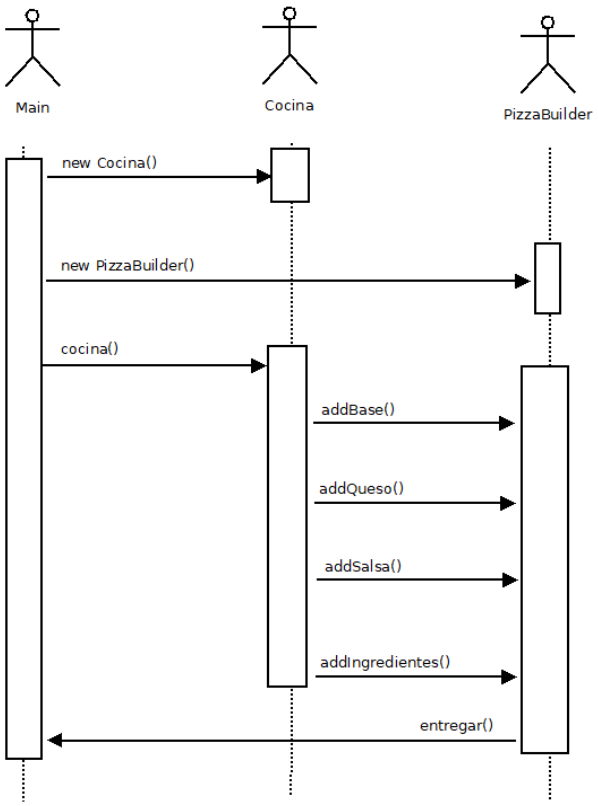
diferentes clases Builder que van a construir las diferentes instancias de la misma en función de lo que necesitemos.

El ejemplo más sencillo es imaginar que nuestra aplicación gestiona un restaurante italiano y queremos crear pizzas para nuestros clientes. Una aproximación sería crear una clase Pizza, y luego una serie de subclases que fueran Margarita, Barbacoa, Carbonara… y así una subclase nueva por cada tipo de pizza que tenemos. Esta aproximación se complicaría mucho cuando nuestra aplicación crezca, ya que resultaría en una estructura de clases muy compleja donde hacer cambios sería demasiado difícil.

El patrón builder en cambio, nos permite crear una sóla clase Pizza, y luego varias clases Builder, la cual crea cada una una nueva Pizza añadiendo los ingredientes correspondientes.



De esta forma, tanto en la Cocina como en el Main, sólo nos tenemos que preocupar de llamar al Builder correspondiente sin necesidad de saber todos los ingredientes y el proceso que lleva crear la pizza, y dejamos que el Builder correspondiente se encargue del resto:



Pizza.java

```
package builder;

public abstract class PizzaBuilder {
	Pizza pizza;

	public PizzaBuilder() {
		pizza = new Pizza();
	}

	public abstract PizzaBuilder addBase(PizzaBuilder receta);

	public abstract PizzaBuilder addQueso(PizzaBuilder receta);

	public abstract PizzaBuilder addSalsa(PizzaBuilder receta);

	public abstract PizzaBuilder addIngredientes(PizzaBuilder receta);

	public Pizza entregar() {
		return this.pizza;
	}
}
```

PizzaBuilder.java

```
package builder;

public abstract class PizzaBuilder {
	Pizza pizza;

	public PizzaBuilder() {
		pizza = new Pizza();
	}

	public abstract PizzaBuilder addBase(PizzaBuilder receta);

	public abstract PizzaBuilder addQueso(PizzaBuilder receta);

	public abstract PizzaBuilder addSalsa(PizzaBuilder receta);

	public abstract PizzaBuilder addIngredientes(PizzaBuilder receta);

	public Pizza entregar() {
		return this.pizza;
	}
}
```

CarbonaraBuilder.java

```
package builder;

public class CarbonaraBuilder extends PizzaBuilder {

	public CarbonaraBuilder() {
		super();
	}

	@Override
	public PizzaBuilder addBase(PizzaBuilder receta) {
		receta.pizza.setBase("Nata");
		return receta;
	}

	@Override
	public PizzaBuilder addQueso(PizzaBuilder receta) {
		receta.pizza.setQueso("Rayado");
		return null;
	}

	@Override
	public PizzaBuilder addSalsa(PizzaBuilder receta) {
		receta.pizza.setSalsa("ninguna");
		return null;
```

```
	}

	@Override
	public PizzaBuilder addIngredientes(PizzaBuilder receta) {
		String[] ingredientes = {"Cebolla","Champiñones","Bacon"};
		receta.pizza.setIngredientes(ingredientes);
		return receta;
	}
}
```

BarbacoaBuilder.java

```
package builder;

public class BarbacoaBuilder extends PizzaBuilder {

	public BarbacoaBuilder() {
		super();
	}

	@Override
	public PizzaBuilder addBase(PizzaBuilder receta) {
		receta.pizza.setSalsa("Tomate");
		return receta;
	}

	@Override
	public PizzaBuilder addQueso(PizzaBuilder receta) {
		receta.pizza.setQueso("Havarti y Mozzarella rallado");
		return receta;
	}

	@Override
	public PizzaBuilder addSalsa(PizzaBuilder receta) {
		receta.pizza.setSalsa("Barbacoa");
		return receta;
	}

	@Override
	public PizzaBuilder addIngredientes(PizzaBuilder receta) {
		String[] ingredientes = {"Carne picada","Pimiento rojo","Cebolleta","Tomate
natural"};
		receta.pizza.setIngredientes(ingredientes);
		return receta;
	}
}
```

MargaritaBuilder.java

```
package builder;

public class MargaritaBuilder extends PizzaBuilder {

	public MargaritaBuilder() {
		super();
	}

	@Override
	public PizzaBuilder addBase(PizzaBuilder receta) {
		receta.pizza.setBase("Tomate");
		return receta;
	}

	@Override
	public PizzaBuilder addQueso(PizzaBuilder receta) {
		receta.pizza.setQueso("Mozzarella");
		return receta;
	}

	@Override
	public PizzaBuilder addSalsa(PizzaBuilder receta) {
		receta.pizza.setSalsa("Aceite");
		return receta;
	}

	@Override
	public PizzaBuilder addIngredientes(PizzaBuilder receta) {
		String[] ingredientes = {"Albahaca"};
		receta.pizza.setIngredientes(ingredientes);
		return receta;
	}
}
```

Cocina.java

```
package builder;

public class Cocina {
	public Pizza cocinar(PizzaBuilder receta) {
		// Dado que todos los métodos de la clase
		//	PizzaBuilder retornan un objeto PizzaBuilder,
		//	estos métodos se pueden encadenar para simplificar
		//	el código e ir añadiendo los diferentes ingredientes
		//  hasta que la pizza está completa, y una vez completa,
		//  se entrega, devolviendo el nuevo objeto pizza.
		return receta.addIngredientes(
					receta.addSalsa(
```

```
                                        receta.addQueso(
                                         receta.addBase(receta)))
                                       ).entregar();
        }
}
```

Main.java

```
package builder;

import java.util.Scanner;

public class Main {
        public static void main(String[] args) {
                Pizza pizza = null;
                Cocina cocina = new Cocina();
                Scanner scan = new Scanner(System.in);
                System.out.println("¿Qué tipo de pizza desea usted?"
                                + "\n1=Margarita\t2=Carbonara\t3=Barbacoa");
                switch(scan.nextInt()) {
                        case 1 -> pizza = cocina.cocinar(new MargaritaBuilder());
                        case 2 -> pizza = cocina.cocinar(new CarbonaraBuilder());
                        case 3 -> pizza = cocina.cocinar(new BarbacoaBuilder());
                }
                System.out.println("Usted va a comer: ");
                System.out.println(pizza);

        }
}
```

Consola

```
¿Qué tipo de pizza desea usted?
1=Margarita    2=Carbonara  3=Barbacoa
3
Usted va a comer:
Una deliciosa pizza con base de Tomate, queso Havarti y Mozzarella rallado, salsa
Barbacoa y los siguientes ingredientes:
[Carne picada, Pimiento rojo, Cebolleta, Tomate natural]
```

PATRONES DE COMPORTAMIENTO:

Los patrones de diseño de comportamiento se ocupan de los diferentes algoritmos y de manejar las responsabilidades y el comportamiento de los diferentes objetos dentro de una aplicación.

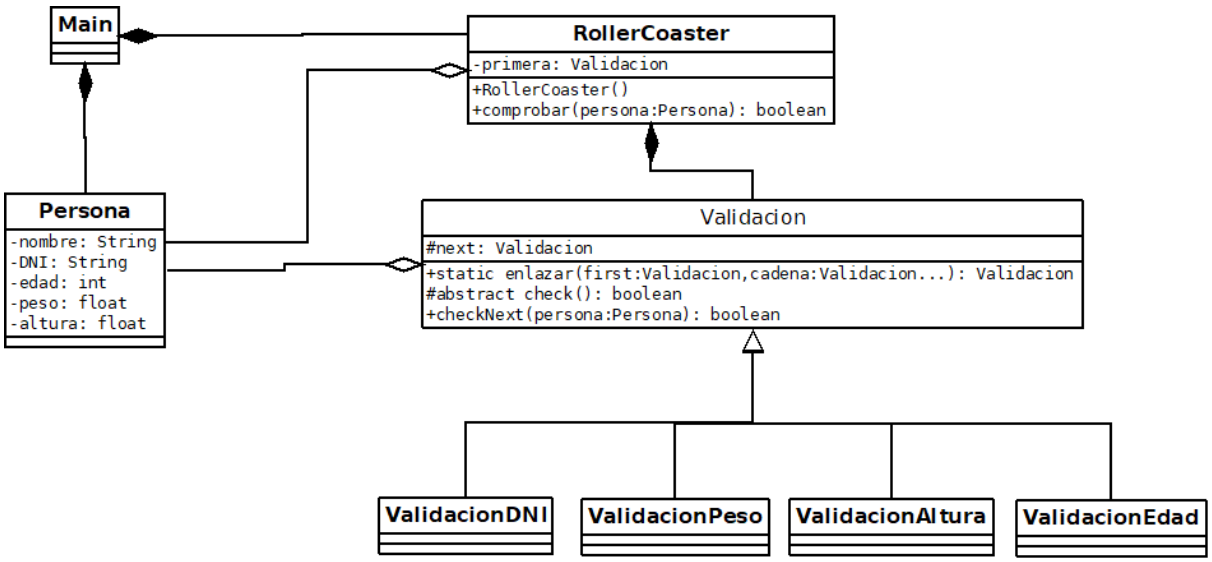
- CHAIN OF RESPONSABILITY: Permite pasar solicitudes a lo largo de una cadena de manejadores, los cuales deciden si procesarla, o si pasárselo al siguiente.
- COMMAND: Convierte cada acción en objetos independientes que se pueden manejar de manera uniforme. Permite crear un historial de las acciones en la aplicación, ordenar las acciones en colas y retrasar o adelantarlas.
- ITERATOR: Permite atravesar una colección de objetos, sin necesidad de exponer su estructura subyacente.
- MEDIATOR: Provee de un marco para que clases diferentes se relacionen a través de la misma interfaz, disminuyendo el acoplamiento de la aplicación y facilitando futuras refactorizaciones.
- MEMENTO: Permite guardar el estado de un objeto para poder restaurarlo sin revelar los detalles de la implementación.
- OBSERVER: Permite crear un mecanismo para notificar a una serie de objetos de los cambios en este objeto mediante un mecanismo de “suscripción”.
- STATE: Permite a un objeto cambiar su comportamiento cuando su estado interno cambia.
- STRATEGY: Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase y hacerlas intercambiables.
- TEMPLATE METHOD: Define el esqueleto de un algoritmo, y permite a sus subclases sobreescribir pasos concretos.
- VISITOR: Permite separar los algoritmos de los objetos sobre los que operan.

Elegir buenos patrones de comportamiento para nuestras aplicaciones nos ayuda a mejorar la eficiencia de las mismas y a manejar con mayor facilidad nuestra estructura de clases y las relaciones entre las mismas a medida que nuestra aplicación crece.

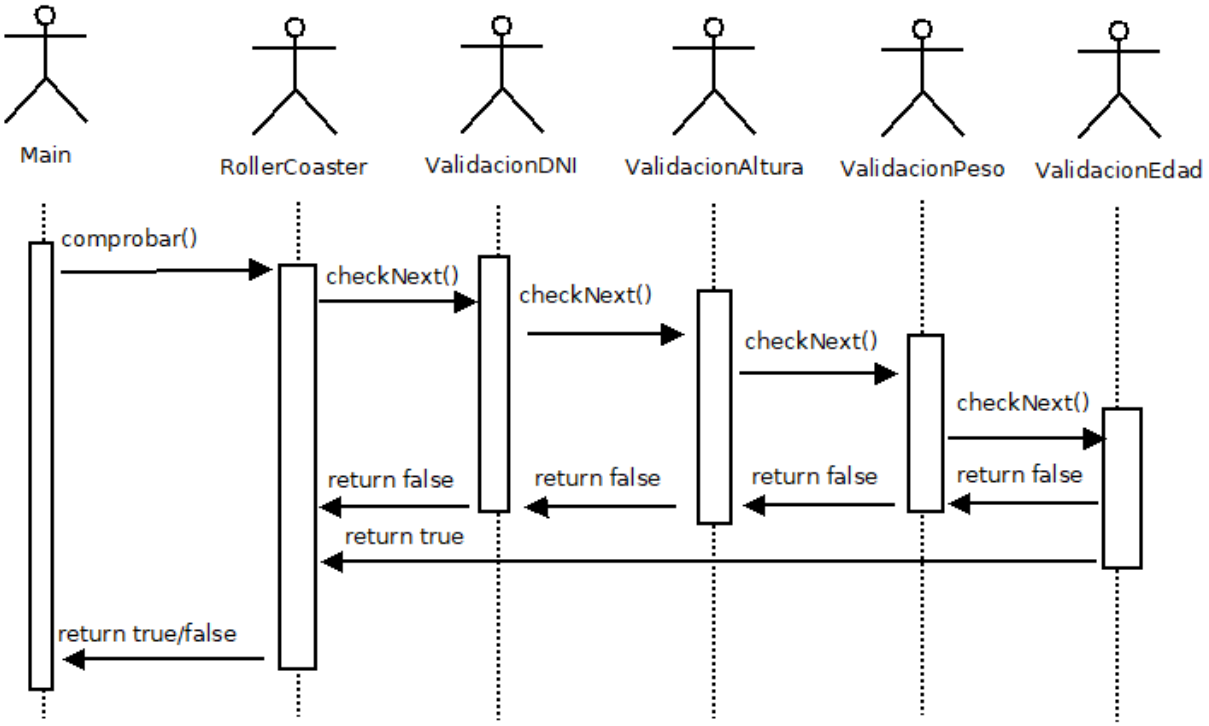
CHAIN OF RESPONSABILITY:

El patrón Chain Of Responsability (cadena de responsabilidad) consiste en una cadena de objetos que se van pasando información de uno a otro para tratarla y validarla hasta conseguir el resultado final. Cada objeto debe contener una referencia del siguiente objeto de la cadena, al cual le pasará la información a través de un método. Se suele utilizar en procesos como validaciones de datos de usuarios o pasarelas de pago.

Como ejemplo, vamos a estudiar como en una montaña rusa podríamos comprobar si una persona se puede subir o no usando este patrón. Necesitaremos crear una clase abstracta que represente las validaciones, y luego las validaciones concretas. Las validaciones deberán contener un puntero a la siguiente validación, para poder realizar la cadena de operaciones enlazadas y que simplemente apuntando a la primera se realizen todas.



De esta forma, cuando se realicen las comprobaciones, se irán realizando en cadena, pasando el objeto persona que queremos validar de una clase a otra hasta que termine la cadena, de la siguiente forma:



Persona.java

```
package chainofresponsability;

public class Persona {
	private String nombre, DNI;
	private int edad;
	private float peso;
	private float altura;
```

```
	public Persona(String nombre, String dNI, int edad, float peso, float altura) {
		this.nombre = nombre;
		DNI = dNI;
		this.edad = edad;
		this.peso = peso;
		this.altura = altura;
	}

	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getDNI() {
		return DNI;
	}
	public void setDNI(String dNI) {
		DNI = dNI;
	}
	public int getEdad() {
		return edad;
	}
	public void setEdad(int edad) {
		this.edad = edad;
	}
	public float getPeso() {
		return peso;
	}
	public void setPeso(float peso) {
		this.peso = peso;
	}
	public float getAltura() {
		return altura;
	}
	public void setAltura(float altura) {
		this.altura = altura;
	}
}
```

Validacion.java

```
package chainofresponsability;

public abstract class Validacion {
	// La dirección de memoria donde se guarda la siguiente validación
	protected Validacion next;
```

```
	public static Validacion enlazar(Validacion first, Validacion... cadena) {
		Validacion head = first; // Asigno la primera validación a la variable head
		for (Validacion siguiente:cadena) { // Por todos los elementos de la cadena
			head.next = siguiente; // Le asigno un elemento siguiente al head
actual
			head = siguiente; // Meto el siguiente elemento de la cabeza en
head
		}
		return first; // Devuelvo el primer elemento de la cadena, ya enlazado con
		// los demás.
	}

	// La comprobación que hace cada Validación de la cadena, diferente en cada una
	protected abstract boolean check(Persona persona);

	// Comprueba si quedan objetos en la cadena, y si no, devuelve true
	public boolean checkNext(Persona persona) {
		if (this.next==null) { // Si el objeto siguiente a este no existe...
			return true; // devuelve true
		}
		// si no...
		return next.check(persona); // Le pasa el turno a la siguiente Validación de
la cadena.
	}
}
```

ValidacionDNI.java

```
package chainofresponsability;

public class ValidacionDNI extends Validacion {

	@Override
	public boolean check(Persona persona) {
		return(persona.getDNI().length()==9)?
				checkNext(persona):false;
	}
}
```

ValidacionAltura.java

```
package chainofresponsability;

public class ValidacionAltura extends Validacion {

	@Override
	public boolean check(Persona persona) {
	return (persona.getAltura() > 1.20 &&
```

```
                        persona.getAltura() < 2.10)?
                        checkNext(persona):false;
        }

}
```

ValidacionPeso.java

```
package chainofresponsability;

public class ValidacionPeso extends Validacion {

        @Override
        public boolean check(Persona persona) {
                return (persona.getPeso() > 45 &&
                                persona.getPeso() <120)?
                                checkNext(persona):false;
        }
}
```

ValidacionEdad.java

```
package chainofresponsability;

public class ValidacionEdad extends Validacion {

        @Override
        public boolean check(Persona persona) {
                return (persona.getEdad()>10)?checkNext(persona):false;
        }
}
```

RollerCoaster.java

```
package chainofresponsability;

public class RollerCoaster {
        // Espacio donde guardamos la primera validacion de la cadena
        private Validacion primera;

        RollerCoaster() {
                // Creamos la cadena de validaciones enlazadas
                primera = Validacion.enlazar(
                                new ValidacionDNI(),
```

```
                                        new ValidacionAltura(),
                                        new ValidacionPeso(),
                                        new ValidacionEdad());
        }

        // Comienza la cadena de validaciones, empezando por la primera
        public boolean comprobar(Persona persona) {
                return primera.check(persona);
                // Al estar todas las validaciones enlazadas, seguirá la
                // cadena hasta el final. Si en la última comprueba que la
                // siguiente es null, devolverá true. Si no pasa alguna de
                // ellas irá devolviendo false de una en una.
        }
}
```

Main.java

```
package chainofresponsability;

public class Main {
        public static void main(String[] args) {
                RollerCoaster DragonKhan = new RollerCoaster();
                Persona adecuada = new Persona("Saul","12345678A",31,78.0f,1.74f);
                Persona noAdecuada = new Persona("Pancracio","12345678A",31,135.0f,1.74f);
                if (DragonKhan.comprobar(adecuada)) {
                        System.out.println(adecuada.getNombre() + " se sube a la montaña
rusa.");
                } else {
                        System.out.println(adecuada.getNombre() + " no se puede subir a la
montaña rusa.");
                }

                if (DragonKhan.comprobar(noAdecuada)) {
                        System.out.println(noAdecuada.getNombre() + " se sube a la montaña
rusa.");
                } else {
                        System.out.println(noAdecuada.getNombre() + " no se puede subir a la
montaña rusa.");
                }
        }
}
```

Consola

```
Saul se sube a la montaña rusa.
Pancracio no se puede subir a la montaña rusa.
```

PATRONES ESTRUCTURALES:

Los patrones estructurales se ocupan de cómo ensamblar objetos y clases en estructuras más grandes, manteniendo la flexibilidad y la eficiencia de las mismas. Los más conocidos son los siguientes:

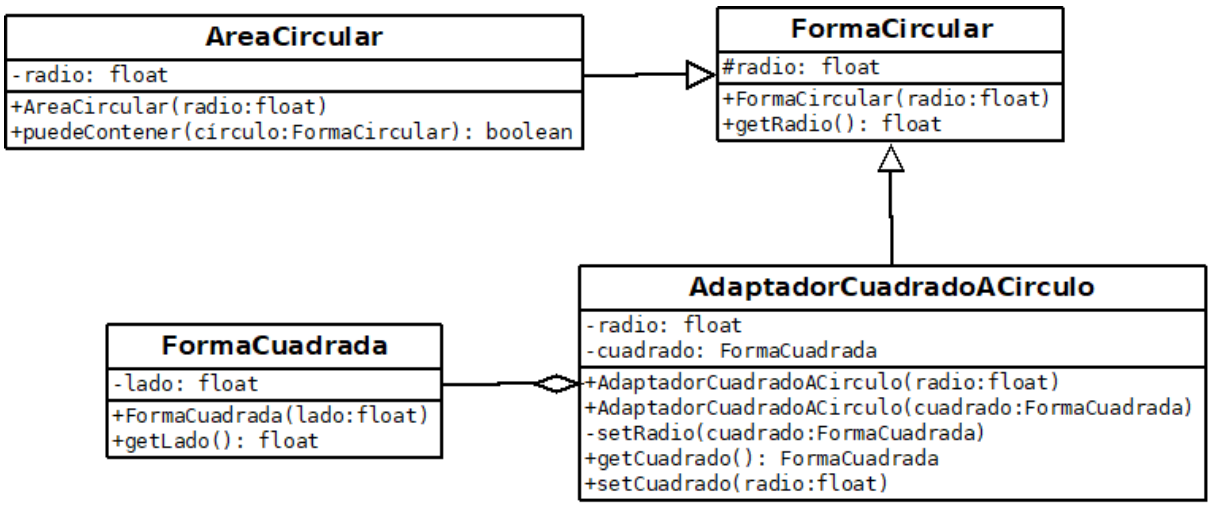
- ADAPTER: Permite la colaboración de objetos e interfaces incompatibles.
- BRIDGE: Permite dividir una jerarquía de clases muy grande en dos o más jerarquías pequeñas, que son mas fáciles de entender y de refactorizar.
- COMPOSITE: Permite organizar objetos en una estructura de árbol, tratándolos de manera uniforme.
- DECORATOR: Permite añadirle funcionalidades a objetos metiéndolos dentro de otros objetos o clases envolventes.
- FACADE: Oculta los detalles de un subsistema de clases complejo y presenta una interfaz simplificada para que el que necesite utilizarla no tenga que preocuparse de los detalles de la implementación.
- FLYWEIGHT: Permite almacenar más objetos en la memoria RAM, compartiendo las partes comunes de los mismos.
- PROXY: Permite crear un sustituto del objeto con el que quieres trabajar, de tal forma que puedas realizar acciones antes o después de acceder a los mismos.

Estos patrones permiten tener una estructura del código más ordenada, eficiente y fácil de mantener a largo plazo.

ADAPTER:

El adapter es un patrón de diseño que surge de la necesidad de trabajar con dos interfaces a priori incompatibles. Esto suele suceder en el desarrollo software cuando trabajamos con librerías de terceros o formatos de archivos predefinidos diferentes que vienen de APIs distintas, y tenemos la necesidad de trabajar con ambas. En estos casos, la opción que nos queda es crear una clase intermedia, que nos permita realizar operaciones entre las diferentes clases que queremos relacionar. El caso es similar a cuando en la vida real compramos un portátil: para cargar la batería no lo podemos conectar directamente a la red eléctrica, ya que el enchufe y el voltaje son incompatibles. Es por esto que necesitamos un adaptador para cuando queremos enchufar el portátil a la red para cargarlo.

Para realizar un ejemplo sencillo, imaginemos que tenemos una clase que define un área circular, que contiene un método para saber si una forma circular cabe dentro de la misma comparando ambos radios. Esta clase no la podemos cambiar, ya que nos ha venido dada por una librería de terceros. Ahora bien, nosotros necesitamos comprobar si una forma cuadrada puede caber completamente dentro de dicha área. La clase para la forma cuadrada nos viene dada por otra librería de terceros que tampoco podemos modificar. La forma más adecuada sería crear una clase intermedia, que pueda realizar la conversión entre las formas cuadrada y redonda, para poder compararlas mediante el método de la clase área circular. Esta sería la estructura básica que seguiría este patrón:



FormaCircular.java

```
package adapter;

public class FormaCircular {
	protected float radio;

	public FormaCircular() {

	}

	public FormaCircular(float radio) {
		this.radio = radio;
	}

	public float getRadio() {
		return radio;
	}
}
```

AreaCircular.java

```
package adapter;

public class AreaCircular extends FormaCircular {

	public AreaCircular(float radio) {
		super(radio);
	}

	public boolean puedeContener(FormaCircular circulo) {
		return radio > circulo.getRadio();
	}
```

```
}
```

FormaCuadrada.java

```
package adapter;

public class FormaCuadrada {
	private float lado;

	FormaCuadrada(float lado) {
		this.lado = lado;
	}

	public float getLado() {
		return lado;
	}
}
```

AdaptadorCuadradoACirculo.java

```
package adapter;

public class AdaptadorCuadradoACirculo extends FormaCircular {
	private FormaCuadrada cuadrado;

	public AdaptadorCuadradoACirculo(float radio) {
		super(radio);
		setCuadrado(radio);
	}

	public AdaptadorCuadradoACirculo(FormaCuadrada cuadrado) {
		this.cuadrado = cuadrado;
		setRadio(cuadrado);
	}

	private void setRadio(FormaCuadrada cuadrado) {
		this.radio = (float) (cuadrado.getLado() * Math.sqrt(2) / 2);
	}

	public FormaCuadrada getCuadrado() {
		return cuadrado;
	}

	public void setCuadrado(float radio) {
		cuadrado = new FormaCuadrada( radio * 2 /  (float) Math.sqrt(2));
	}
}
```

Main.java

```
package adapter;

public class Main {
	public static void main(String[] args) {
		AreaCircular area = new AreaCircular(20.0f);
		FormaCircular circuloMenor = new FormaCircular(10.0f);
		FormaCircular circuloMayor = new FormaCircular(30.0f);
		FormaCuadrada cuadradoMenor = new FormaCuadrada(10.0f);
		FormaCuadrada cuadradoMayor = new FormaCuadrada(30.0f);

		System.out.print("Comprobando si círculo pequeño cabe:\t");
		System.out.println(area.puedeContener(circuloMenor));
		System.out.print("Comprobando si círculo grande cabe:\t");
		System.out.println(area.puedeContener(circuloMayor));
		System.out.print("Comprobando si cuadrado pequeño cabe:\t");
		System.out.println(area.puedeContener(new AdaptadorCuadradoACirculo(cuadradoMenor)));
		System.out.print("Comprobando si cuadrado grande cabe:\t");
		System.out.println(area.puedeContener(new AdaptadorCuadradoACirculo(cuadradoMayor)));
	}
}
```

Consola

```
Comprobando si círculo pequeño cabe:	true
Comprobando si círculo grande cabe:	false
Comprobando si cuadrado pequeño cabe:	true
Comprobando si cuadrado grande cabe:	false
```

CONCLUSIONES:

Los patrones de diseño son una herramienta muy poderosa y necesaria para el momento en el que nuestras aplicaciones comienzan a crecer, ya que nos permiten manejar volúmenes mucho mayores de código de una forma más ordenada y evitando que nuestro código se convierta en un monstruo incomprensible e imposible de mantener.

No obstante, debemos tener en cuenta de que no son un dogma que deba respetarse totalmente en todos los casos. Esto quiere decir que debemos usar el patrón correcto cuando realmente estemos seguros de que va a mejorar la eficiencia, la claridad y la mantenibilidad de nuestro código, ya que en caso contrario, podemos acabar haciendo las aplicaciones más complejas de lo que deberían ser.

Cada patrón de diseño viene con unas ventajas e inconvenientes, y antes de implementarlos debemos analizar si realmente el patrón que queremos aplicar va a ser una ayuda, o va a causar complejidad innecesaria.

Personalmente, el conocer estos patrones ha abierto mi mente a un mundo nuevo de posibilidades a la hora de diseñar aplicaciones. Experimentar con ellos me ha hecho ver cómo crear aplicaciones más grandes sin que el código se vuelva un lío imposible de mantener y refactorizar. No sólo he experimentado con los patrones descritos en este

trabajo, sino con otros más, y planeo seguir haciendo pruebas con ellos. Es por ello que he decidido compartir el repositorio donde estoy implementando los diferentes patrones para que podáis beneficiaros de este trabajo, y si queréis, colaborar conmigo. Aquí está el enlace:

https://gitlab.com/screspos/design-patterns-ceinmark.git

Existen numerosos patrones además de los citados en este trabajo, y seguirán surgiendo patrones nuevos a medida que se sigue desarrollando software y siguen surgiendo nuevos problemas. Como diseñadores de software, debemos estar atentos a los problemas recurrentes que se dan a medida que trabajamos en nuestras bases de código, ya que podemos encontrar nuevas soluciones que se acaben convirtiendo en patrones de diseño establecidos en el futuro.

FUENTES:

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Larman, C. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.). Addison Wesley Professional.
- Refactoring Guru. (13-12-2023). Design Patterns. Recuperado de https://refactoring.guru/es/design-patterns
- OODesign. (13-12-2023). Object Oriented Design. Recuperado de https://www.oodesign.com/