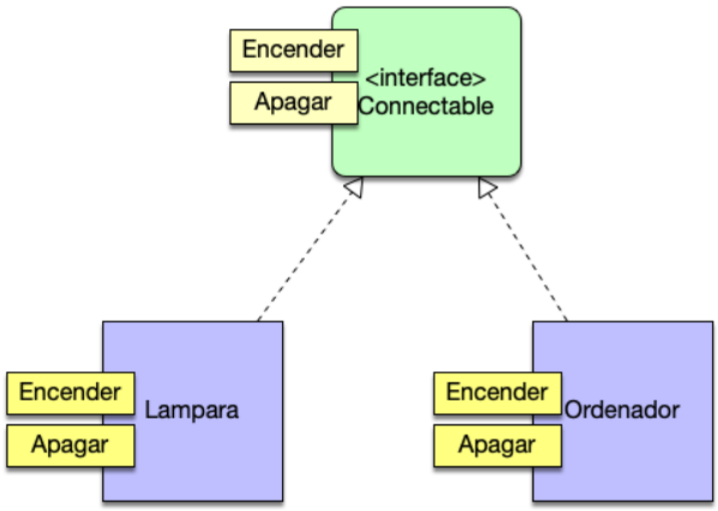
Unidad 10. Manejo Avanzado de Clases. Patrones Estructurales

Contenido

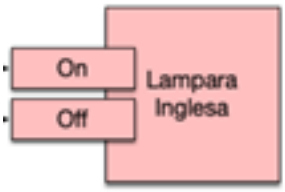
1 Actividad inicial.

Pregunta o problema del día para los alumnos

Dado el siguiente modelo de clases con un interfaz y dos tipos de electrodomésticos.



Si **añadimos un nuevo aparato, una nueva clase**, con **métodos diferentes para apagar y encender**. Como podemos hacer para que se **adapte a nuestro interfaz y funcionamiento sin modificar la clase.**



Se **les da diez minutos para intentar resolver el problema** de orientación a objetos.

2 Introducción. Actividad de exposición

En este **capítulo vamos continuamos explicando los diferentes patrones** o acercamientos que **usaremos para resolver nuestros problemas de programación en programación orientada a objetos**. En este caso **patrones relacionados con colecciones. Ampliamos el uso de funciones como parámetros** y el operador ::. Todo esto **nos llevará a ver el patrón Strategy en versión funcional.**

Recordamos **el capítulo anterior donde repasábamos los patrones de diseño** de orientación a objetos. **Usamos las estructuras y herramientas del tema anterior** como base para **mejorar nuestro diseño de clases.** Las **clases y los interfaces son unas estructuras de control** más en programación orientada

objetos. Nuestra labor es saberlas, conocerlas y utilizarlas apropiadamente.

Recordamos que tenemos cuatro tipos de patrones en orientación a objetos.

Patrones básicos: hacemos **manejo básico de encapsulación, herencia, polimorfismo y abstracción** para resolver problemas de orientación a objetos. Son la base o los cimientos sobre los que vamos a construir el resto de patrones de diseño.

Patrones de creación o creacionales: nos ayudarnos con la **creación de objetos de un framework** o conjunto de clases y subclases para hacerlo más transparente. En la mayoría de las ocasiones **enmascararemos los detalles internos** de esas clases. En otras sólo nos interesará usar una o dos operaciones comunes a todo ese tipo de clases.
Patrones estructurales nos **permiten resolver problemas de orientación a objetos creando nuevas estructuras** en nuestras **clases para resolver múltiples problemas de incompatibilidades entre subclases** que deben **colaborar** y mostrar un comportamiento común.
Patrones de comportamiento: Los patrones de comportamiento **tratan con los algoritmos** y como asignar de manera correcta las responsabilidades entre objetos y a los diferentes objetos. Son **fundamentales para que nuestras aplicaciones sean dinámicas**. Además, **nos van a permitir usar y aplicar** la **nueva programación funcional**.
Pero empezaremos introduciendo los principios, fundamentos de diseño orientado a objetos

3 Fundamentos de diseño y programación de modelos de clases

Empezamos con una serie de reglas básicas para la programación

3.1 Encapsular lo que varía

El objetivo principal de este principio es **minimizar el efecto causado** por cambios. Imagina que **tu programa es una nave,** y los **cambios son minas** que permanecen bajo el agua. Golpeado por la mina, se **hunde el barco**.
Sabiendo esto, puede **dividir el casco de la nave en compartimentos independientes** que **se pueden sellar de forma segura** para limitar el daño a un compartimiento único. Ahora, si **el barco golpea una mina**, la nave como un todo **permanece a flote**.
Del mismo modo, **puede aislar las partes del programa** que varían en módulos independientes, **protegiendo el resto del código de efectos adversos**. Como resultado, pasa **menos tiempo arreglando el diseño del programa**, la implementación y probando los cambios. Cuanto menos tiempo se pase haciendo cambios, más tiempo se tiene para implementar nuevas características.

3.2 Encapsulación a nivel de método

Supongamos que **estás haciendo un sitio web de comercio electrónico**. En tu código, puede **haber un método getTotalPedido** que **calcula el total del pedido**, incluidos los impuestos.

Podemos anticipar que el **código relacionado con los impuestos** podría **necesitar cambiar en el futuro**. La tasa impositiva depende del país, estado o incluso la ciudad donde reside el cliente, **y la fórmula real puede cambiar con el tiempo debido a nuevas leyes o regulaciones**. Como resultado, **tendrá que cambiar el método getTotalPedido** bastante a menudo. Pero incluso el nombre del método indica que no importe cómo se calcula el impuesto. El **ejemplo es orientativo y teórico**, sólo para explicar la teoría

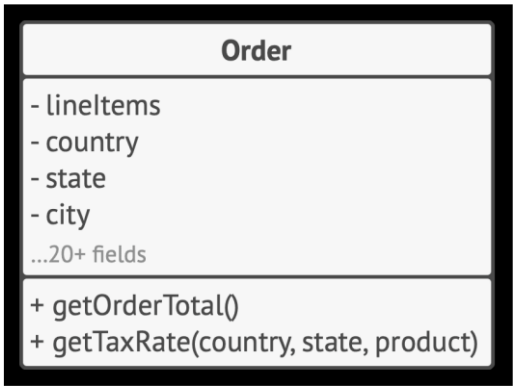
```
public double getTotalPedido(Producto ...pedidos) {
	double total = 0
		for(Producto Item:pedidos )  {
			total += item.precio * item.cantidad;

		if (pedidos.pais == "USA")
			total += total * 0.07; //
			else if (pedidos.pais == "UE")
			total += total * 0.20; //
		}
	 return total;
	}
```

Lo ideal en esta situación **es sacar los impuestos en un método aparte**:

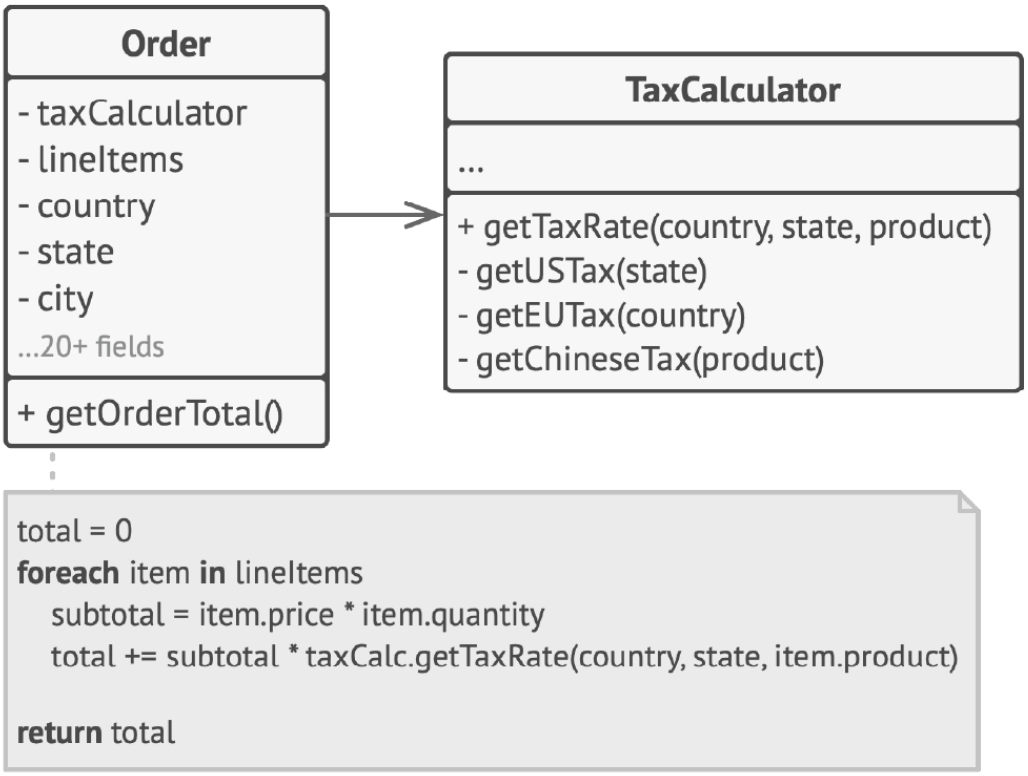
```
public double getTotalPedido(Producto ...pedidos) {
	double total = 0
		for(Producto Item:pedidos )
			total += item.precio * item.cantidad*getImpuestos(item);


			return total;
		}

	private double getImpuestos (Producto item) {

	double impuestos=0.0;

	if (item.pais == "USA")
		impuestos= 0.07 ;
		else if (item.pais == "UE"):
		impuestos=  0.20 ;

		return impuestos;
	}
```

3.3 Encapsulación a nivel de clase

Con el tiempo, **podrían agregarse más y más responsabilidades** (funciones) a un método que **solía hacer una cosa simple**. Estos comportamientos **añadidos a menudo vienen con sus propios atributos y métodos** que eventualmente **diluyen la responsabilidad y el significado principal de la clase**. **Extraer todo a una nueva** clase podría hacer el **código mucho más claro y simple**. Por ejemplo, en la siguiente clase se calculan los impuestos.



Si la **clase crece demasiado**, nos puede **interesar encapsular los impuestos** en otra clase.

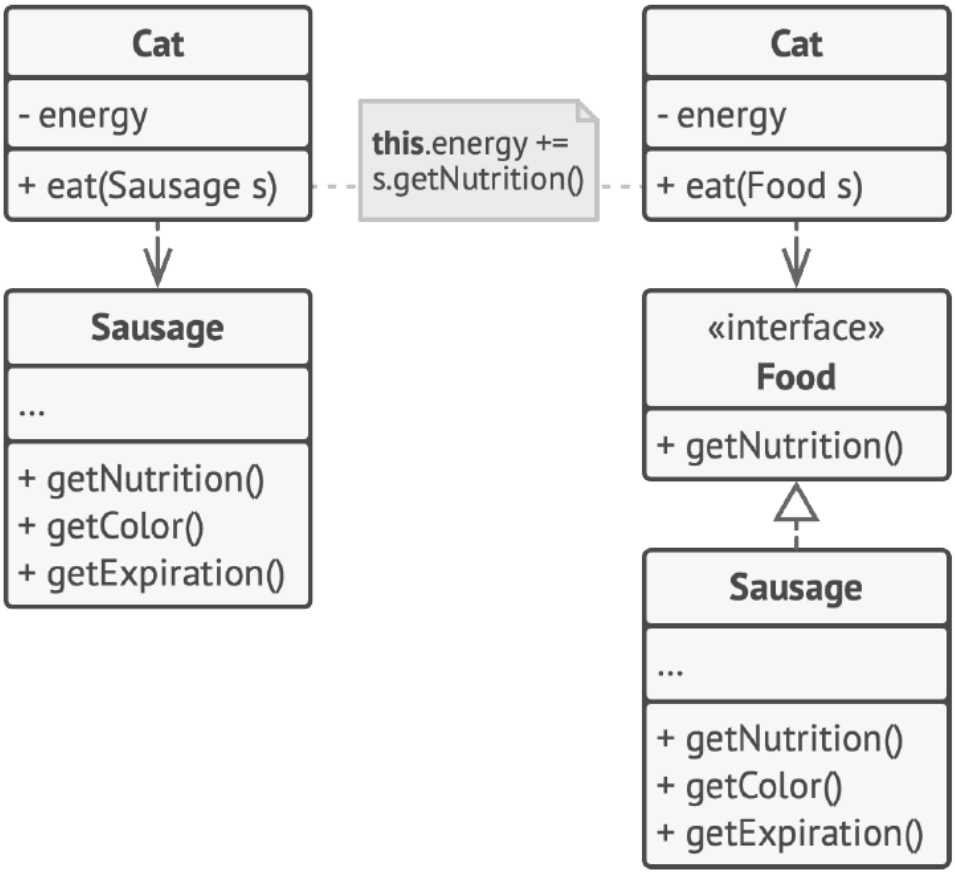


3.4 Programa usando interfaces no implementaciones en la variables.

Programa tu clase para un interfaz, no para una implementación, una **clase concreta**. Que **nuestras variables, atributos parámetros** dependan de abstracciones, no de clases concretas.

Se puede **decir que el diseño es lo suficientemente flexible** si se puede **ampliar sin dañar ningún código existente**. Vamos a asegurarnos de que esta afirmación es correcta mirando el ejemplo de la clase Cat. **Un Gato que puede comer cualquier alimento** es **más flexible que uno que pueden comer sólo salchichas.**
Cuando se **desea hacer que dos clases colaboren**, puede comenzar **haciendo que una de ellas dependa** de la otra. Sin embargo, **hay otro método**, más flexible para configurar la colaboración entre objetos:

1. **Determina qué necesita exactamente un objeto del** otro: ¿Qué métodos ejecuta?
2. Describir estos **métodos en una nueva interfaz** o clase abstracta.
3. Haz que **la clase principal implemente** esta interfaz.
4. Ahora **haced que la segunda clase que dependía** de la **anterior dependa de esta interfaz** en vez de en la clase concreta. Todavía va a funcionar con objetos de la clase original, pero la conexión es ahora mucho más flexible.



Después de hacer este cambio, no hay ventaja aparente. Al contrario, **el código se ha vuelto más complicado** de lo que era antes. Sin embargo, es un **buen punto de partida para añadir para alguna funcionalidad adicional**. Nos va a dar muchas ventajas como añadir más subclases a la interfaz, para que **pueden ser usadas por la clase cat**.

3.5 Favorecer la composición sobre la herencia

La **herencia es probablemente la forma más obvia** y **fácil de reutilización de código** entre clases. Si tenemos **dos clases con el mismo código**. Se crea **una clase base común** para estas dos y se mueve el **código similar en él**. Desafortunadamente, la herencia viene con requisitos que a menudo hacerse evidente sólo **después de que su programa ya tiene demasiadas clases** y cambiarlo es bastante difícil. Problemáticas del uso de herencia

1. Una subclase **no puede reducir la interfaz de la superclase**. tienen que implementar todos los métodos abstractos de la clase primaria incluso si no los usarás.

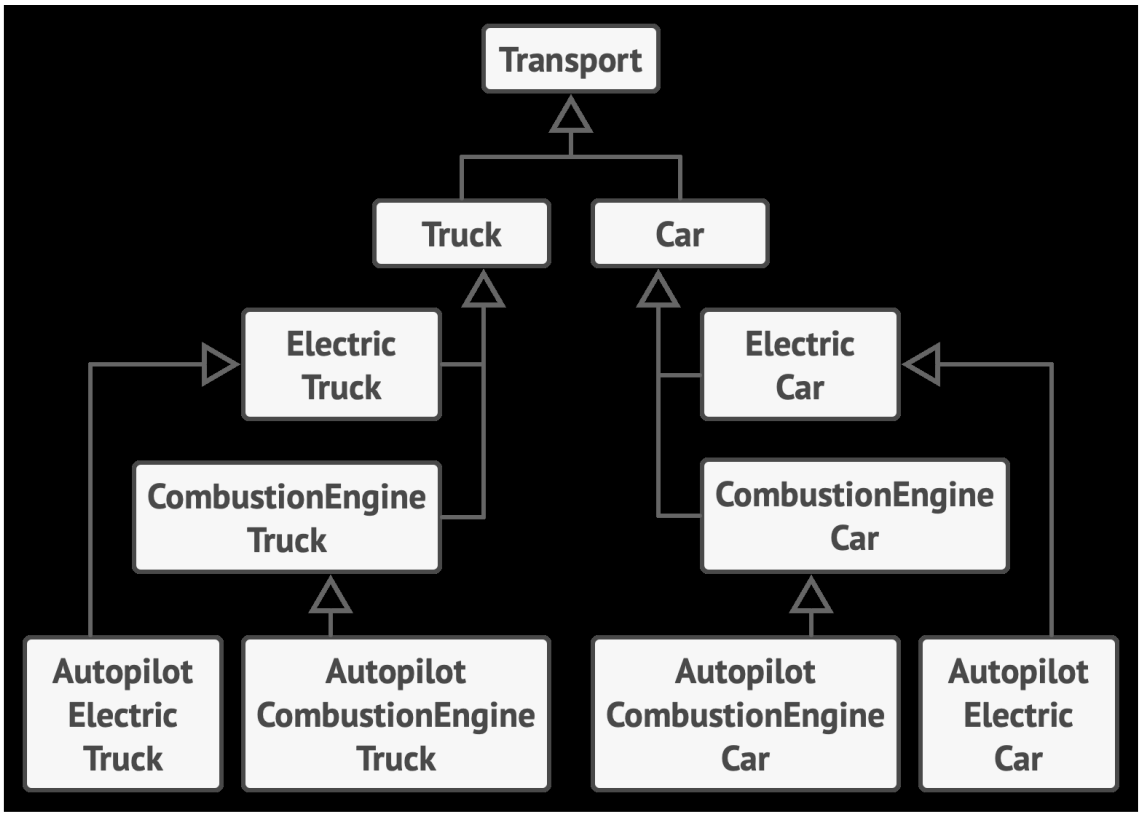
2. Al sobreescribir métodos, **debe asegurarse de que el nuevo comportamiento es compatible con el comportamiento base**. Es importante porque los objetos de la subclase se pueden pasar a cualquier código que espera objetos de la superclase y no quieres que código falle.

3. La **herencia rompe la encapsulación de la superclase** porque los detalles internos de la clase padre están disponibles para el Subclase. **Puede haber una situación opuesta** en la que un programador hace que una superclase conozca de algunos detalles de las subclases en aras de facilitar la herencia. Esto no es del todo correcto.

4. Las **subclases están estrechamente acopladas a las superclases**. Cualquier cambio **en una superclase puede** dañar la funcionalidad de las **subclases**.

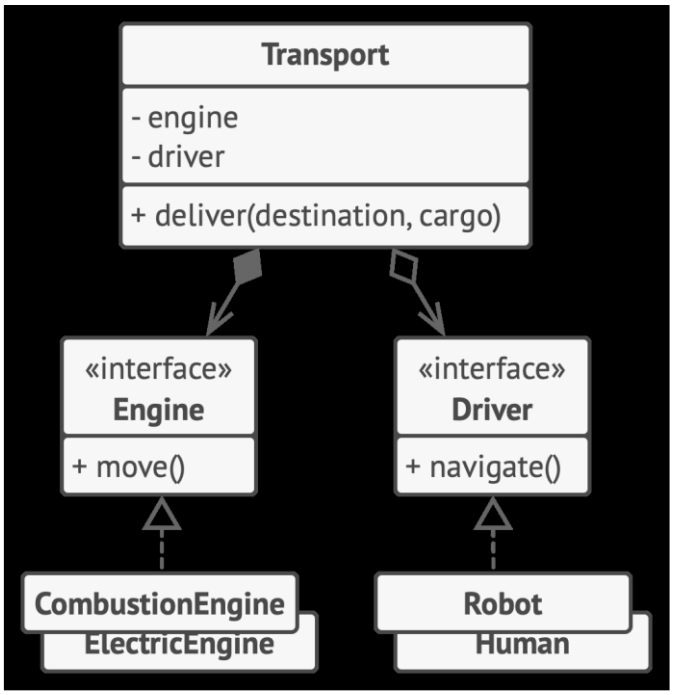
5. **Intentar reutilizar código a través de la herencia** puede llevar a crear jerarquías de **herencia paralelas**. La **herencia** suele lugar en **una sola dimensión**. Pero cada vez que hay dos o más dimensiones, hay que crear muchas combinaciones de clases, llevar la jerarquía de clases a un tamaño excesivo.

Hay una **alternativa a la herencia llamada composición**. Mientras que la herencia **representa la relación "es un"** entre (un coche es un transporte), la composición representa **"tiene un"** relación (un coche tiene un motor).
En el siguiente ejemplo tenemos un desmedido número de clases en el modelo.



Como se puede ver, **cada parámetro adicional da como resultado la multiplicación el número de subclases**. Hay un montón de código duplicado entre subclases porque una subclase no puede extender dos clases al mismo tiempo.
Puede **resolver este problema con la composición**. En **lugar de la clase coche que implementan un comportamiento por su cuenta**, **pueden delegar** a otros objetos.
La **ventaja añadida es que se puede reemplazar un comportamiento en tiempo de ejecució**n. Por ejemplo, **puedo reemplazar un objeto de motor vinculado** a un objeto de coche **simplemente asignando un objeto de motor diferente al coche**. Fijaos en la **siguiente figura componiendo Transporte**, con un engine (motor) y un driver (conductor) como **usando interfaces podemos añadir las subclases directamente sin crear clases abstractas intermedias**, tenemos un modelo de 5 clases donde antes teniamos once. Por eso tendemos a usar interfaces sobre clases.

En resumen, en vez de crear **muchas clases para tipos de transporte como en el modelo anterior**, creamos **una sóla clase Transport** y le **añadimos elementos**. A través del **interfaz engine le añadimos un motor eléctrico**, **electric engine**, y un a través del interfaz driver, le añadimos un autopilot. Tenemos un **medio de transporte eléctrico** y **con autopilot**, como en el modelo anterior, pero con muchas menos clases. **Hemos ahorrado 6 clases en nuestro modelo.**



3.6 Principios Sólidos del diseño orienta a objetos.

Principios sólidos viene del acrónimo en inglés **SOLID**, representando a **cinco principios de diseño** , que detallaremos en estos apuntes:

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

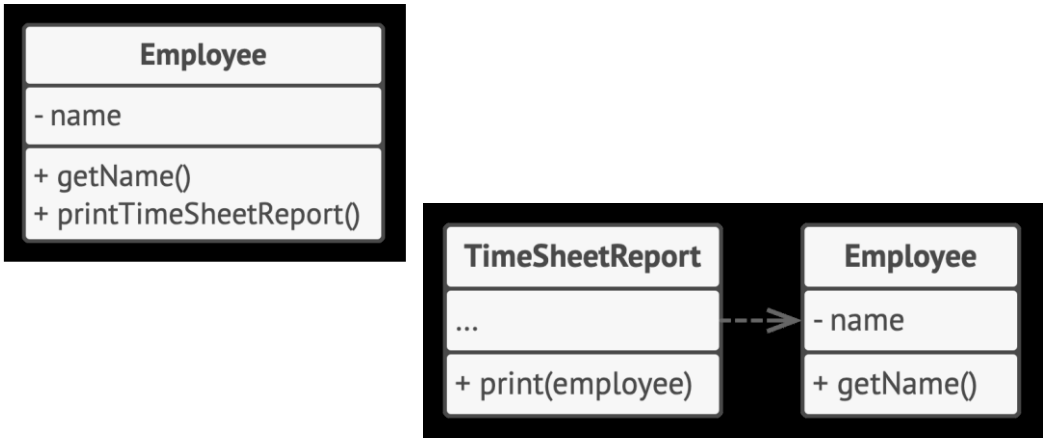
Empezamos por la **S,** el primero, e iremos explicando brevemente uno a uno.

3.6.1 Single Responsibility Principle

El **principio de responsabilidad única** **lo que intenta** es **hacer cada clase responsable de una única funcionalidad**, para separar o dividir bien las funcionalidades de nuestra aplicación. Se **hace todo el tiempo en interfaces funcionales y lambdas.**
El **objetivo principal de este principio es reducir la complejidad**. No es necesario crear **un diseño sofisticado para un programa** que sólo **tiene alrededor de 200 líneas de código**. Hacer **una docena de métodos limpios y correctos** es la **solución adecuada.**

Los **problemas reales surgen cuando el programa constantemente crece y cambia**, las **clases se vuelven tan grandes que ya no se puede recordar** sus detalles. Navegar por **el de código es más difícil y tienes que buscar a través de todo clases** o incluso todo un programa para encontrar detalles específicos.

Hay más: **si una clase hace demasiadas cosas, tienes que cambiar la clase cada** vez que **una de estas funcionalidades cambia**, poniendo **en riesgo código de la clase** que **ni atienden a esa funcionalidad.**
En la clase Employee (Empleado) hay **varias razones para introducir cambios.**
La **primera razón podría estar relacionada** con la función principal de la clase: **la gestión de datos de los empleados.** Sin embargo, hay otra razón**: el formato del informe de horas puede cambiar con el tiempo**, lo que r**equiere que cambiar el código dentro de la clase**. Una **funcionalidad como está que no es pura del empleado**, es un informe se **debe llevar a otra clase Report** (informe).



3.6.2 Open/Closed Principle

La **idea principal de este principio es evitar que el código** existente deje de compilar al implementar nuevas características.

Una **clase es abierta si se puede heredar de ella o extender**, **producir una subclase** y hacer lo que quieras con ella, añadir nuevos métodos o campos,

invalidar el comportamiento base, etc. **Algunos lenguajes de programación permiten restringir más herencias de una clase** con palabras **clave especiales, como final**. Después de asignar un final a la clase, **la clase ya no es abierta**. Al mismo tiempo, una clase **está o es cerrada** (también se puede decir completa) si **está 100% preparada para ser utilizada** por otras clases: su **interfaz ha sido definida claramente y no se cambiará en el futuro.**

Una clase es cerrada si su interfaz está perfectamente definido y no va a ser cambiado con el tiempo. Es muy importante que las superclases sean cerradas, porque un cambio en una superclase puede afectar a muchas subclases.

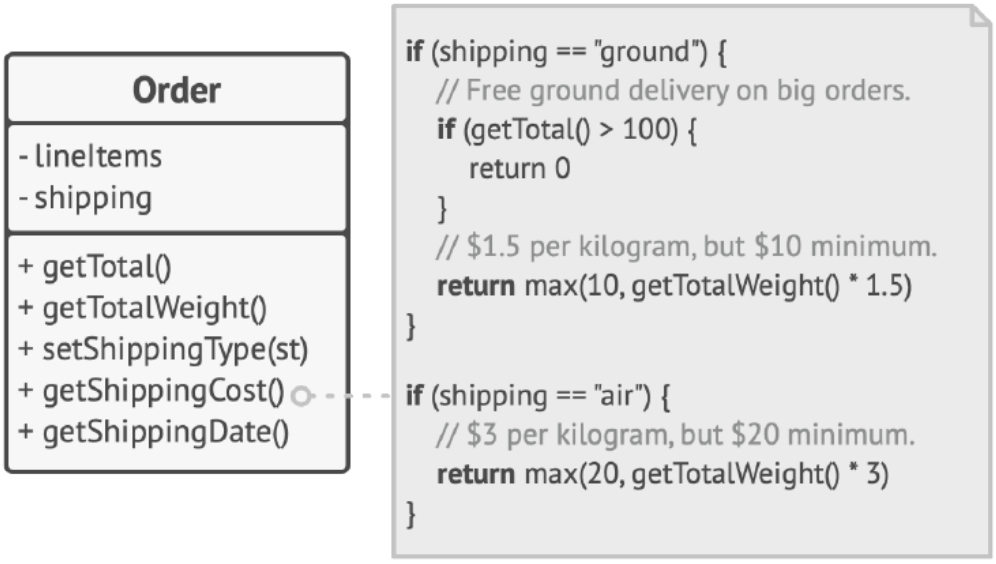
Cuando **se empieza a trabajar con este principio**, **es confuso porque las palabras abiertas y cerradas suenan mutuamente excluyentes**. Pero en términos de este principio, **una clase puede ser abierta (para extensión**) y **cerradas (para su modificación)** al **mismo tiempo**.

Si una clase ya **está desarrollada, probada, revisada e incluida en algún framework** o **utilizada de otra manera en una aplicación**, tratar de **desordenar su código es arriesgado**. En **lugar de cambiar el código de la clase directamente**, se **debe de crear subclases y reemplazar las partes de la clase original** de las que desea modificar su comportamiento. De **esta manera consigues tu objetivo sin modificar la clase original** y hacer que **todo el programa siga funcionando y compilando**.

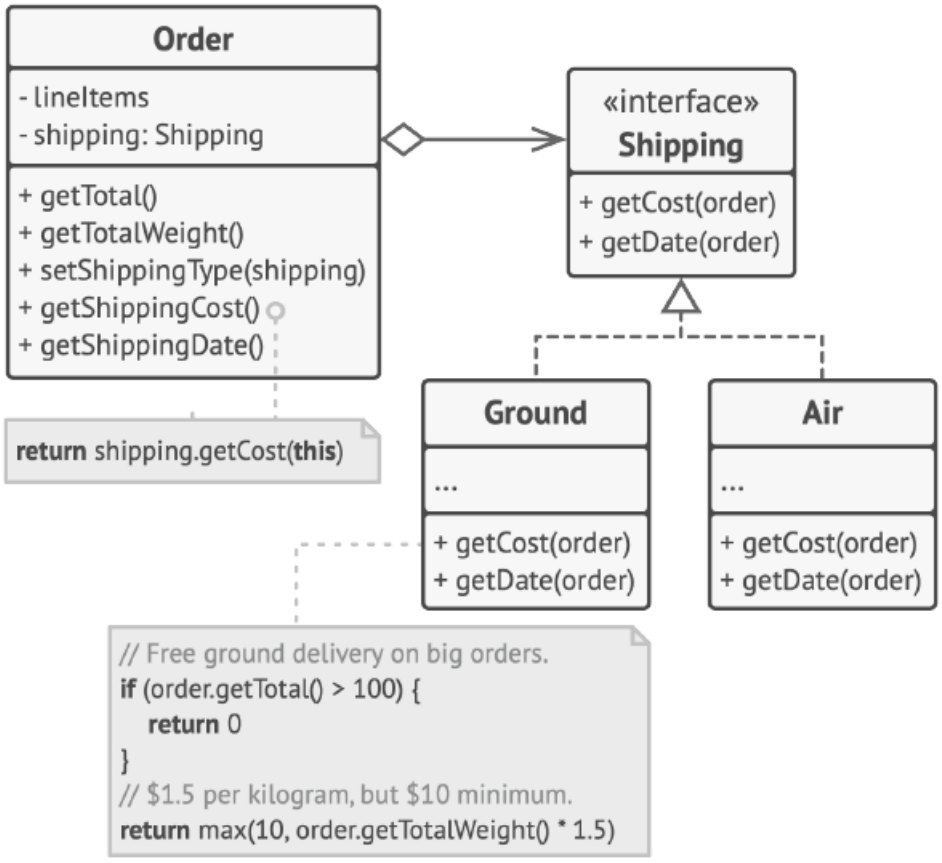
Este principio no está destinado a aplicarse a todos los cambios en una clase. Si sabes que **hay un error en la clase, hay que arreglarlo**, no crear una subclase para él.

Ejemplo

Si tienes una **aplicación de comercio electrónico con una clase Order** que **calcula los costos de envío y todos los métodos de envío** son **codificados dentro de la clase**. Si **necesita agregar un nuevo método de envió**, tienes que **cambiar el código de la clase** Order **a riesgo de extropear tu código.**



Aplicando **el patrón Strategy aquí podemos arreglar el problema extrayendo el método común getShippingCost();** para **implementar la funcionalidad del envió en clases separadas**, con un interfaz común



Ahora, **cuando necesita implementar un nuevo método de envío, puede derivar a una nueva clase que implemente el interfaz shipping** sin tocar el

código de la clase Order.
Si el atributo shipping toma como tipo el interfaz Shipping, **en tiempo de ejecución asignaremos a ese atributo un objeto Ground o Air** en **función de la elección del usuario para hacer el pedido.**

Nota: Es una de las prácticas más habituales en orientación a objetos.

3.6.3 Liskov Substitution Principle

El principio de substitución de Liskov nos indica que básicamente a un método o **clase cualquiera podemos pasar** indistintamente **un objeto de una clase o de sus clases hijas sin** que **el código deje de funcionar.**

Esto **significa que la subclase debe seguir siendo compatible con el comportamiento de la superclase. Al sobreescribir un método, debemos ampliar el comportamiento de la clase base** en **lugar de reemplazarlo con algo totalmente distinto.**

El principio de sustitución es un conjunto de comprobaciones que **ayudan a predecir si una subclase sigue siendo compatible** con **el código que fue capaz de trabajar con objetos de la superclase**. Este **concepto es fundamental a la hora de desarrollar bibliotecas y frameworks** porque **sus clases van a ser utilizados por otros programadores** cuyo código no se puede acceder y cambiar directamente.

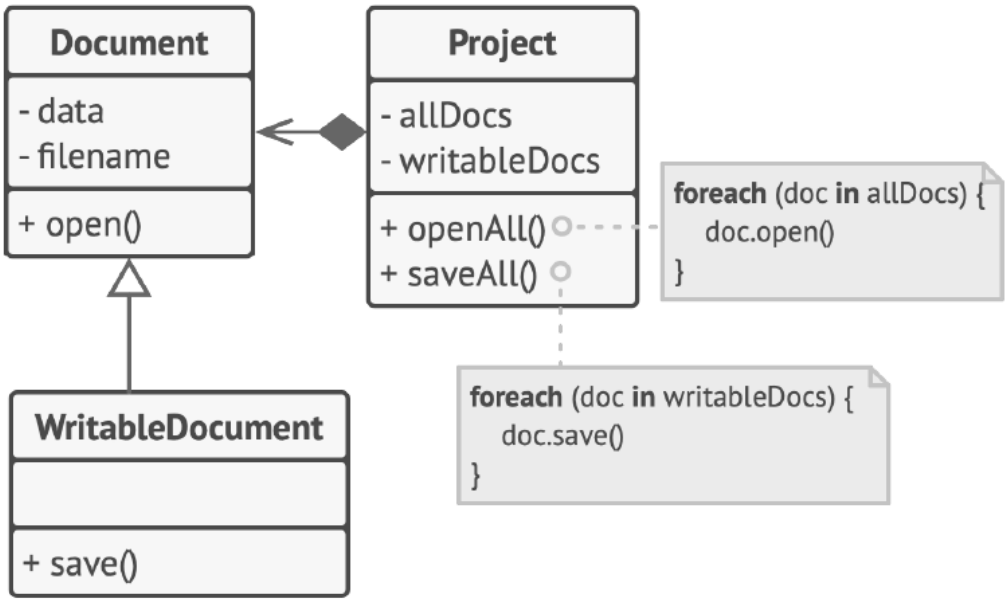
Los **tipos de parámetros y los tipos retornados de un método de una subclase** deben **coincidir o ser más abstracto que los tipos de parámetros en el método de la Superclase**. Vamos a tener un ejemplo.
Supongamos que hay una clase con un **método que se supone que alimenta gatos: feed (Cat c)** . El código de cliente siempre **pasa objetos de tipo Cat (gato) en este método**.

Habitualmente **cuando tenemos clases que representa animales**, tendremos **una clase abstracta Animal**. Lo **correcto en este caso sería** que el método **feed(Animal a)** reciba la **clase abstracta Animal**. Así **el método nos sirve para cualquier tipo de animales**.

Lo **incorrecto sería tener una subclase de Cat**, **WildCat**, y **sobreescribir el método feed(WildCat wa)**. Esto **restringe el uso de la subclase WildCat** e invalida poder **castear WildCat como un Cat genérico** si nos hace falta.
Para el **tipo de métodos correcto el funcionamiento** es el mismo **public Animal BuyAnimal()**, se debe implementar igual en la clase Cat y en la clase WildCat. **public Cat BuyAnimal().** Para la **clase WildCat el tipo retornado del método** debe ser el mismo, **public Animal BuyAnimal().**

En **el siguiente ejemplo podéis ver un diseño correcto**, tanto Document como WritableDocument, en el método open, **deben devolver el tipo Doc**, para **poder ser manejado por la clase Project**. En **allDocs tenemos todos los**

documentos, en WritebleDocs, **solo los documentos que se pueden editar**. En **allDocs abrimos con open todos los documentos**, writables o no. Por eso el open de WritableDocument debe devolver un tipo Doc, no writableDoc.

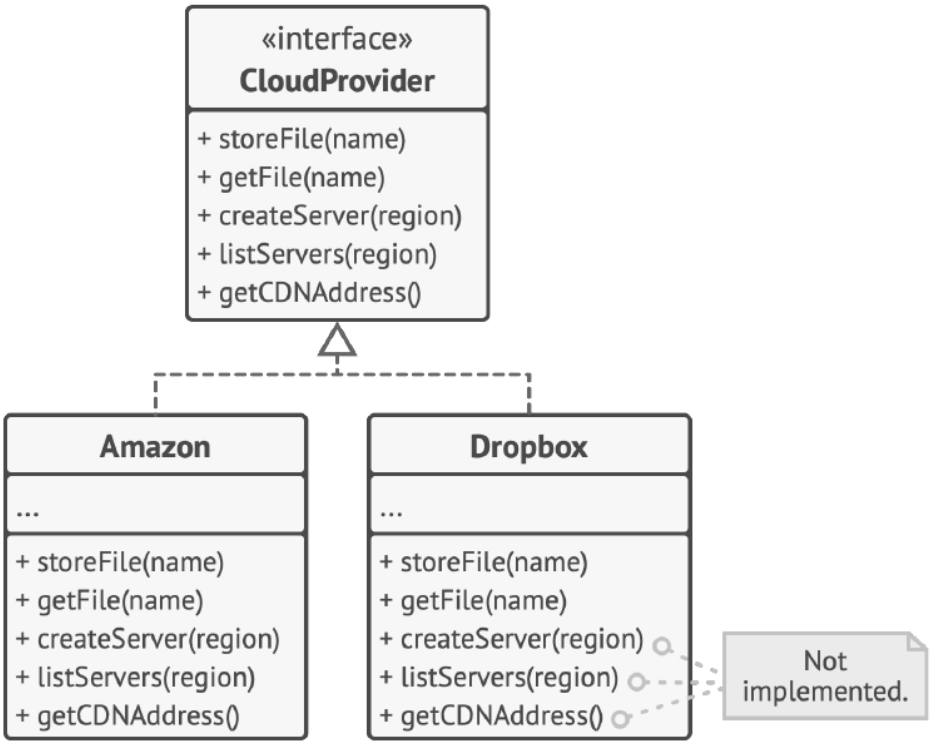


3.6.4 Interface Segregation Principle

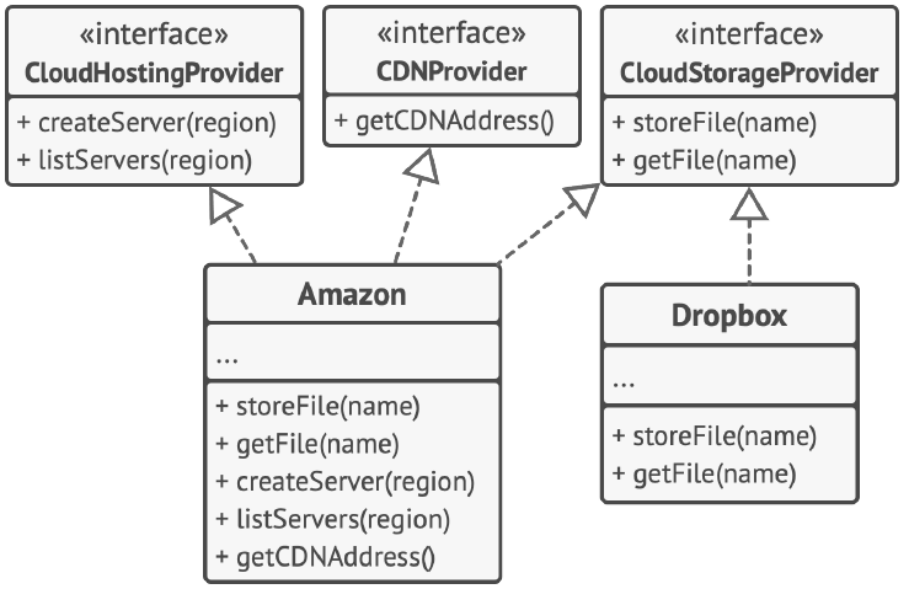
Los clientes no deben ser forzados a depender de los métodos que no usan y necesitan.

Trata de hacer que sus interfaces sean lo suficientemente específicos como para que las **clases de cliente no tienen que implementar comportamientos que no necesitan.**

De acuerdo con este principio, **los interfaces demasiado amplios deberían descomponerse en mas específicos**. Esto **es debido a que las clases clientes deben implementar sólo los métodos necesarios**, no los que no necesitan. En caso contrario, si **tienes una interface demasiado compleja**, un **cambio en este va a afectar a clases que no usan esa funcionalidad**. En resumen, no se **deben añadir métodos a un interfaz que no estén relacionados**, es **mejor dividir el interface complejo** en unos más específicos.
Por ejemplo, imaginad **que habeis creado una librería o API para integrar varios proveedores en la nube**. Mientras que **en la versión inicial sólo se soportaba Amazon Cloud**, que **nos cubría todos los servicios y características de la** nube. En el momento de **programar tu Api asumiste que el resto de proveedores** ofrecía los **mismos servicios que Amazon**. Pero **cuando ha tocado añadir nuevos proveedores te das cuenta de que no es así**, y que **el interfaz que definiste para Amazon** no es valido para el resto. **Algunos de los métodos de tu interfaz** ofrecen características que **no usan otros proveedores de la nube.**



Fijaos en **la figura en Dropbox no tenemos servidores, ni CDNAddress**. La solución es **dividir nuestro interfaz en tres interfaces**. Es muy habitual este paso en programación orientada a objetos. Uno **para los proveedores de servicios** que **ofrecen un servidor**. **Otro para los que ofrecen dirección CDN** y **otro para los que sólo ofrecen archivos y directorios** como Dropbox.



Al **igual que con los otros principios, se puede ir demasiado lejos con este**. **No dividas aún más un interfaz** que ya es **suficientemente específica**. Recuerde que cuantas más interfaces se crean, más complejo se convierte el

código. Mantén el equilibrio.

3.6.5 Dependency Inversion Principle

Las clases de alto nivel no deben depender de clases de bajo nivel. Ambos deben depender de abstracciones. **Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.**

Por lo general, al diseñar software, puede hacer una distinción entre dos niveles de clases.
• Las **clases de bajo nivel implementan operaciones básicas** como trabajar con un disco, transferir datos a través de una red, conectárse a un base de datos, etc.
• Las **clases de alto nivel contienen una lógica de negocios compleja** que clases de bajo nivel para hacer algo.

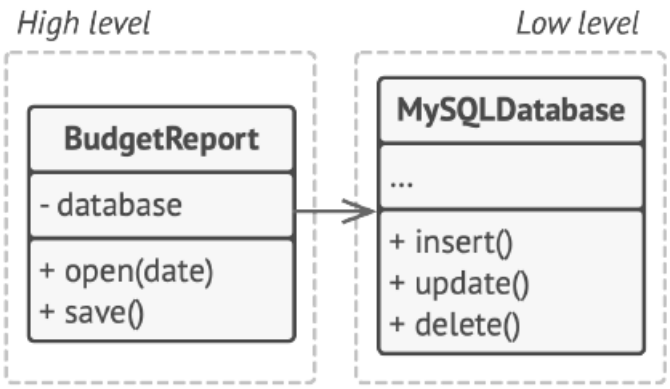
1. Para **las clases cabecera, deben describirse interfaces** para **operaciones de bajo nivel de las que las clases de alto nivel dependen**, preferiblemente con **terminología del Negocio que estamos informatizando.** Por ejemplo, la lógica **de negocios debe llamar a un método openReport(file)** en lugar de una serie de métodos openFile(x) , readBytes(n) , closeFile(x) . Estas **interfaces se cuentan como interfaces de alto nivel**.

2. Ahora **puedo hacer que las clases de alto nivel dependan de interfaces**, en lugar **de en clases concretas de bajo nivel**. Éste **dependencia será mucho más ligera que la original**. Esto **hace que tu código si sufre modificaciones** en **las clases concretas** no tengas que modificar las clases de alto nivel.

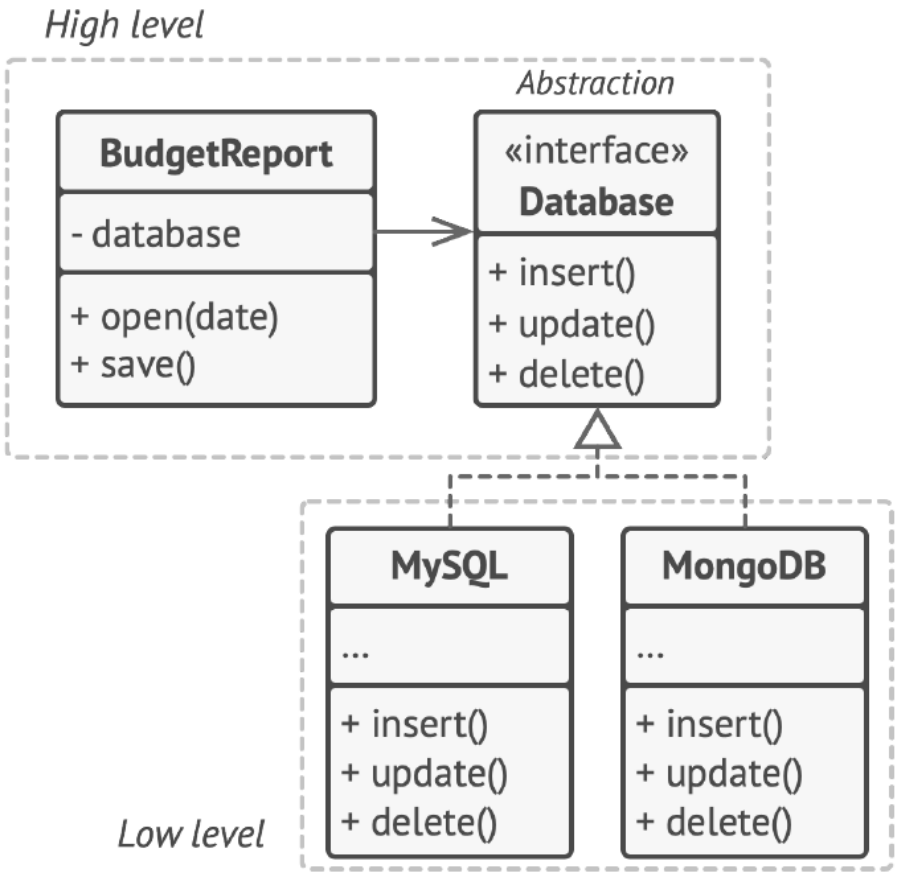
3. Una **vez que las clases de bajo nivel implementan estos interfaces**, **dependen del nivel de lógica de negocio**, **invirtiendo la dirección** de la dependencia original.

El **principio de inversión de dependencia** a menudo **va de la mano del open/closed principle**: se puede **ampliar las clases de bajo nivel para utilizar con diferentes clases de lógica de negocios** sin dañar o modificar las clases existentes.
Por ejemplo:
En este ejemplo, **la clase BudgetReport de alto nivel utiliza un clase** de base de datos **MySQLDatabase** de bajo nivel **para leer y conservar sus datos**. Esto significa que **cualquier cambio en la clase de bajo nivel**, como cuando se publique una nueva versión del **servidor de bases de datos, puede afectar a la clase de alto nivel**, que **no se supone que se preocupe por los detalles** de almacenamiento de datos.
Nota: Es **uno de los mayores errores que se puede cometer** en **programación**

orientada a objetos. En ocasiones **no queda más remedio que saltarse alguno de estos principios** debido al **framework o API** que estas usando.



Se **puede solucionar este problema creando una interfaz de alto nivel** que describe **las operaciones de lectura/escritura** y la **realización de los informes de la clase de bajo nivel**. y **utilizar esa interfaz en lugar de la clase de bajo nivel**. A continuación, se **puede cambiar o ampliar la clase de bajo nivel** original a **implementar la nueva interfaz de lectura/escritura** declarada por la lógica de la empresa.



3.7 Practica guiada Principios sólidos de diseño

Igualmente se espera que los interfaces tengan las responsabilidades muy definidas y realicen una labor como por ejemplo el siguiente interfaz que implementa la clase trabajador su función es muy clara calculo de sueldo de trabajadores. **Interfaz segregation Principle** y **Single Responsability Principle**

SueldoTrabajadores.java

```
package patronescreacionales.interfacesvariasclases;

interface  SueldoTrabajadores {


	double calculaSueldo();
	double calculaImpuestos();



}
```

Lo primero a la hora de definir un modelo de clases **es definir clases abiertas para ser heredadas**, **pero cerradas con responsabilidades muy claras y definidas de antemano** para que no las tengamos que modificarlas y a todas sus subclases. Open/Close principle. La siguiente clase Trabajador tiene todas esas características. Además sólo realiza una función que es manejar trabajadores **Single Responsability Principle**.

Trabajador.java

```
package patronescreacionales.interfacesvariasclases;

public abstract class  Trabajador implements SueldoTrabajadores {
	protected int id;
	protected String nombre;
	protected double sueldo;


	public abstract String funcionTrabajador();
```

```
	public Trabajador() {


	}



	public Trabajador(int id, String nombre, double sueldo) {

		this.id=id;
		this.nombre = nombre;
		this.sueldo = sueldo;
	}



	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public double getSueldo() {
		return sueldo;
	}

	public void setSueldo(double sueldo) {
		this.sueldo = sueldo;
	}



	@Override
	public String toString() {
		return "Trabajador [id=" + id + ", nombre=" + nombre + ",
sueldo=" + sueldo + "]";
	}

	@Override
	public int hashCode() {

		return id;
	}

	@Override
	public boolean equals(Object obj) {
```

```
			return this.id==((Trabajador) obj).getId();

		}

		@Override
		public double calculaSueldo() {
			// TODO Auto-generated method stub


			return sueldo - calculaImpuestos();
		}

		@Override
		public double calculaImpuestos() {
			// TODO Auto-generated method stub
			return sueldo*0.10;
		}



}
```

La clase profesor y conserje son clases abierta y cerrada igualmente. Podríamos seguir heredando y especializando.

```
package patronescreacionales.interfacesvariasclases;

public class Profesor extends Trabajador implements SueldoTrabajadores {


		private int horasLectivas=0;


		public Profesor() {


		}

		public Profesor(int id, String nombre, double sueldo, int
horasLectivas) {
			super(id,nombre,sueldo);

			this.horasLectivas=horasLectivas;

		}
```

```
	public int getHorasLectivas() {
		return horasLectivas;
	}

	public void setHorasLectivas(int horasLectivas) {
		this.horasLectivas = horasLectivas;
	}

	@Override
	public String toString() {
		return "Profesor [id=" + id + ", nombre=" + nombre + ", sueldo="
+ sueldo + "]";
	}


	@Override
	public double calculaSueldo() {
		// TODO Auto-generated method stub


		return sueldo + 200 - calculaImpuestos();
	}

	@Override
	public double calculaImpuestos() {
		// TODO Auto-generated method stub
		return sueldo*0.20;
	}

	@Override
	public String funcionTrabajador() {
		// TODO Auto-generated method stub
		return "Enseñar";
	}









}
```

Conserje.java

```
package patronescreacionales.interfacesvariasclases;

public class Conserje extends Trabajador implements SueldoTrabajadores {

	private int numHorasDia=0;


	public Conserje() {
```

```
	}

	public Conserje(int id, String nombre, double sueldo,int numHorasDia)
{
		super(id,nombre,sueldo);

		this.numHorasDia=numHorasDia;

	}



	public int getNumHorasDia() {
		return numHorasDia;
	}

	public void setNumHorasDia(int numHorasDia) {
		this.numHorasDia = numHorasDia;
	}

	@Override
	public String toString() {
		return "Conserje [id=" + id + ", nombre=" + nombre + ", sueldo="
+ sueldo + "]";
	}

	@Override
	public String funcionTrabajador() {
		// TODO Auto-generated method stub
		return "Atended conserjeria";
	}

}
```

La clase ProfesorTecnico especializa brevemente a profesor modificando sólo un comportamiento, el sueldo.

```
package patronescreacionales.interfacesvariasclases;

public class ProfesorTecnico extends Profesor {


	@Override
	public double calculaSueldo() {
		// TODO Auto-generated method stub


		return sueldo + 150 - calculaImpuestos();
	}

}
```

Veamos nuestro programa principal. Nuestro programa principal cumple los otros dos principios sólidos de diseño. Liskov Substitution Principle nos indica que podemos movernos de clase a subclase y el programa debe seguir

funcionando.

```
		Trabajador profTrab=(Trabajador) prof;

		Profesor profOb= (Profesor )prof;
```

Igualmente cumple el Dependency Inversión Principle, que **las clases de alto nivel en este caso el programa principal no deben depender de clases de bajo nivel**. Ambos deben depender de abstracciones. **Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones. Como veis las clases de bajo nivel Conserje y Profesor son las que implementan el interfaz SueldoTrabajadores**

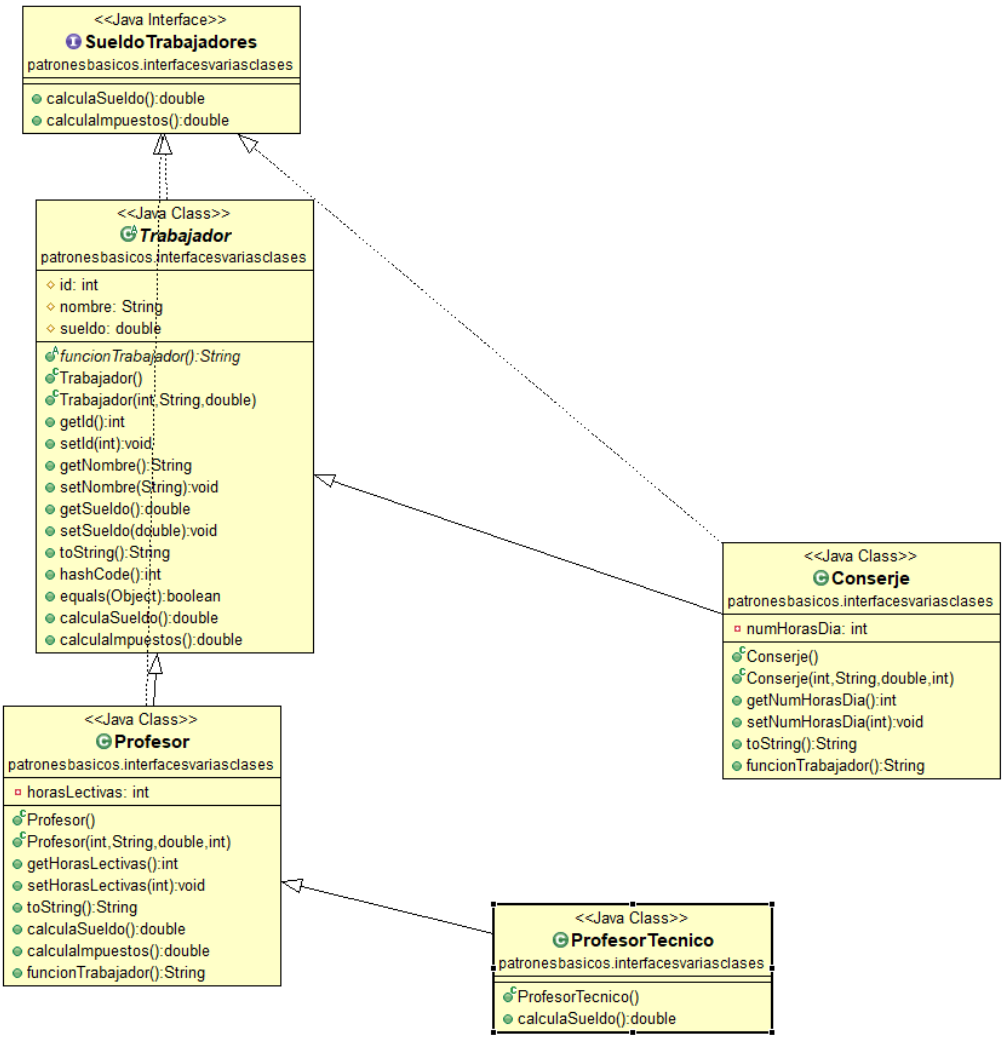
Nuestro programa está calculando impuestos, en el ejemplo para calcular impuestos sólo necesitamos variables de tipo interfaz, no especializaciones como veis marcado en verde. Programamos para las abstracciones, genérico, no para las implementaciones.

```
SueldoTrabajadores trab = new Conserje(1, "Mateo",15000, 7);

SueldoTrabajadores prof = new Profesor(1, "Jesus",30000,20);


		double totalImpuestos = trab.calculaImpuestos() +
prof.calculaImpuestos();

		double totalSueldos = trab.calculaSueldo() +
prof.calculaSueldo();
```

```
package patronescreacionales.interfacesvariasclases;

public class Main {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		SueldoTrabajadores trab = new Conserje(1, "Mateo",15000, 7);

		SueldoTrabajadores prof = new Profesor(1, "Jesus",30000,20);

		Trabajador profTrab=(Trabajador) prof;

		Profesor profOb= (Profesor )prof;

		double totalImpuestos = trab.calculaImpuestos() +
prof.calculaImpuestos();

		double totalSueldos = trab.calculaSueldo() +
prof.calculaSueldo();


		System.out.println("Total  a pagar de impuestos: " +
totalImpuestos);
```

```
            System.out.println("Total a pagar de sueldos sin contar
impuestos: " +(totalSueldos-totalImpuestos));

            System.out.println("Total a pagar: " + totalSueldos);


            System.out.println("Horas lectivas profesor: " +((Profesor)
prof).getHorasLectivas());

            //Horas del conserje
            System.out.println("Horas diarias conserje: " +((Conserje)
trab).getNumHorasDia());


      }

}
```

El modelo de clases sería el siguiente siguiendo todos los principios sólidos de diseño.



3.7.1 Notas Cornell:

1. Para el **ejemplo anterior, para cada línea de código señalada con el marcador, comenta su funcionamiento**. Apóyate en los apuntes para realizar los comentarios.
2. **Describe brevemente la relación existente entre clases e interfaces** en el modelo de clases anterior.

4 Patrones estructurales

Tratar con **objetos que delegan responsabilidades** a otros objetos. Esto da como resultado en **una arquitectura en capas de componentes** con bajo grado de acoplamiento. Facilita **la comunicación entre objetos** cuando un **objeto no es accesible** para el otro por medios normales o cuando un **objeto no es utilizable debido a su interfaz incompatible**. Proporciona **formas de estructurar un objeto agregado para que se cree en su totalidad y recuperar los recursos del sistema** de manera oportuna.

Visto de otra manera, **los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes**, **manteniendo estas estructuras flexibles y eficientes**. Cada **patrón describe un problema que se produce en orientación a objetos** una y **otra vez en nuestro entorno**, y luego **describe el núcleo de la solución a ese problem**a, de tal **manera que puede usar esta solución un millón de veces más**, sin hacerlo de la misma manera dos veces.

Tenemos diez patrones estructurales:
Adapter: Permite que **los objetos con interfaces incompatibles colaboren**. Interfaz como ya sabeis, es el **conjunto de métodos que ofrece ese objeto para ser utilizado**.
Bridge: Permite **dividir una clase grande o un conjunto de clases estrechamente relacionadas** en **dos jerarquías separadas —abstracción e implementación**— que se pueden **desarrollar independientemente una de la otra**.
Composite: es un **patrón de diseño estructural que te permite componer objetos en estructuras de árboles** y luego **trabajar con estas estructuras como si fueran objetos individuales**.
Decorator: es **un patrón de diseño estructural que te permite conectar nuevos comportamientos** a **los objetos colocando estos objetos dentro objetos contenedor** especiales que **contienen dichos comportamientos.**
Low Coupling: Es un patron que **indica como asignar responsabilidades** entre clases de manera que **disminuyamos las dependencias entre ellas al mínimo**. De esta manera **un cambio en una clase afecte lo mínimo al resto**. No lo vamos a implementar pues mas o menos lo aplicamos en otros patrones.

Flyweight: es un **patrón de diseño estructural que te permite encajar más objetos en la cantidad disponible de memoria RAM** compartiendo partes de estado entre varios objetos en lugar de mantener todos los de los datos de cada objeto. Con **nuestras memorias RAM actuales está en desuso.**

Façade: es un **patrón de diseño estructural que proporciona un patrón simplificado de interfaz a una biblioteca**, un **framework** o cualquier otro **conjunto complejo de clases**. Adapter intenta adaptar tu interfaz actual. **Facade añade uno nuevo**. Es una clase que nos da acceso y uso a una **biblioteca de objetos entera.**
Proxy: es un **patrón de diseño estructural** que le permite **proporcionar una sustituto o referencia para otro objeto**. Un **proxy controla el acceso al objeto original**, lo que le **permite realizar algo antes o después de que la solicitud llegue a el objeto original**.
Veremos algunos de los más importantes a lo largo de este tema. **Los primeros cuatro el ABCD** de los **patrones estructurales, plantearemos tres con ejemplos, el cuarto, Composite**, lo dejaremos **para temas posteriores**. A estos patrones se lo conoce como **The gang of Four**.
Vamos a introducir conceptos relacionados con patrones
Intent, intención. En patrones, se **define el intent como el objetivo** o el problema para el que **están diseñados**.
Problema: cual es **el problema de programación que se nos plantea** y como resolverlo.
Solución: es **la solución orientada a objetos** que **se adopta** para el problema.

4.1 ABCD (Adapter, Bridge, Composite, Decorator)

4.1.1 Adapter Pattern. Practica guiada

Intent

El patrón **de adaptador es un patrón de diseño estructural** que permite que los objetos **con interfaces incompatibles** trabajen juntos.

Problema

Tenemos que introducir en nuestro programa una clase nueva que no se adapta a una interfaz común que nos ofrece la mayoría de nuestros objetos de nuestro sistema. Ofrece unos métodos diferentes a los habituales al cliente.

Solución

Implementa **una interfaz conocida por sus objetos clientes y proporciona acceso a una instancia de una clase no conocida** por sus objetos clientes.
Interfaz **quiere decir método o métodos de acceso** al objeto
AdapterClient: el código o clase cliente.
Adapter: la clase Adapter que reenvía las llamadas al adaptee.
Adaptee: el código antiguo que necesita ser adaptado.
Target: la nueva interfaz a admitir.

Ejemplo en el mundo real

Casos de uso

1. Cuando **se quiere utilizar una clase existente, y su interfaz no es válido** para la interfaz que el cliente necesita necesita.
2. Cuando **desea crear una clase reutilizable que coopera con clases no relacionadas** que son clases que no tienen necesariamente interfaces incompatibles.
3. Donde **debe producirse la traducción de la interfaz** entre **varias fuentes**.

Vamos a ver un ejemplo práctico para que entendáis mejor este patrón.

Como vemos **nuestro sistema gira en torno a los motores**, **utilizamos la herencia para compartir funcionalidades comunes para los diferentes tipos de motores** con los que trabajaremos. Sin embargo ,evidenciamos que no todos ellos se comportan de la misma manera como es el caso del **Motor Eléctrico**,

por tal razón **no podemos hacer que herede directamente** de la clase **Motor**, ya que **los métodos que esta nos provee** no **serian útiles para esta clase**....

En este **punto es donde hacemos uso** de una clase **Adapter** que serviría de puente entre **la clase padre y la Clase que debe ser adaptada**, así **este adaptador seria el encargado de establecer comunicación con el motor Eléctrico** y **ejecutar las solicitudes** que el **cliente** realice...

Clase Motor.

Esta clase **es desde la cual heredaran los diferentes tipos de motores**, provee los métodos comunes (encender, acelerar, apagar) para el funcionamiento de los mismos.

```
public abstract class Motor {
   abstract public void encender();
   abstract public void acelerar();
   abstract public void apagar();
}
```

Motores Común.

Esta clase representa la estructura de los motores normales con los que el sistema que esta provee.

```
public class MotorComun extends Motor {


    public MotorComun(){
        super();
        System.out.println("Creando el motor comun");
    }


    @Override
    public void encender() {
        System.out.println("encendiendo motor comun");
    }


    @Override
    public void acelerar() {
        System.out.println("acelerando el motor comun");
    }
```

```
    @Override
    public void apagar() {
        System.out.println("Apagando motor comun");
    }
}
```

Motor eléctrico.

Esta es la clase que nos da problemas porque tiene un interfaz diferente al resto , **funciona diferente al resto**. Como consecuencia **tenemos que buscar una manera de integrarla en el sistema** y que **funcione como el resto**. Para ello vamos a usar la clase **siguiente el MotorElectricoAdapter que se encargará de adaptar el funcionamiento a nuestro sistema**. Los **motores normales, se encienden, apagan y aceleran**. El **motor eléctrico, se conecta se activa y se mueve más rápido**. Tiene un **comportamiento diferente.**

```
public class MotorElectrico {

    private boolean conectado = false;

    public MotorElectrico() {
        System.out.println("Creando motor electrico");
        this.conectado = false;
    }

    public void conectar() {
        System.out.println("Conectando motor electrico");
        this.conectado = true;
    }

    public void activar() {
        if (!this.conectado) {
            System.out.println("No se puede activar porque no " +
              "esta conectado el motor electrico");
        } else {
            System.out.println("Esta conectado, activando motor" +
```

```
                " electrico....");
        }
    }

    public void moverMasRapido() {
        if (!this.conectado) {
            System.out.println("No se puede mover rapido el motor " +
              "electrico porque no esta conectado...");
        } else {
            System.out.println("Moviendo mas rapido...aumentando voltaje");
        }
    }

    public void detener() {
        if (!this.conectado) {
            System.out.println("No se puede detener motor electrico" +
              " porque no esta conectado");
        } else {
            System.out.println("Deteniendo motor electrico");
        }
    }

    public void desconectar() {
        System.out.println("Desconectando motor electrico...");
        this.conectado = false;
    }
}
```

Clase MotorElectricoAdapter.

Aquí se establece el puente por medio del **cual la clase incompatible puede ser utilizada**, hereda de **la clase Motor y por medio de la implementación** dada, **realiza la comunicación con la clase a adaptar** usando para esto una instancia de la misma...

```
public class MotorElectricoAdapter extends Motor{
    private MotorElectrico motorElectrico;

    public MotorElectricoAdapter(){
        super();
        this.motorElectrico = new MotorElectrico();
        System.out.println("Creando motor Electrico adapter");
    }
    @Override
    public void encender() {
        System.out.println("Encendiendo motorElectricoAdapter");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando motor electrico...");
        this.motorElectrico.moverMasRapido();
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor electrico");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```

```
public class AplicacionMotor {
          public static void main(String[] args) {


            Motor motor = new MotorComun();
```

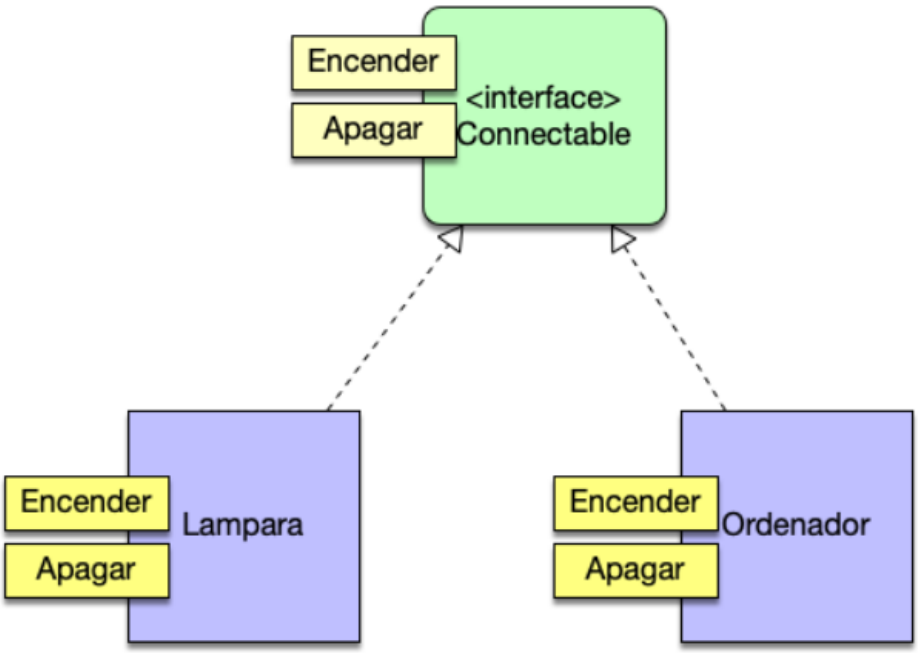
```
            motor.encender();
            motor.acelerar();
            motor.apagar();

             motor = new MotorElectricoAdapter() ;
            motor.encender();
            motor.acelerar();
            motor.apagar();

             motor = new MotorComun();
            motor.encender();
            motor.acelerar();
            motor.apagar();

          }

}
```

Observar como usamos el MotorElectrico como el **resto de motores comunes**, con la **clase MotorElectricoAdapter**.
Cread un **proyecto, montar y ejecutar el ejemplo**. El **resultado de la ejecución** será algo como:

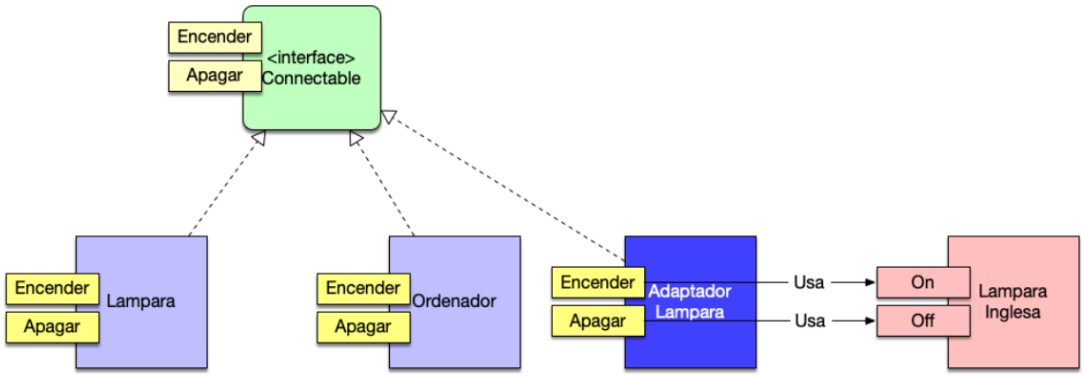
```
Creando el motor comun
encendiendo motor comun
acelerando el motor comun
Apagando motor comun
Creando motor electrico
Creando motor Electrico adapter
Encendiendo motorElectricoAdapter
Conectando motor electrico
Esta conectado, activando motor electrico....
Acelerando motor electrico...
Moviendo mas rapido...aumentando voltaje
Apagando motor electrico
Deteniendo motor electrico
Desconectando motor electrico...
Creando el motor comun
encendiendo motor comun
acelerando el motor comun
Apagando motor común
```

4.1.2 Practica independiente Adapter

Tenemos el siguiente modelo en Java con un interfaz Connectable que marca el comportamiento de Lampara y Ordenador



Pero nos viene un nuevo producto de Inglaterra llamado LamparaInglesa, y se pide adaptarla con el patrón adapter para que funcione en nuestro modelo de clases ya creado.



Se os proporciona el código. Realizar AdaptadorLampara para que el programa principal funcione.

Principal.java

```
public class Principal {
public static void main(String[] args) {
         Conectable l1= new Lampara();
         encenderAparato(l1);
         Conectable o1= new Ordenador();
         encenderAparato(o1);
         Conectable l2= new AdaptadorLampara();
         encenderAparato(l2);
}
private static void encenderAparato(Conectable l1) {
         l1.encender();
         System.out.println(l1.estaEncendida());
         }
}
```

Connectable.java

```
Public interface Connectable {

public void encender();

  public void apagar();

}
```

Lampara.java

```
public class Lampara implements Connectable {
  private boolean encendida;
  public boolean estaEncendida() {
    return encendida;
```

```
  }
  public void encender() {

    encendida=true;
  }

  public void apagar() {

    encendida=false;
  }

}
```

Ordenador.java

```
public class Ordenador implements Connectable {
  private boolean encendida;
  public boolean estaEncendida() {
    return encendida;
  }
  public void encender() {

    encendida=true;
  }

  public void apagar() {

    encendida=false;
  }
```

Pero nos llega una lampara inglesa cuyos métodos de interfaz son diferentes y tenemos que aplicar el patrón adapter para hacerla funcionar:

```
package com.arquitecturajava.ejemplo2;
```

```
public class LamparaInglesa {
private boolean isOn;
public boolean isOn() {
return isOn;
}
public void on () {
isOn=true;
}
public void off() {
isOn=false;
}
}
```

4.1.3 Bridge

Intent

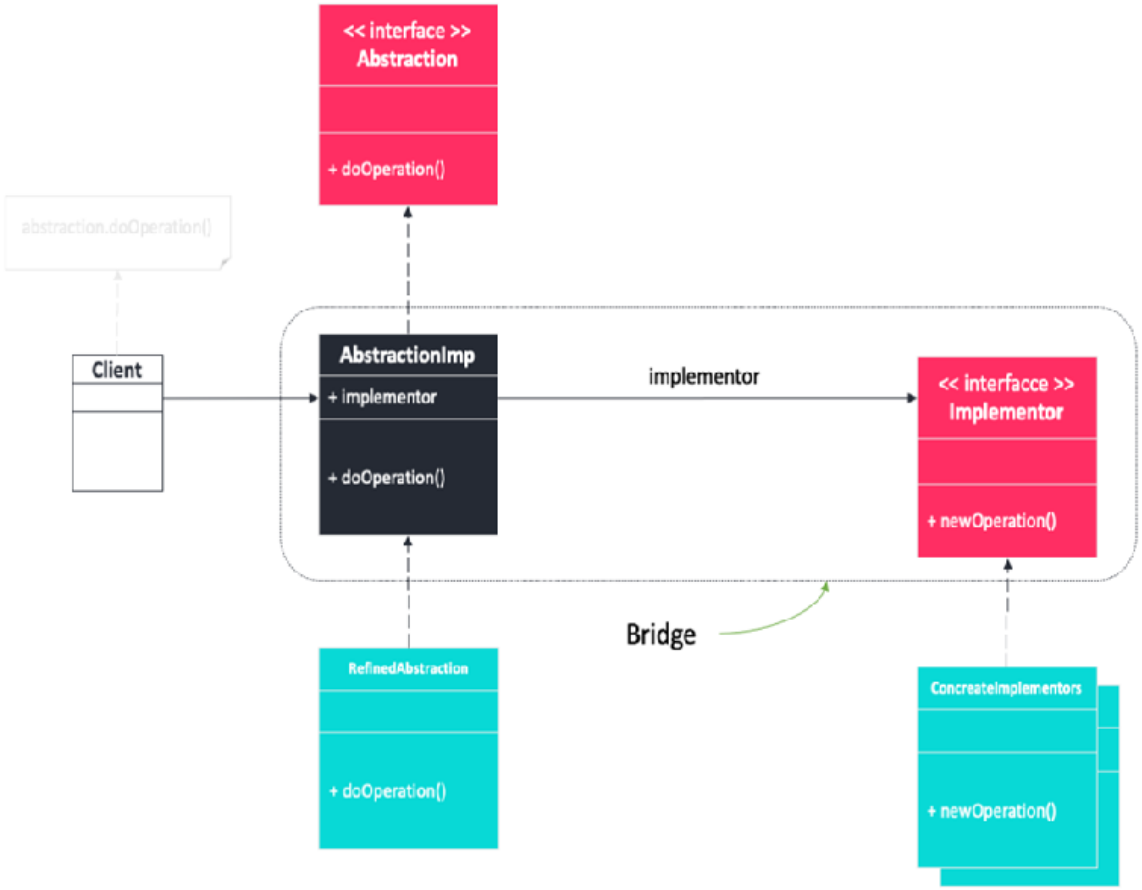
Divide un componente complejo en dos jerarquías de herencia independientes pero relacionadas: la ***abstracción*** funcional y la **implementación** interna.

Solución

El diagrama siguiente muestra una posible implementación de Bridge.

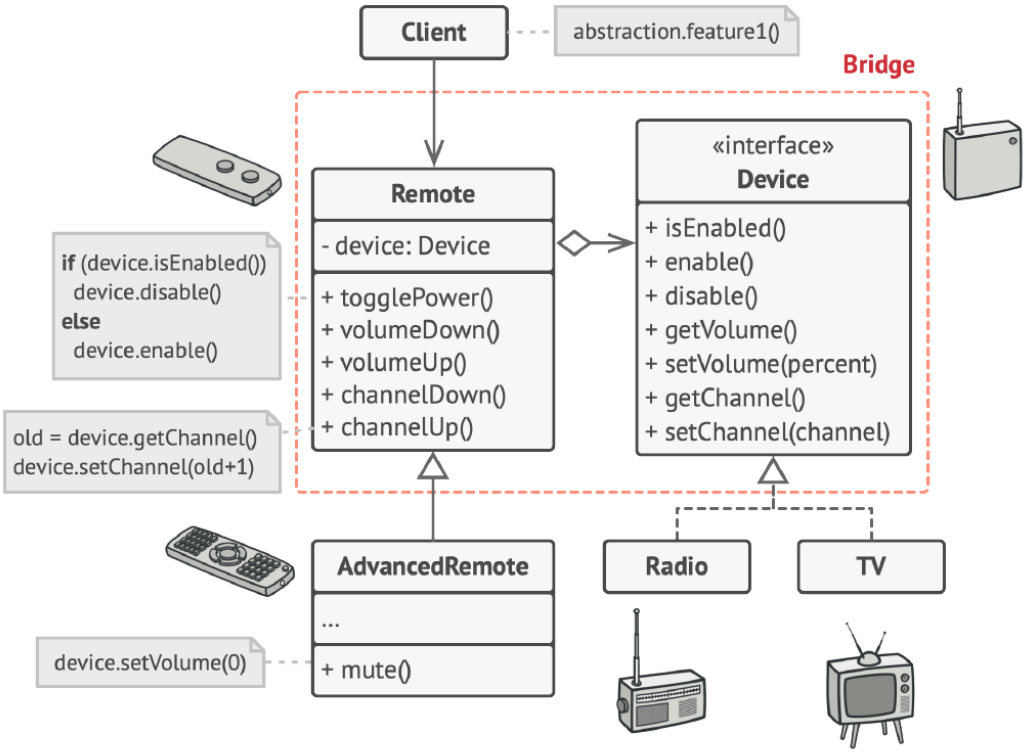
- **Abstraction**: este es el componente de abstracción.
- **Implementor**: esta es la implementación abstracta.
- **RefinedAbstraction**: este es el componente refinado.

- **ConcreateImplementors**: esas son las implementaciones concretas.



Ejemplo del mundo real

Tenemos unos aparatos, **radio y televisión que son la implementación concreta** de un interfaz. Queremos que los use **una mando a distancia, que es la abstraccion refinada**



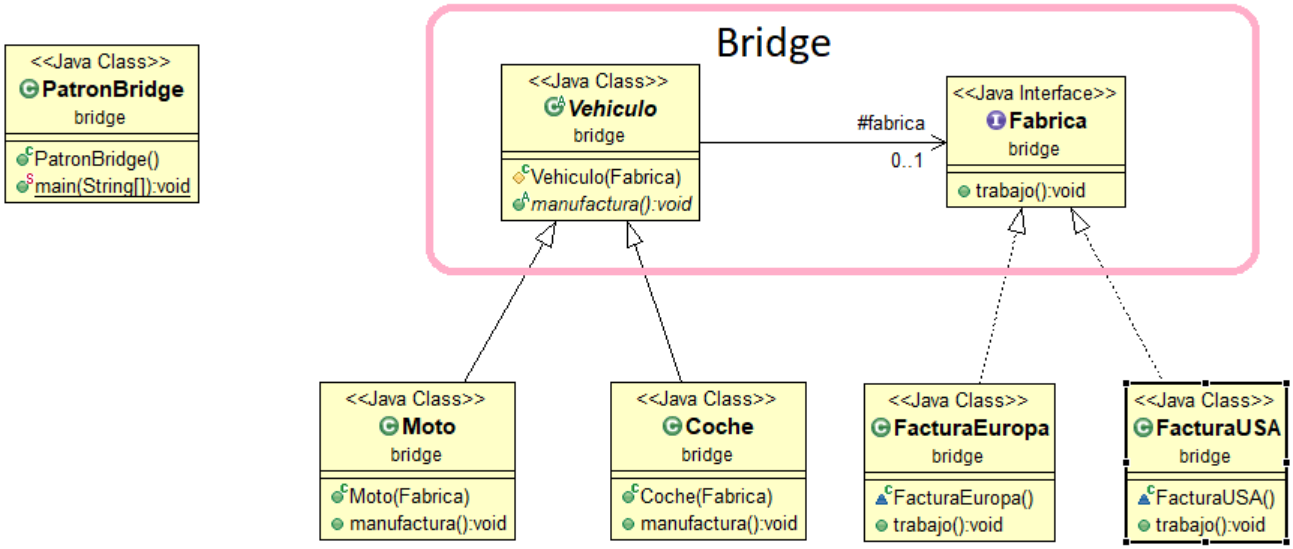
En interfaz **device**, es el **implementator**,y **Remote la clase abstracta o abstracción**.

Para ello usamos **una abstracción, una clase abstracta Remote**, que entre sus atributos tiene **el interfaz Device**. Esta c**lase Abstracta hace de puente entre el mando a distancia el AdvancedRemote y el Device**, el interfaz. Pero si os fijas añade cosas extras. Porque **un control remoto complejo** o un móvil, **puede hacer más cosas que las que puede hacer una radio**. De esta manera **no limitamos a un mando a distancia a hacer las cosas de una tele o una radio,** podrá **hacer más y tendrá mas métodos**. Para por ejemplo controlar un coche de radiocontrol, o **un reproductor de DVD**. Pero le **permitimos acceder a la tele y la radio**.

Vamos a explicar este patrón con un ejemplo

Ejemplo patrón Bridge.

Vamos a realizar una aplicación para un marca de vehículos que tiene una fabrica coches y motocicletas. Vamos **a necesitar este patrón porque tenemos dos vehículos**, y **dos fábricas, que fabrican diferente**. Este es el problema, **observar como Moto y Coche tienen un interfaz manufactura(),** que **no es compatible con el interfaz de Fabrica, trabajo().**



Vamos a identificar elemento a elemento de este patrón bridge. **Las fabricas FacturaEuropa serán la implementación concreta del implementator** que **es el interfaz fabrica el implementator**. **Moto y Coche necesitan usar de FacturaEuropa y FacturaUSA para fabricarse en Europa y en Estados Unidos.**

Si **no tuviéramos este patrón**, tendríamos que **construir 4 subclases más**. MotoFacturaEuropa MotoFacturaUSA, CocheFacturaEuropa, CocheFacturaUSA. Si añadimos otro vehículo como Camion, más clases todavía, CamionFacturaEuropa CamionFacturaUSA. Este **patrón de diseño ahorra la generación de muchas subclases**

Usando Bridge nos evitamos ese problema. La **solución al problema es sencilla y es parecida a Factory**. Uno de los **atributos de la clase Vehículo será del tipo interfaz fábrica**. En **tiempo de ejecución le asignamos la fábrica que nosotros queremos**, Europea o Americana. Veámoslo en el siguiente ejemplo:

A vehículo le estamos asignando una fabrica con un parámetro cuyo tipo es el interfaz fábrica. No sabemos aun que fabrica va a ser

```
abstract class Vehiculo {
 protected Fabrica fabrica;

 protected Vehiculo(Fabrica fabrica)
 {
```

```
        this.fabrica = fabrica;


    }
```

Tanto **Coche como Moto heredan de vehículo** pero también reciben el parámetro Fabrica en su constructor.

```
class Coche extends Vehiculo {
    public Coche(Fabrica fabrica)
    {
        super(fabrica);
    }
```

Tenemos definido un interfaz fabrica, el implementator, y **varias fábricas que son implementaciones concretas o concreciones** del interfaz fábrica. Dos fabricas la Europea y la de USA

```
//Implementor para bridge
interface Fabrica
{
    abstract public void trabajo();
}
```

Como **veis cada fábrica, manufactura de manera diferente**, pero tienen muchas cosas en común. Por eso **tienen su abstracción en el interfaz Fabrica**.

```
class FacturaEuropa implements Fabrica {
    @Override
    public void trabajo()
    {
        System.out.print("Producido en Europa\n");
    }
}


class FacturaUSA implements Fabrica {
    @Override
    public void trabajo()
```

```
 {

     System.out.println(" Producido en USA.\n");
 }
}
```

Este diseño nos da la posibilidad de fabricar coches y motos en cualquier fábrica sólo hay que asignarle la fábrica de antemano. En el ejemplo podéis ver como fabricamos dos vehículos diferentes, en dos fabricas diferentes.

```
Vehiculo vehiculo1= new Coche(new FacturaEuropa());
     vehiculo1.manufactura();
     Vehiculo vehiculo2 = new Moto(new FacturaUSA());
     vehiculo2.manufactura();
```

PatronBridge.java

```
abstract class Vehiculo {
 protected Fabrica fabrica;

 protected Vehiculo(Fabrica fabrica)
 {
     this.fabrica = fabrica;

 }

 abstract public void manufactura();
}



class Coche extends Vehiculo {
 public Coche(Fabrica fabrica)
 {
     super(fabrica);
 }
```

```
 public void manufactura()
 {
     System.out.print("Coche ");
     fabrica.trabajo();

 }
}


class Moto extends Vehiculo {
 public Moto(Fabrica fabrica)
 {
     super(fabrica);
 }

 @Override
 public void manufactura()
 {
     System.out.print("Moto ");
     fabrica.trabajo();

 }
}

//Implementor for bridge pattern
interface Fabrica
{
 abstract public void trabajo();
}


class FacturaEuropa implements Fabrica {
 @Override
 public void trabajo()
```

```
 {
     System.out.print("Producido en Europa\n");
 }
}

class FacturaUSA implements Fabrica {
 @Override
 public void trabajo()
 {

     System.out.println(" Producido en USA.\n");
 }
}

//Demonstration of bridge design pattern
public class PatronBridge {
 public static void main(String[] args)
 {
     Vehiculo vehiculo1= new Coche(new FacturaEuropa());
     vehiculo1.manufactura();
     Vehiculo vehiculo2 = new Moto(new FacturaUSA());
     vehiculo2.manufactura();

     Vehiculo vehiculo3= new Coche(new FacturaUSA());
     vehiculo1.manufactura();
     Vehiculo vehiculo4 = new Moto(new FacturaEuropa());
     vehiculo2.manufactura();
 }
}
```

4.1.4 Practica independiente de Patrón Bridge.

Para el modelo anterior de lamparas se añadirá el método fabricar(Fabrica fabrica) a Connectable. Se creará el interfaz Fabrica del que herederan a FabricaInglesa y FabricaEspanola que serán necesarias para la creación del Connectable y del patron Bridge. Se pedirá el modelo de clases para este ejercicio.

4.1.5 Patrón Composite

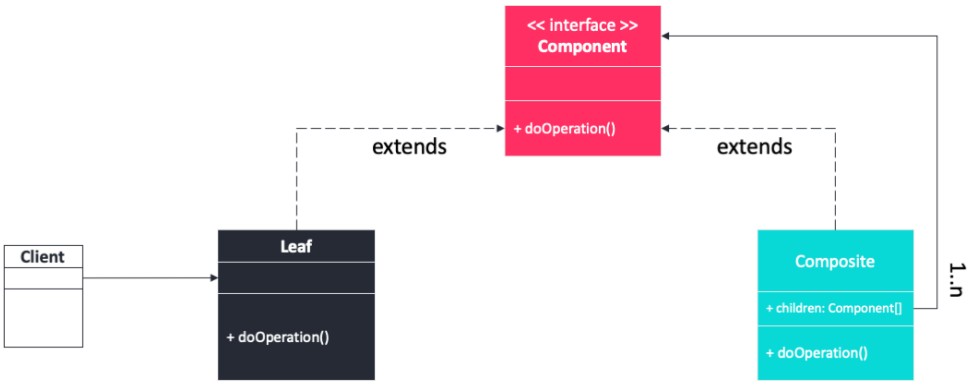
El patrón Composite nos permite mantener en un mismo diseño objetos simples y objetos compuestos. Estos objetos compuestos estarán formados por objetos simples y otros objetos compuestos y así sucesivamente. Lo estudiaremos en el tema 8, con listas.

Intent

El patrón compuesto **permite crear estructuras jerárquicas de árbol de complejidad variable** al tiempo que permite que cada elemento de la estructura funcione con una interfaz uniforme.

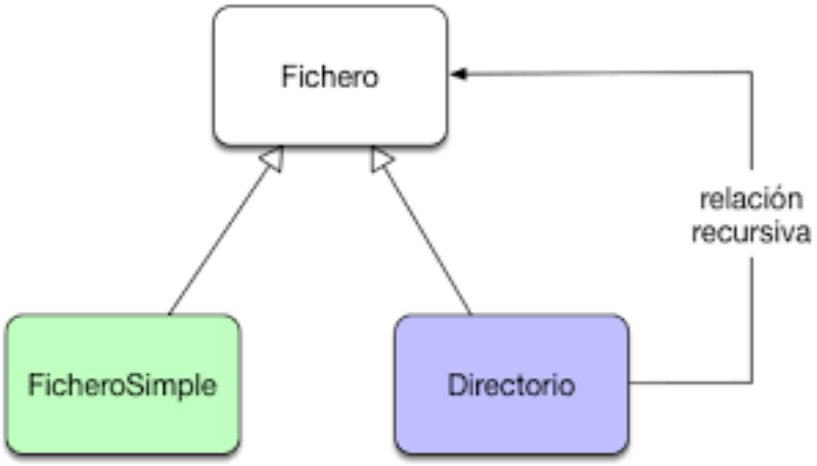
Solución

El patrón Compuesto combina objetos en estructuras de árbol para representar toda la jerarquía o una parte de la jerarquía.



Un ejemplo de **patrón Composite sería el manejo de ficheros**. Tenemos ficheros **simples que son las hojas, y directorios que a su vez están compuestos de otros ficheros**. Como veis hay una relación de recursividad en este patrón. Debemos aplicar nuestra lógica de programación a Ficheros simples y a directorios**. Pero los directorios contienen ficheros y más directorios**. Con lo que **habrá que aplicar nuestra lógica de programación a los ficheros y directorios** que contiene el directorio recursivamente. El proceso acabara cuando hayamos tratado todos los archivos y ficheros que contiene un

directorio.

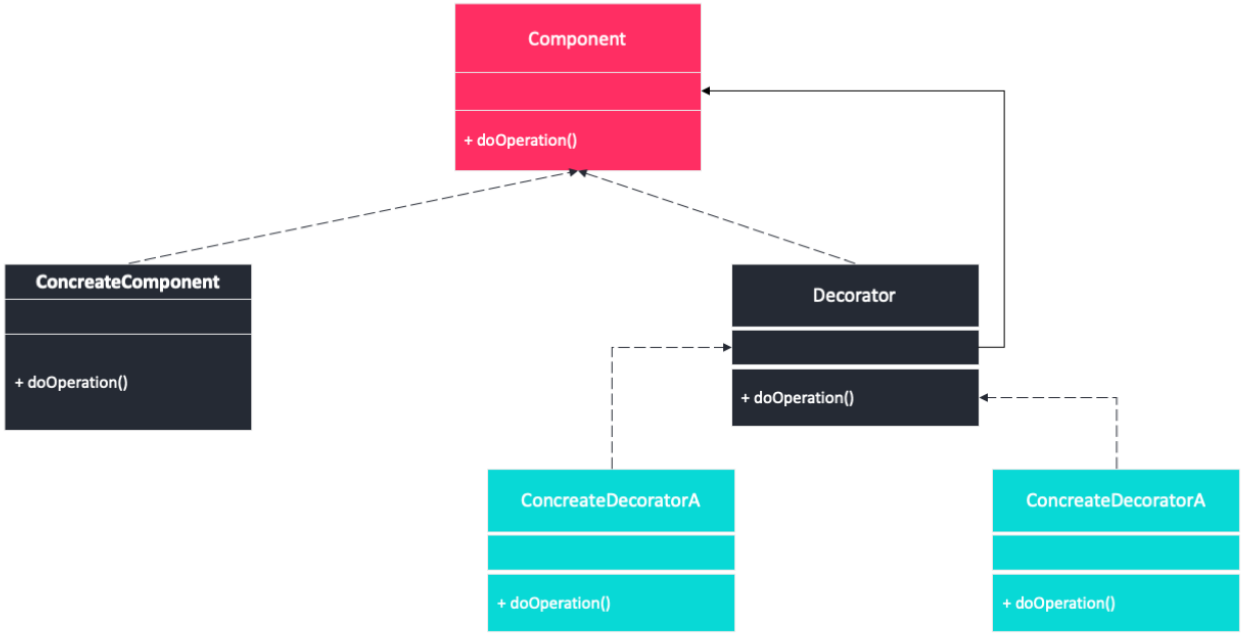


4.1.6 Decorator Pattern. Patrón Decorator. Practica guiada.

Intent

El **patrón Decorator** **permite agregar o quitar la funcionalidad** del **objeto sin cambiar la apariencia externa** o la función del objeto. Te permite añadir nuevos comportamientos **a los objetos colocando estos objetos dentro objetos tipo contenedor** especiales que contienen los comportamientos.

Solución



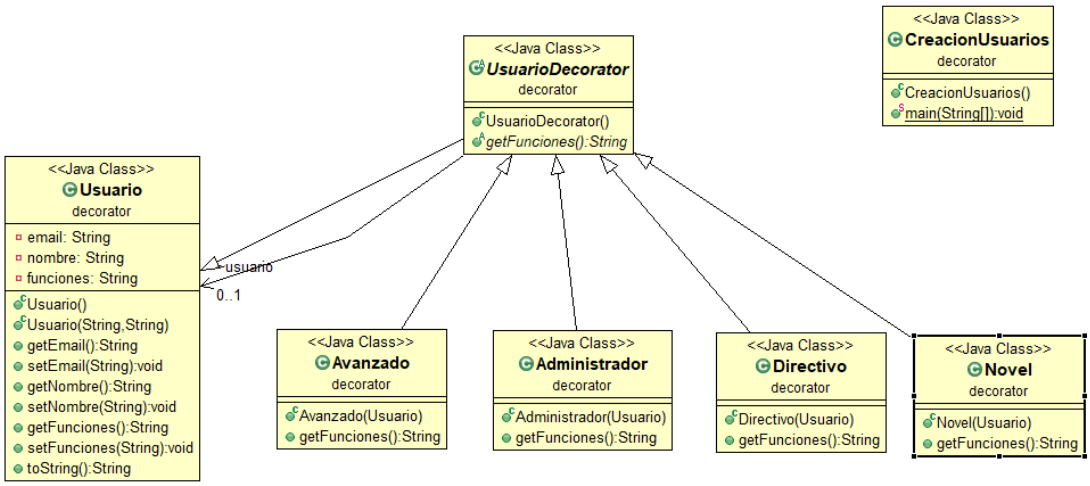
Cambia **la funcionalidad de un objeto de una manera que es transparente** para **sus clientes** **mediante una instancia de una subclase** de la clase original que **delega las operaciones** sobre el **objeto original**.

ConcreteComponent define un objeto sobre el que se pueden agregar responsabilidades adicionales.
El **Decorator** **mantiene una referencia a un Objeto Component** y **define una interfaz que se ajusta** a la interfaz de **Component.**
ConcreteDecorators amplía la funcionalidad del componente agregando **un estado** o agregando **un comportamiento**.

Vamos **a explicar este patrón sobre el siguiente ejemplo**, y el siguiente modelo. Queremos **crear usuarios de nuestro sistema**, que **tengan características especiales, como por ejemplo**, **tipos de usuario**. Tendremos **Avanzado, Administrador, Directivo y Novel**. Y queremos que en **nuestro Usuario resultado se mezclen estas categorías**, para dar como resultado un usuario con varias categorias. Por ejemplo, **que nuestro usuario pueda ser Directivo, Administrador y Avanzado** al mismo tiempo.

Partiendo de esa **base**, **nuestro ConcreteComponent es usuario. Nuestro Decorator es UsuarioDecorator**, y **nuestros ConcreteDecorators** son

Avanzado, Administrador, Directivo y Novel. Son como **ingredientes que podemos añadir a nuestro usuario**. Podríamos **hacerlo igual con el ejemplo de las pizza, y pizzaFactory**, añadiendo los ingredientes a nuestro gusto sobre la Pizza. Pero **este ejemplo es más realista**.



Vamos a empezar viendo nuestra clase principal, que **tiene la función main y como creamos esos objetos con características mezcladas**. Observad como creamos un usuario DirectorTecnico añadiéndole categorías. Lo primero que sea usuario, nuestro objeto concreto, segundo es nuevo, Con lo que es Novel, es Administrador, y directivo.

```
usDirectorTecnico = new Novel(usDirectorTecnico);
usDirectorTecnico = new Administrador(usDirectorTecnico);
usDirectorTecnico = new Directivo(usDirectorTecnico);
```

El **usuario resultado tendrá el comportamiento de los tres ConcreteDecorators**. El comportamiento lo vamos a obtener del método getFunciones(), que es la ingrediente, la función que va a añadir cada ConcreteDecorator. Cada **ConcreteDecorator, añade una función.** Es el secreto del patrón Decorator.

CreacionUsuarios.java

```
public class CreacionUsuarios {
```

```
          public static void main(String[] args) {



                    Usuario usDirectorTecnico = new Usuario("luloperez@company.
com", "Lulo");


                    usDirectorTecnico = new Novel(usDirectorTecnico);
                    usDirectorTecnico = new Administrador(usDirectorTecnico);
                    usDirectorTecnico = new Directivo(usDirectorTecnico);


                    System.out.println("Las funciones de nuestro nuevo director
técnico son: ");
                    System.out.println(usDirectorTecnico.getFunciones());


                    Usuario usAdministradorAvanzado = new Usuario("luloperez@co
mpany.com", "Lulo");
                    usAdministradorAvanzado = new Administrador(usAdministrador
Avanzado);
                    usAdministradorAvanzado = new Avanzado(usAdministradorAvanz
ado);



                    System.out.println("Las funciones de nuestro nuevo administ
rador avanzado son: ");
                    System.out.println(usAdministradorAvanzado.getFunciones())
;


          }


        }
```

La **clase Usuario es la clase típica** que ya hemos visto **antes y no hace falta explicarla en detalle**. Sólo hacer referencia al comportamiento que vamos a modificar, getFunciones(). En el **atributo funciones guardamos** las **funciones del usuario**. La **primera es la básica.** En funciones **es donde cada ConcreteDecorator** añade **otras funciones.**

```
private String funciones="Como usuario normal el usuario podrá acceder y mane
jar los módulos standard de la aplicación";
```

Usuario.java

```
public class Usuario {


        private String email;

        private String nombre;

        private String funciones="Como usuario normal el usuario podrá accede
r y manejar los módulos standard de la aplicación.";


        public Usuario() {


        }


        public Usuario(String email, String nombre) {


                this.email = email;

                this.nombre = nombre;


        }



        public String getEmail() {

                return email;

        }


        public void setEmail(String email) {

                this.email = email;

        }


        public String getNombre() {

                return nombre;

        }


        public void setNombre(String nombre) {

                this.nombre = nombre;

        }
```

```
        public String getFunciones() {
                return funciones;
        }

        public void setFunciones(String funciones) {
                this.funciones = funciones;
        }



        public String toString() {

                return "usuario {email=" + this.email + ",nombre= " + this.no
mbre + "}";

        }



}
```

Y ahora vamos con lo nuevo en este patrón, el Decorator. El **Decorator** es **una clase abstracta** que **hereda de la clase Component**, en este **caso de Usuario**. Igualmente obliga a **los ConcreteDecorators** a **implementar un comportamiento**, getFunciones()

```
public abstract String getFunciones();
```

```
public abstract class UsuarioDecorator extends Usuario {
        Usuario usuario;

        public abstract String getFunciones();
}
```

En este caso el **comportamiento que queremos modificar añadiendo** y **sobreescribiendo es getFunciones()**. Lo declaramos como **abstract**. Y ahora vamos a ir viendo los ConcreteDecorators. Son todos iguales sólo hará falta explicar uno.

Directivo es un ConcreteDecorator que hereda de UsuarioDecorator. Su trabajo principal será sobreescribir getFunciones añadiendo la nueva funcionalidad.

```
                public String getFunciones() {

                        return usuario.getFunciones() +  " Como usuario avan
zado maneja los módulos más sensibles de la aplicación para dirigir la empres
a. ";


        }
```

Como veis en **getFunciones()** añadimos una función más al usuario. De esta manera, podemos añadir **tantas funcionalidades como ConcreteDecorators**.
Además, y esto es de la **cosecha del profesor**, en el **constructor hago una copia, o clone, del objeto usuario,** con el **fin de mantener todos los campos**, y que no haya perdida de datos. En muchos ejemplos de Decorators, no se realizar. **Es mejor mantener todos los datos del objeto**. Si os fijais, **UsuarioDecorator** hereda de **Usuario**, y Directivo de **UsuarioDecorator**. Por tanto, **Directivo** es un Usuario, **hereda de Usuario**.
Lo primero en el **patrón Decorator** es quedarnos **con una referencia del usuario original** que **recibimos en el constructor**. Pero además, estamos copiando los campos de usuario para no tener perdida de datos.

Nos quedamos con la referencia al usuario.

```
public Directivo(Usuario usuario) {
                this.usuario=usuario;
```

Y copiamos los campos, **porque Decorator es también un Usuario**. En otros ejemplos de Decorator, no haría falta, pero **en este si lo hace**.

```
            this.setEmail(usuario.getEmail());


            this.setNombre(usuario.getNombre());
```

```
public Directivo(Usuario usuario) {
                this.usuario=usuario;


                this.setEmail(usuario.getEmail());


                this.setNombre(usuario.getNombre());
```

```
                }
```

Directivo.java

```
public class Directivo extends UsuarioDecorator{



         public Directivo(Usuario usuario) {
                 this.usuario=usuario;

                 this.setEmail(usuario.getEmail());

                 this.setNombre(usuario.getNombre());

                }
                public String getFunciones() {
                        return usuario.getFunciones() +  " Como usuario directivo maneja los módulos más sensibles de la aplicación para dirigir la empresa. ";


        }
}
```

El **resto de clases Concrete Decorator que indico a continuación**, son exactamente igual cada una añade una funcionalidad distinta, extra.

Administrador.java

```
public class Administrador extends UsuarioDecorator{



        public Administrador (Usuario usuario) {

                this.usuario=usuario;

                this.setEmail(usuario.getEmail());
```

```
                this.setNombre(usuario.getNombre());

        }

        @Override
        public String getFunciones() {

                return usuario.getFunciones() + " La función del administrado
r será la de instalar y configurar la aplicación.";
        }

}
```

Avanzado.java

```
public class Avanzado extends UsuarioDecorator{

         public Avanzado(Usuario usuario) {

                        this.usuario=usuario;

                        this.setEmail(usuario.getEmail());

                        this.setNombre(usuario.getNombre());

                }

        @Override
        public String getFunciones() {


                return usuario.getFunciones() + " Como usuario avanzado coord
ina y maneja los módulos más complejos de la aplicación ";

        }
```

```
}
```

Novel.java

```
public class Novel extends UsuarioDecorator{



        public Novel(Usuario usuario) {



this.usuario=usuario;

                this.setEmail(usuario.getEmail());

                this.setNombre(usuario.getNombre());

        }

        @Override
                public String getFunciones() {



                        return usuario.getFunciones() + " Como usuario novel
tiene acceso a los módulos más básicos de la aplicación. ";

        }

}
```

Si **ejecutais el programa comprobareis** como **cuando creo los usuarios con las características mezcladas**, se **añaden todas las funcionalidades**, en múltiples sobreescrituras. Es un **uso intensivo de polimorfismo**.

```
  Usuario usDirectorTecnico = new Usuario("luloperez@company.com", "Lulo");

                usDirectorTecnico = new Novel(usDirectorTecnico);
                usDirectorTecnico = new Administrador(usDirectorTecnico);
```

```
                usDirectorTecnico = new Directivo(usDirectorTecnico);
```

En la ejecución se ha ejecutado, el **getDescripcion()** de **Novel, Administrador y Avanzado**, en el **orden en que lo hemos añadido**. Si os fijais recogemos el resultado de los news, en una variable usuario, todos **los Decorators son Usuarios igualmente**. Pero para crear un usuario completo, **necesitas el Component**, **componente original Usuario**.

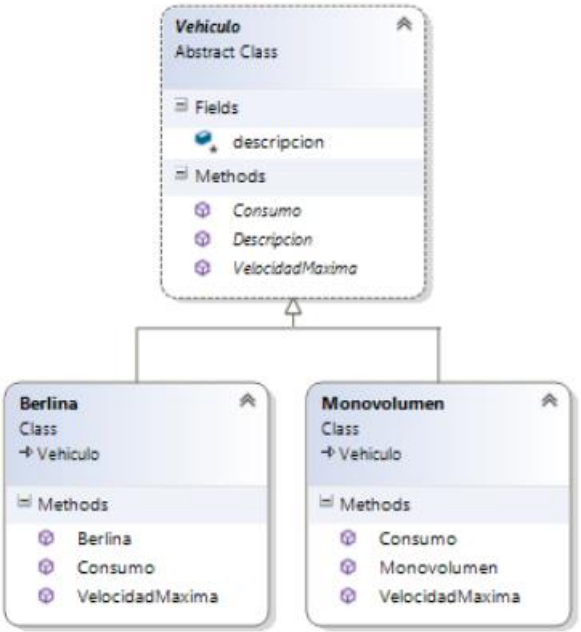
Ejemplo de ejecución

```
Las funciones de nuestro nuevo director técnico son:
Como usuario normal el usuario podrá acceder y manejar los módulos standard
de la aplicación. Como usuario novel tiene acceso a los módulos más básicos
de la aplicación.  La función del administrador será la de instalar y
configurar la aplicación. Como usuario directivo maneja los módulos más
sensibles de la aplicación para dirigir la empresa.
```

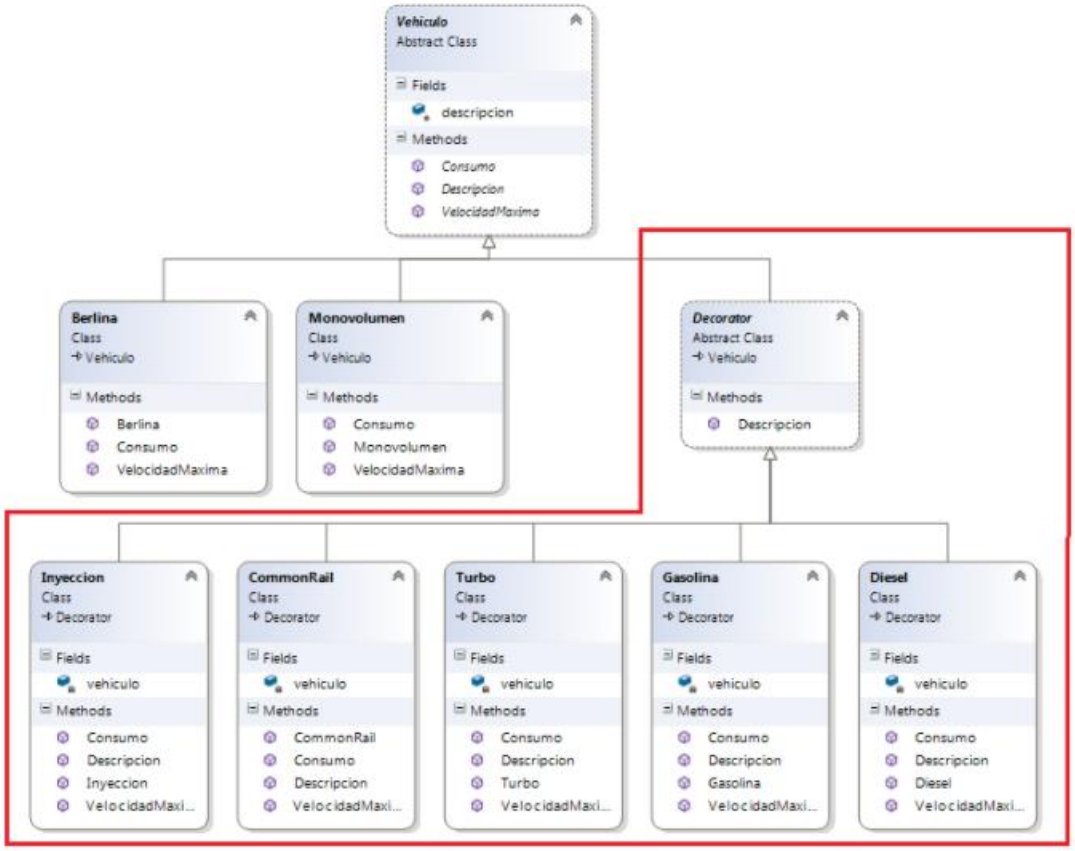
4.1.7 Practica independiente de patrón Decorator

Dado el siguiente modelo de clases. Se pide

1. Implementar el modelo inicial, siguiendo el diagrama de clases.



2 **Añadir una clase Decorator que cambie la descripción de cada vehículo añadiendo de características: Dieses, Gasolina, Turbo, Common Rail e Inyeccción** teniendo en cuenta que estas descripciones se pueden combinar. Aplicando el patrón Decorator para ello.



4.2 Patron Facade. Practica guiada

El objetivo de patrón es ofrecer una clase que maneje un **framework, librería o API concreta** como si **fuera una sóla clase**. Por dentro **hay muchas clases y subclases que se inicializan y realizan una función** pero el usuario de **la librería sólo usa una clase** para **manejar toda la API**. Casi todos los frameworks y API de Java, Android, Microsoft .NET, javascript, etc, funcionan de esta manera y suelen estar bien diseñados. Ofrecen una **clase de entrada a una funcionalidad pero detrás hay muchas clases**. Por ejemplo, **la APIStream** que vimos en el tema anterior está diseñada de esta manera, con **varios patrones Facade**, entre otros patrones.

Intent

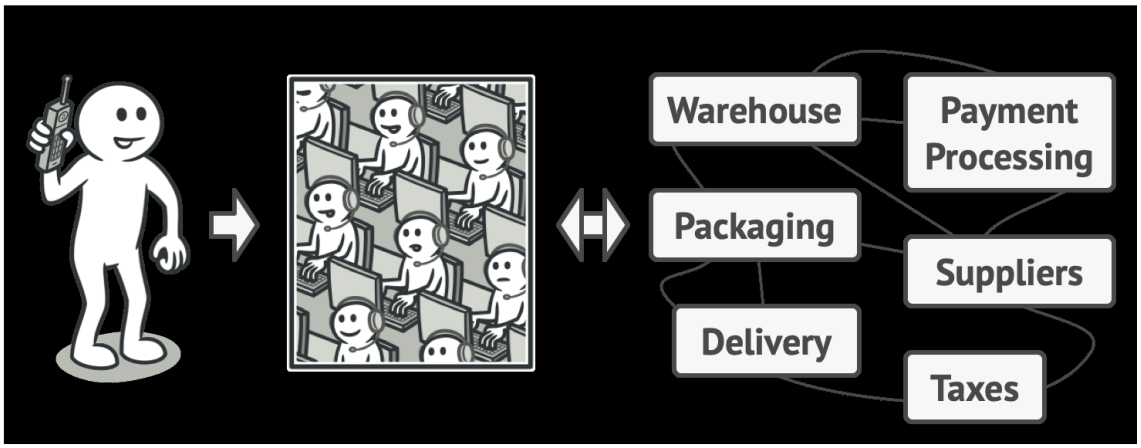
- **Proporciona una interfaz unificada a un conjunto de interfaces** en un subsistema. **Facade define una interfaz de nivel superior que facilita el uso del subsistema.**
- Envuelva un subsistema complicado con una interfaz más sencilla.

Problema

Imagina que debes hacer que tu código funcione con un conjunto amplio de objetos que **pertenecen a una biblioteca o framework sofisticado**. Normalmente, tendría que **inicializar todos esos objetos, mantener seguimiento de las dependencias**, **ejecutar métodos en el orden correcto**, y así sucesivamente.
Como **resultado, la lógica de negocios de sus clases dependería de los detalles de implementación de las clases 3a parte**, lo que dificulta la comprensión y el mantenimiento.

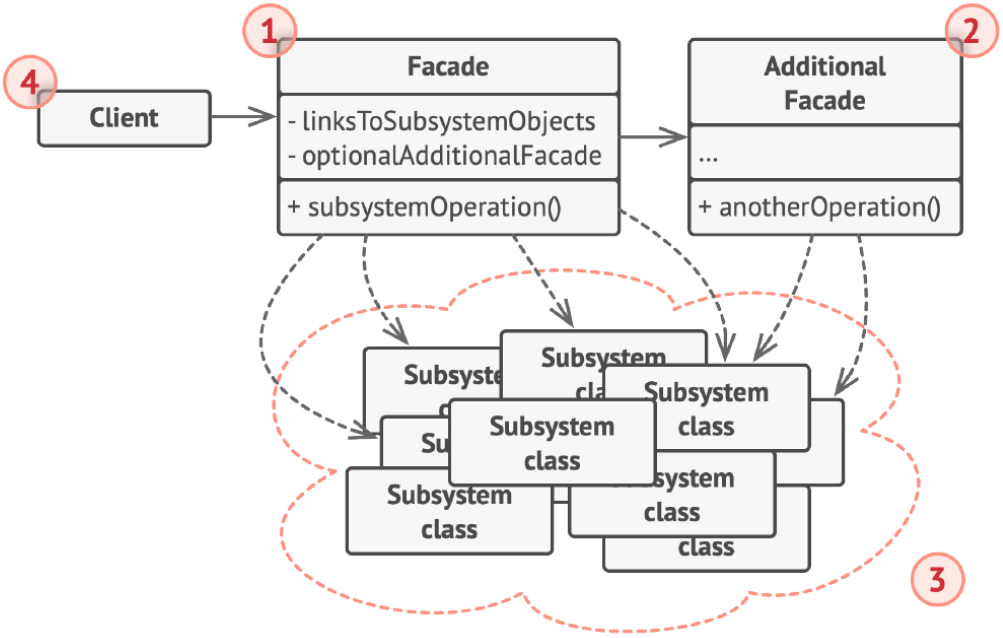
Solución

Cuando **llamas a una tienda para realizar un pedido telefónico**, un **operador es su fachada a todos los servicios y departamentos de la tienda**. El **operador le proporciona una interfaz de voz simple para el sistema de pedidos**, las pasarelas de pago y varias Servicios.



Estructura

Como veis hay una o varias clases de entrada para manejar todo el sistema de clases de la librería.



1. El **Facade proporciona un cómodo acceso** a **una parte de la funcionalidad del subsistema**. Sabe dónde dirigir la solicitud del cliente y cómo operar con todas las partes del subsistema, los conjuntos de subclases.

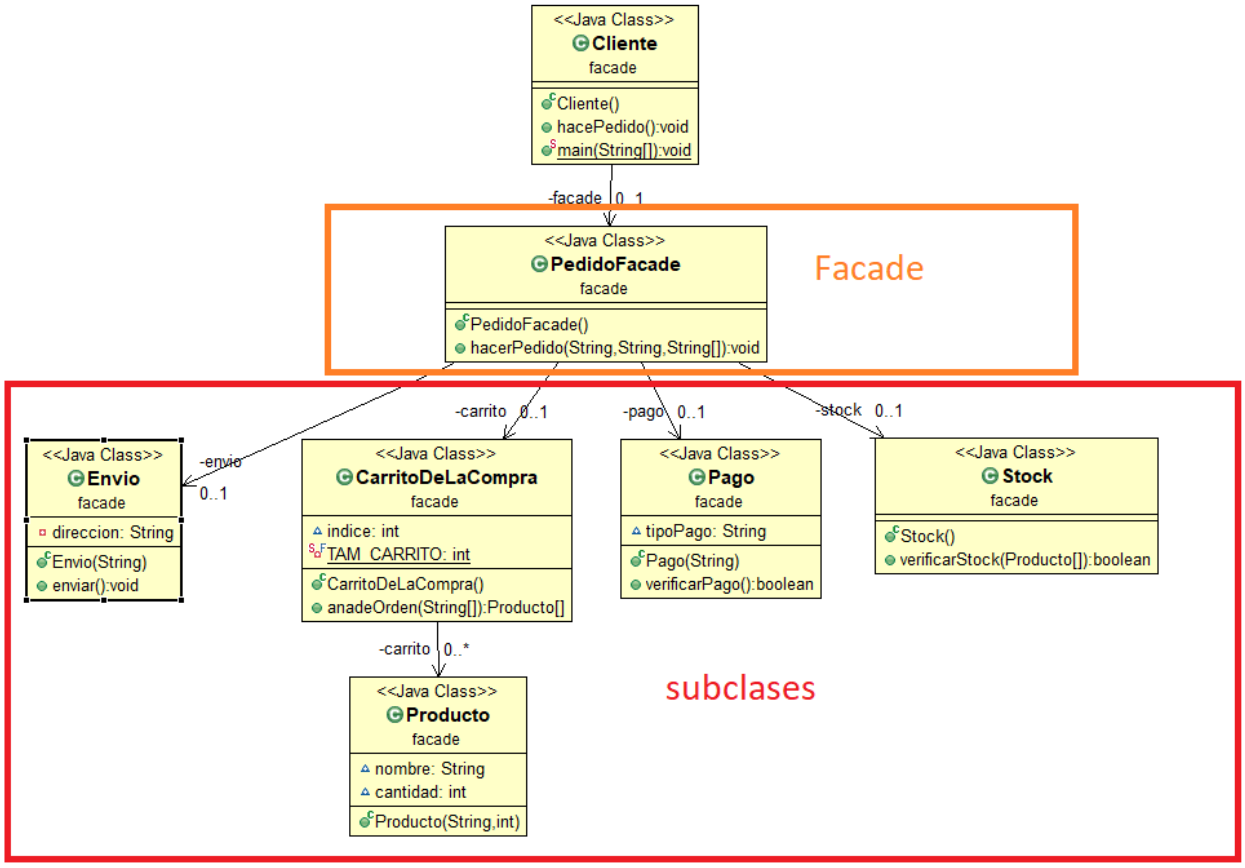
2. Se puede **crear una clase Facade adicional** para evitar que se **un solo Facade** con características no relacionadas que **podrían manejar otra estructura compleja de nuestra librería**. Las fachadas adicionales pueden ser utilizadas por tanto los clientes como otras fachadas. Es lo habitual en frameworks de gran tamaño.

3. El **Subsistema Complejo** consta de **docenas de objetos diversos**. Para que el conjunto de todos realice una labor significativa, tienes que bucear en los detalles de implementación del subsistema, como **la inicialización objetos** en el **orden correcto** y suministre con datos en el formato adecuado. Las **clases de subsistema no son conscientes de la existencia de la fachada. Ellos operan dentro del sistema** y **trabajan entre sí directamente**.

4. El **Cliente utiliza el objeto facade** en **lugar de llamar a los objetos del subsistema** directamente.

Otra vez vamos a usar un ejemplo para explicar **el patrón facade**. En este caso, vamos a **crear un subsistema de pedidos** con carrito de la compra, verificación de Stock, pago y envío, como el que os podéis encontrar en cualquier página

web.



El cliente va a usar la clase **FacadePedido** para hacer el pedido**. Al cliente no le interesa como los pedidos se hacen por dentro**, sólo **el resultado**. El **FacadePedido se encargará de crear el pedido, crear el carrito, el stock, el pago, y el envio**. El **cliente usará todas esas subclases a través de FacadePedido**, pero no tendrá que hacer nada más que **introducir los productos, la forma de pago y la dirección de envío**.

En la clase **PedidoFacade creamos todas las clases que necesitamos** y hacemos **el pedido en orden**. Primero **añadimos los ítems**. Luego **comprobamos stock y pago**. Y si **todo ha ido correcto enviamos el pedido.**

`PedidoFacade`.java

```
public class  PedidoFacade {
```

```
        private CarritoDeLaCompra carrito = new CarritoDeLaCompra();
        private Pago pago;
        private Stock stock;
        private Envio envio;

        public PedidoFacade() {



        }

        public void hacerPedido(String direccion, String tipoPago, String ...
items ){

                Producto[] carritoArray = carrito.anadeOrden(items);

                stock= new Stock();
                pago= new Pago(tipoPago);
                if (stock.verificarStock(carritoArray) && pago.verificarPago(
)) {

                        envio=new Envio(direccion);
                        envio.enviar();

                }

        }

}
```

En la **clase cliente sólo hacemos el pedido**. El cliente **no tiene porque saber como funciona nuestro negocio** o framework por dentro. Su única función **aquí es realizar el pedido con los productos** que le interesan, su **método de pago y su dirección de envio**, **a través del Facade**.

Cliente,java

```

```

```
public class Cliente{

        private  PedidoFacade facade = new PedidoFacade ();

        public Cliente() {



        }

        public void hacePedido () {

        facade.hacerPedido("Calle numero 7", "Tarjeta", "Producto1", "Product
o2", "Producto3");



        }
    public static void main(String[] args) {

      Cliente cl= new Cliente();
      cl.hacePedido();
   }
}
```

El resto de **subclases son parte del framework, no necesitan de explicación a este** nivel de los apuntes. Mirar **el código he intentar entenderlo**.

Producto.java

```
public class Producto {

        String nombre ="";
        int cantidad=0;
        public Producto(String nombre, int cantidad) {
                super();
                this.nombre = nombre;
                this.cantidad = cantidad;
```

```
        }



}
```

CarritoDeLaCompra.java

```
public class CarritoDeLaCompra {

        private Producto carrito[];
        int indice=0;
        private static final int TAM_CARRITO=3;

        public CarritoDeLaCompra() {

                 carrito = new Producto[TAM_CARRITO];

        }



        public Producto[] anadeOrden(String ... items) {

        carrito = new Producto[items.length];

                for(String item: items) {
                        carrito[indice]=new Producto(item,1);
                        indice= indice+1;

                }

        return carrito;

        }

        }
```

Stock.java

```
public class Stock {

	public Stock() {



	}

	public boolean verificarStock(Producto ...items) {


		System.out.println("Stock correcto");
		return true;

	}

}
```

Pago.java

```
public class Pago {

	String tipoPago="";

	public Pago(String tipoPago) {

		this.tipoPago=tipoPago;
	}

	public boolean verificarPago() {
```

```
                if (tipoPago=="Tarjeta" || tipoPago=="PayPal")    {

                        System.out.println("Pago correcto");
                        return true;

                }

                else  {
                        System.out.println("Pago invalido");
                        return false;
                }
        }

}
```

Envio.java

```
public class Envio {
private String direccion ="";

        public Envio(String direccion) {

                this.direccion=direccion;
        }



        public void enviar() {

                System.out.println("Pedido enviado a la dirección :" + direcc
ion);
        }



}
```

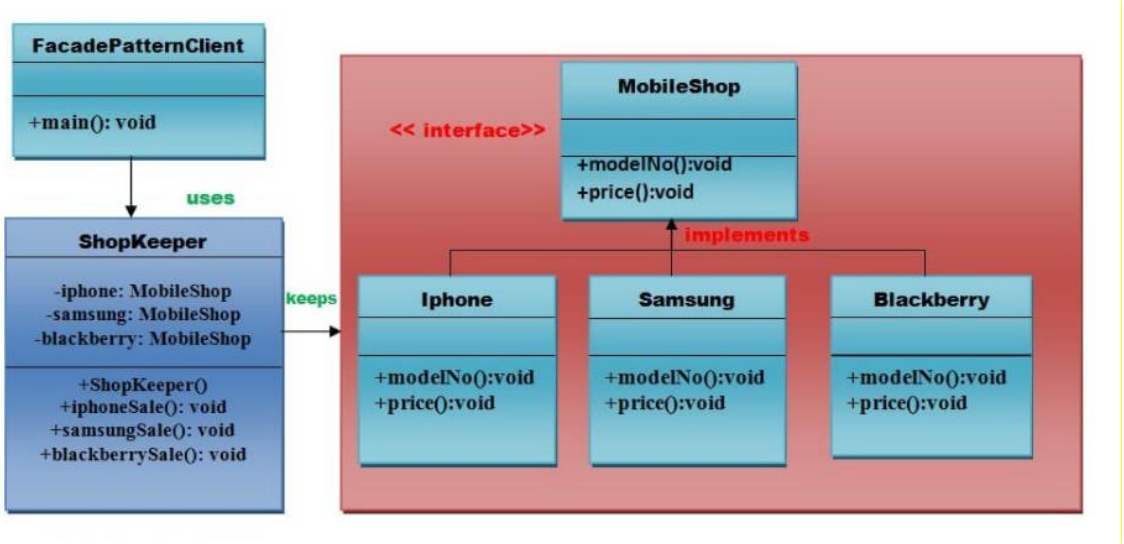
Nota: Muy importante que montéis los ejemplos y los ejecutéis.

Fijaos como el objeto Facade hace todos los pasos por el cliente en la ejecución

```
Stock correcto
Pago correcto
Pedido enviado a la dirección :Calle numero 7
```

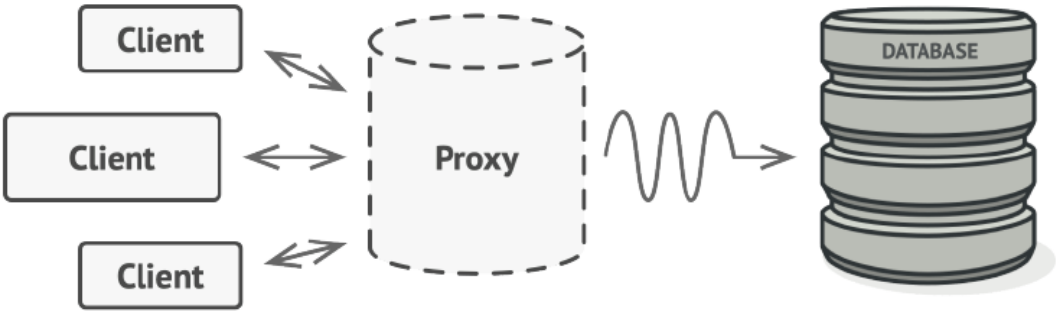
4.2.1 Practica independiente de Patrón Facade

Se pide implementar el siguiente modelo de clases con el patrón Facade. La clase Facade será el ShopKeeper que controlará las ventas de los diferentes teléfonos.



4.3 Introduccion al patron Proxy

Proxy es un patrón de diseño estructural que le **permite proporcionar un sustituto o marcador de posición** para otro objeto. Un **proxy controla el acceso al objeto original**, lo **que le permite realizar algo antes o después de que la solicitud** llegue al objeto original.



El **objeto Proxy intermedio se disfraza a si mismo de base de datos**. Da **la sensación de ser la base de datos, pero no lo es**. Se lo proporcionamos al cliente como un controlador de los accesos a base de datos. Nos puede **permitir realizar caching y evaluación Lazy**. Este **modelo es más avanzado, lo podremos introducir desde un punto de vista práctico** en el tema de base de datos.

5 Bibliografía y referencias web

Referencias web

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Bibliografía

Dive Into DESIGN PATTERNS, Alexander Shvets, Refactoring.Guru, 2020