Unidad 8. Diseño Orientado a Objectos. Principios Sólidos. Patrones de diseño creacionales.

Contenido

1 Actividad inicial

Se pregunta a los alumnos si han oído **hablar de los patrones de diseño o de los principios sólidos de diseño**. Y se propone el **siguiente problema y pregunta** dado el siguiente código

```
interface  SueldoTrabajadores {

      double calculaSueldo();
      double calculaImpuestos();

}

Public class Trabajador implements SueldoTrabajadores {

}
```

Que será mas correcto, declarar la variable Trabajador1 y trabajado2 como variable de interfaz `SueldoTrabajadores`

```
SueldoTrabajadores Trabajador1 y trabajado2;
```

O como Variable de clase:

```
Trabajador Trabajador1 y trabajado2;
```

Deben escribir las respuestas por escrito dando su razonamiento.

En Java premiamos o intentamos programar para las abstracciones. Ir arriba todo lo posible en la jerarquía. Si podemos programamos para un interfaz. Si me programa sólo hace calculo de sueldos, uso el interfaz. Si tiene que tocar otros detalles de trabajador uso la Clase. Igual de aplicable es para las superclases o clases padre.

2 Introducción.

En este **capítulo vamos a explicar los diferentes patrones o acercamientos**

que usaremos para resolver nuestros problemas de programación en programación orientada a objetos. Empezaremos por los **patrones más sencillos** e iremos avanzando hacia patrones más complicados.

Pero primero **explicaremos los principios solidos de diseño**, en los que se **fundamentan la programación orientada a objetos y los patrones de diseños**. Son cinco que estudiaremos en el siguiente capitulo.

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 nuestras 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**.

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; //
```

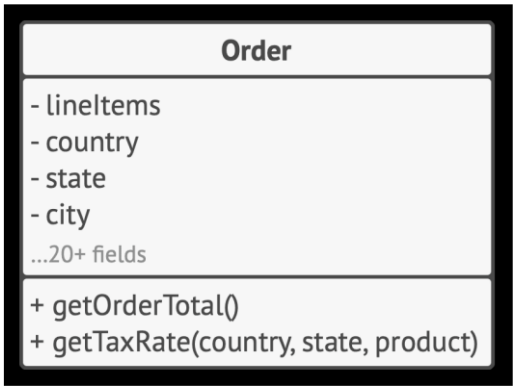
```
            }
         return total;
       }
```

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

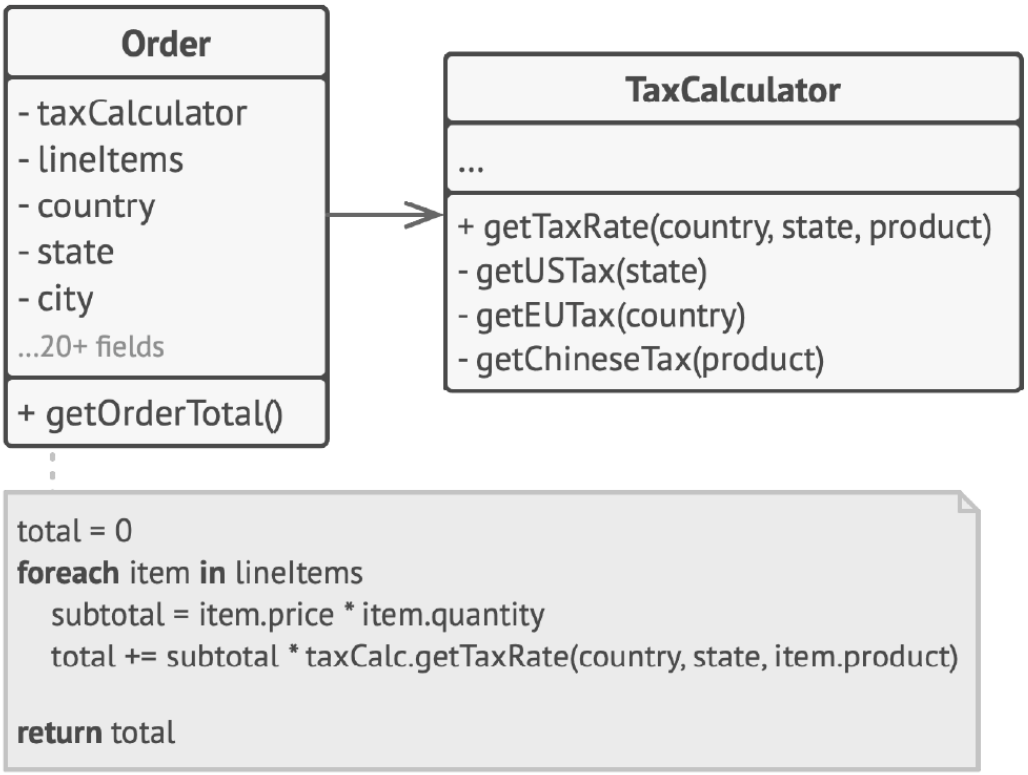
```
public double getTotalPedido(Producto ...pedidos) {
       double total = 0
              for(Producto Item:pedidos )
                     total += item.precio * item.cantidad*getImpuestos(item);


                     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.



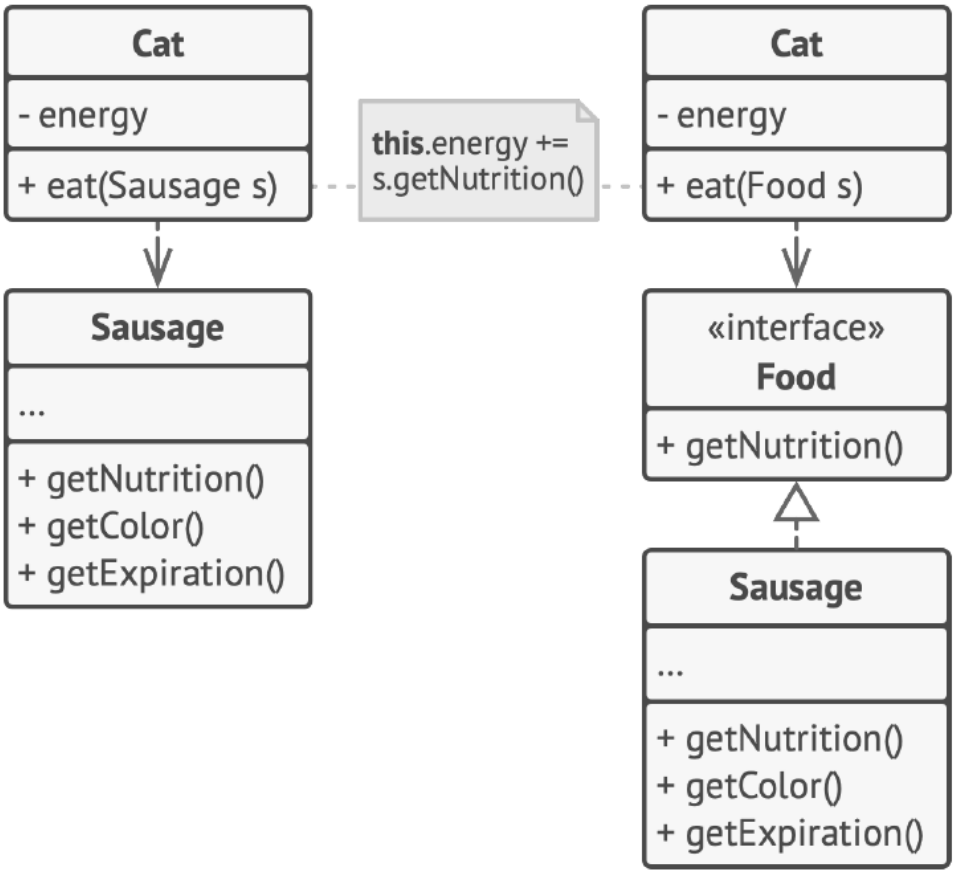
3.4 Programa usando interfaces no implementaciones en las variables.

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.

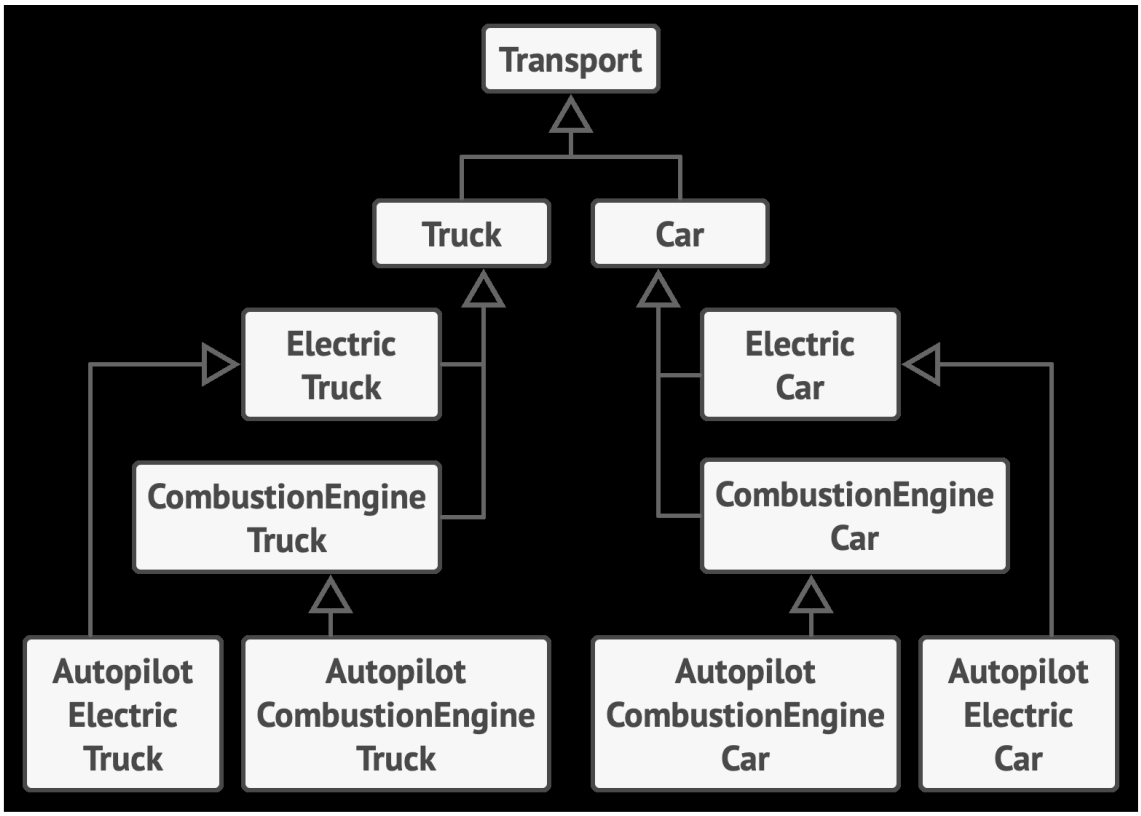
2. Al sobreescribir métodos, **debe asegurarse de que el nuevo comportamiento es compatible con el comportamiento base**. Es importante porque los objetos de la subclase se pueden pasar a cualquier código que espera objetos de la superclase y no quieres que código falle.

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.

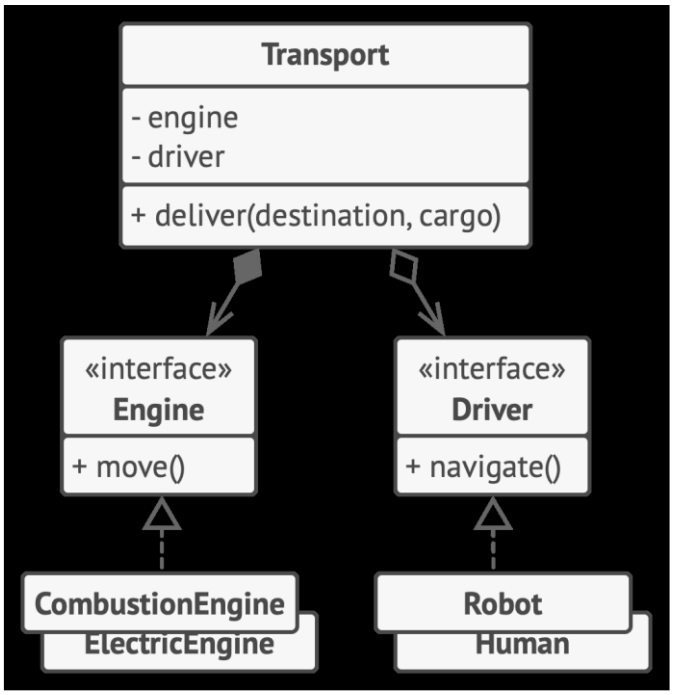


Como se puede ver, **cada parámetro adicional da como resultado la multiplicación el número de subclases**. Hay un montón de código duplicado entre subclases porque una subclase no puede extender dos clases al mismo tiempo.

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

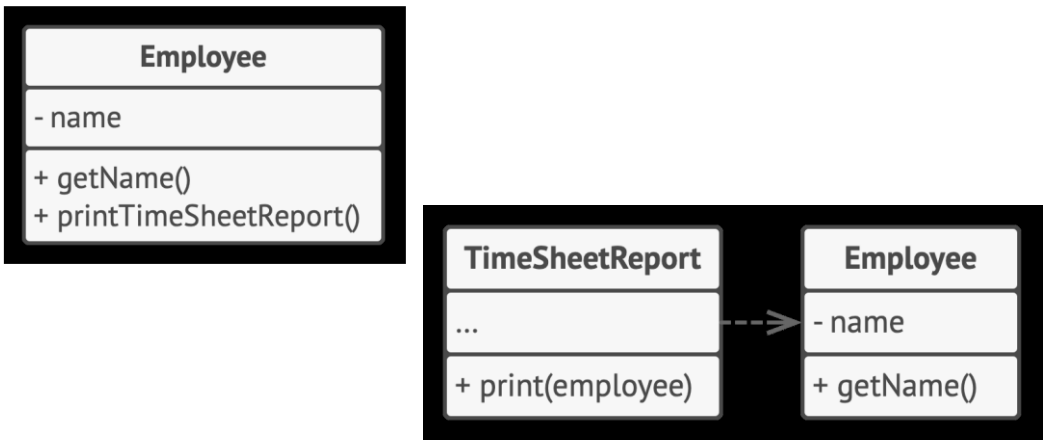
El **principio de responsabilidad única** lo que intenta es **hacer cada clase responsable de una única funcionalidad**, para separar o dividir bien las funcionalidades de nuestra aplicación. Se **hace todo el tiempo en interfaces funcionales y lambdas**.

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.

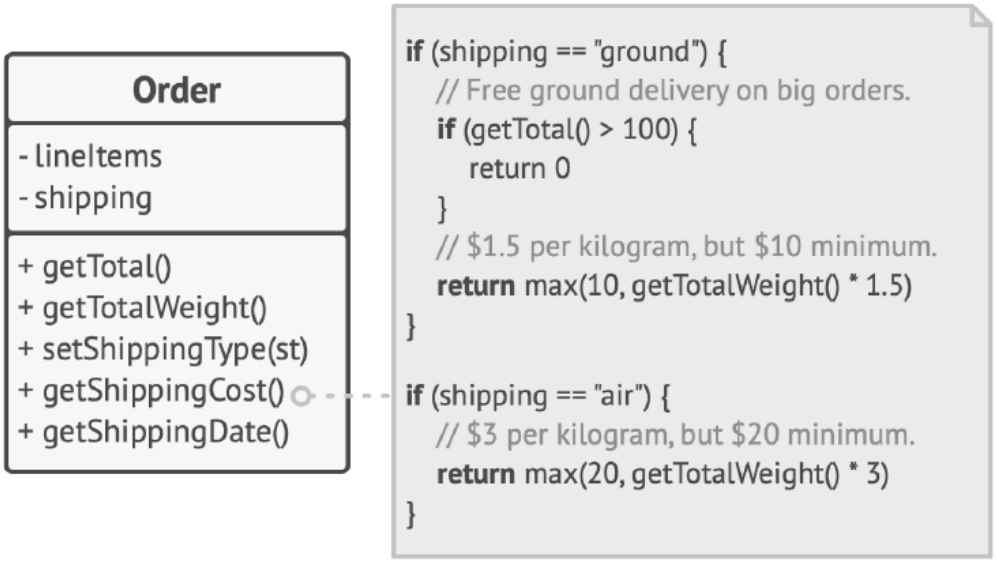
Cuando **se empieza a trabajar con este principio**, **es confuso porque las palabras abiertas y cerradas suenan mutuamente excluyentes**. Pero en términos de este principio, **una clase puede ser abierta (para extensión**) y **cerradas (para su modificación)** al **mismo tiempo**.

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**.

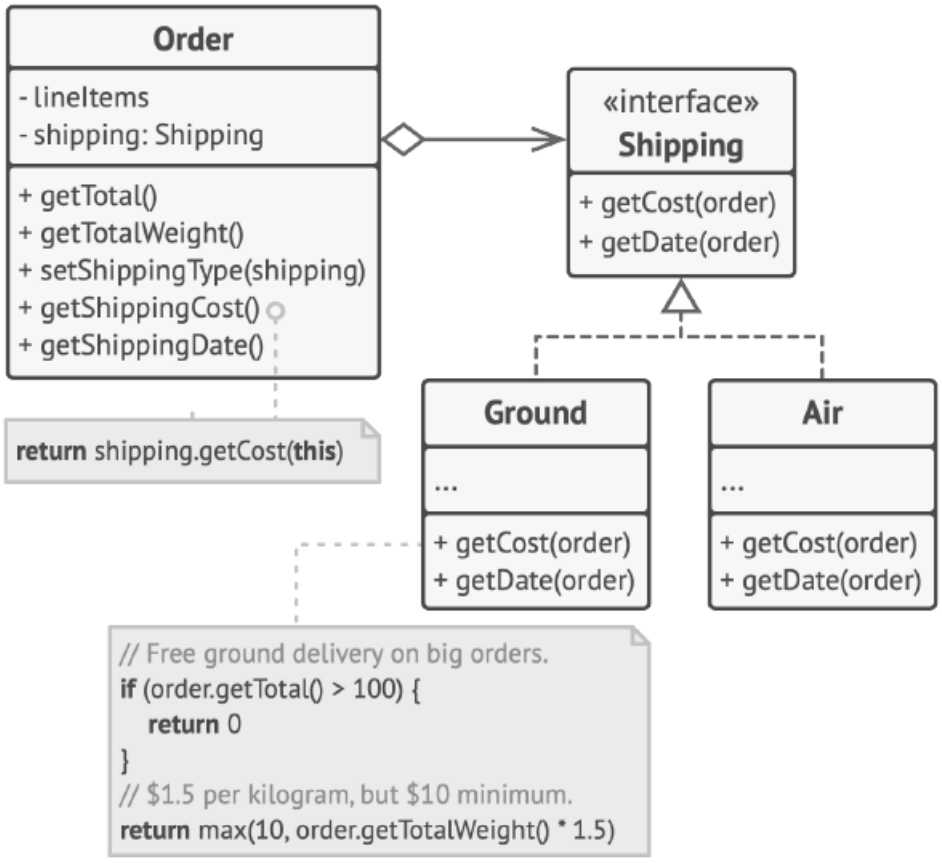
Este principio no está destinado a aplicarse a todos los cambios en una clase. Si sabes que **hay un error en la clase, hay que arreglarlo**, no crear una subclase para él.

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.

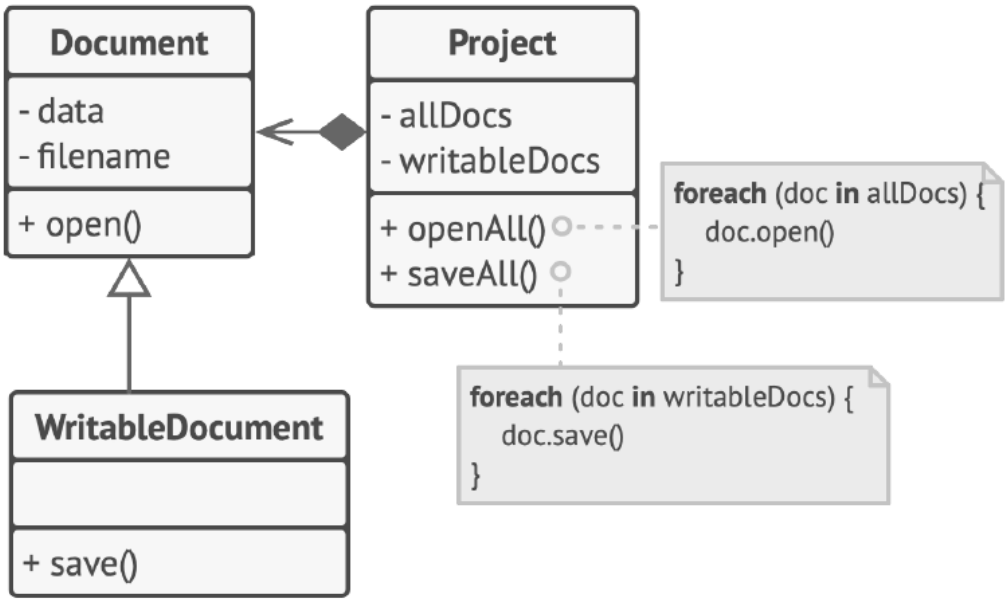
Los **tipos de parámetros y los tipos retornados de un método de una subclase** deben **coincidir o ser más abstracto que los tipos de parámetros en el método de la Superclase.** Vamos a tener un ejemplo.
Supongamos que hay una clase con un **método que se supone que alimenta gatos: feed (Cat c)** .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

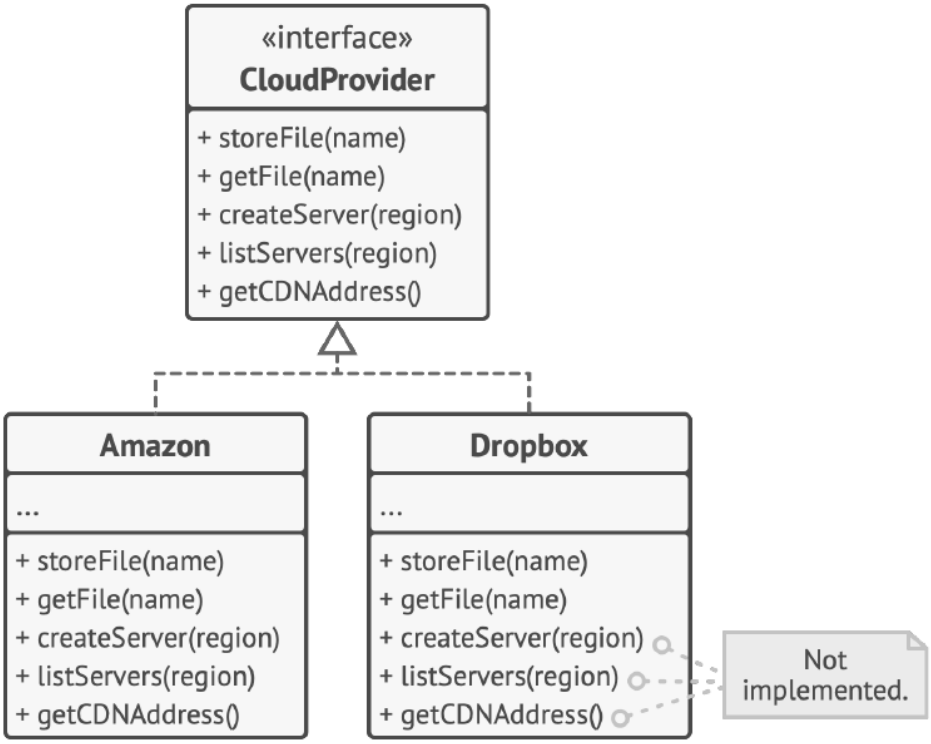
Los clientes no deben ser forzados a depender de los métodos que no usan y necesitan.

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.**

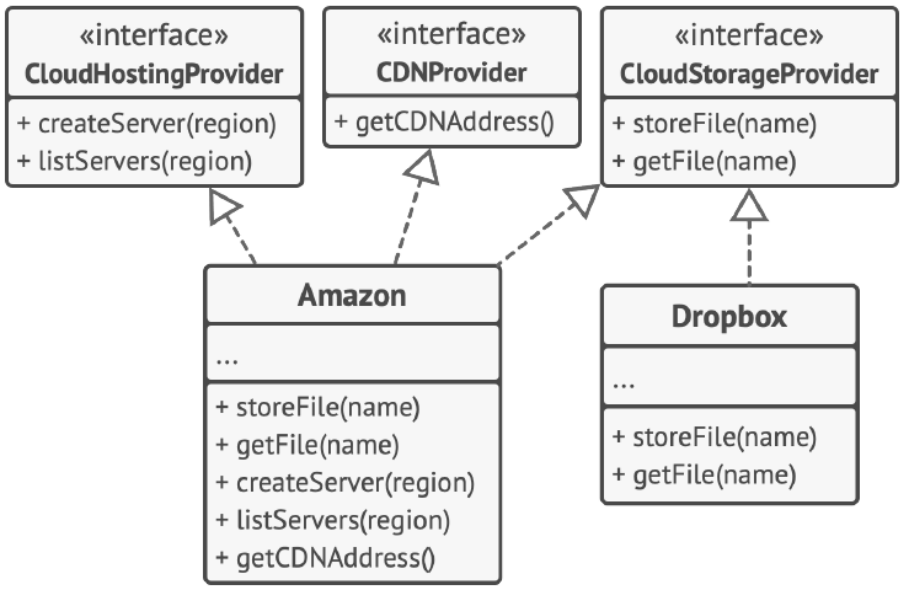
De acuerdo con este principio, **los interfaces demasiado amplios deberían descomponerse en mas específicos**. Esto **es debido a que las clases clientes deben implementar sólo los métodos necesarios**, no los que no necesitan. En caso contrario, si **tienes una interface demasiado compleja**, un **cambio 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.

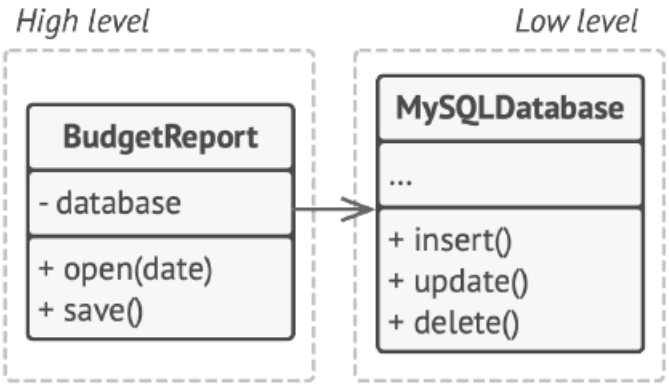
3. Una **vez que las clases de bajo nivel implementan estos interfaces**, **dependen del nivel de lógica de negocio**, **invirtiendo la dirección** de la dependencia original.

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.

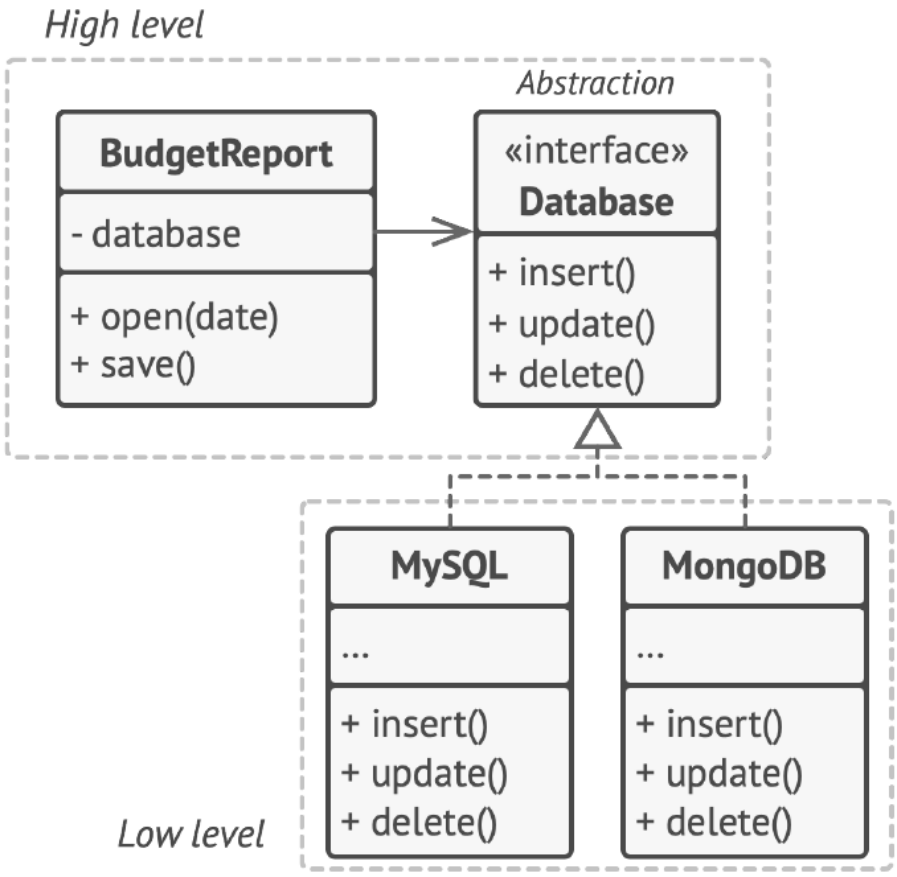
Por ejemplo:

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, pero no va hacer falta añadir nada nuevo. La clase Trabajador esta cerrada a nuevas funcionalidades, y está abierta a que

otras hereden o extiendan de ella.

```
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. De esta manera puedo crear un Conserje almacenado en una variable de tipo Interfaz o de tipo SueldoTrabajadores y el programa funciona exactamente igual.**

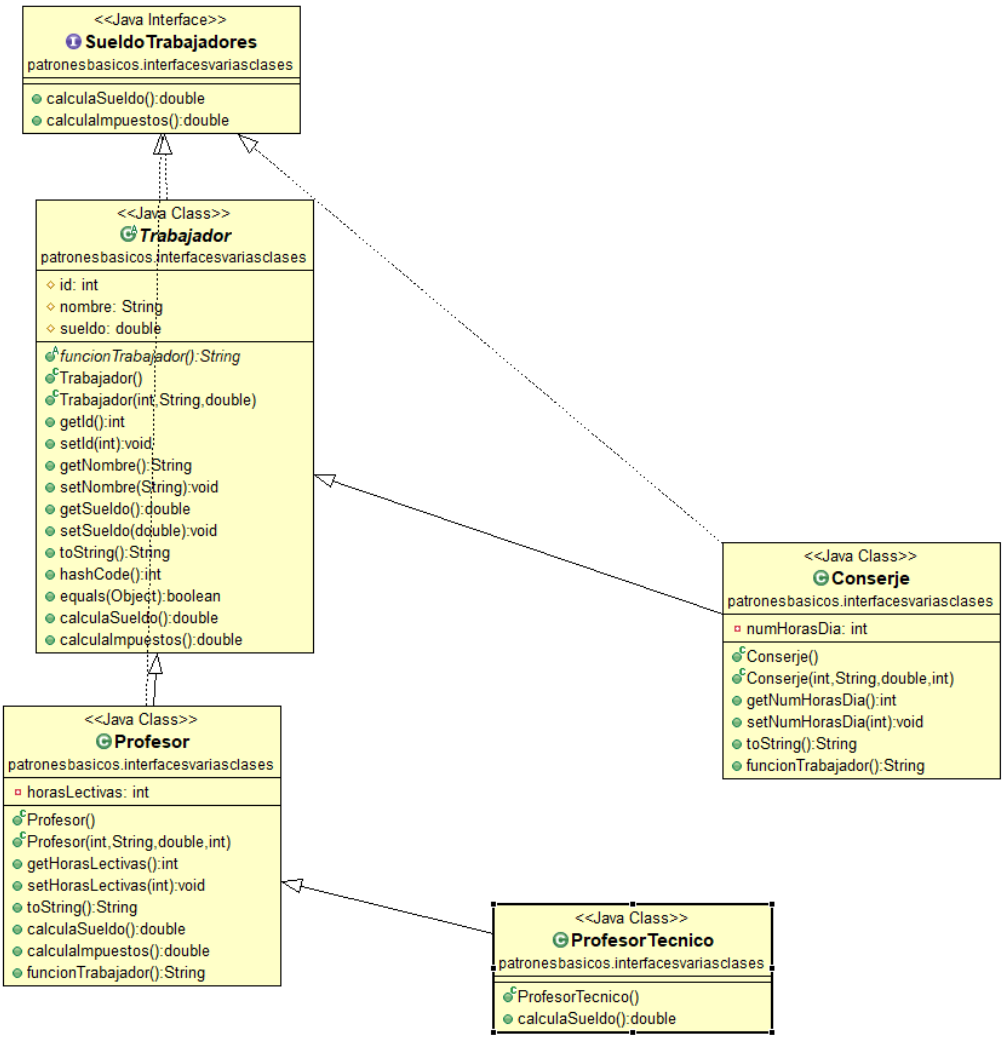
Nuestro programa está calculando impuestos, en el ejemplo para calcular impuestos sólo necesitamos variables de tipo interfaz, no especializaciones como veis marcado en verde. Programamos para las abstracciones, genérico, no para las implementaciones.

```
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 de creación

Vamos a ver en esta primera parte dos patrones de creación que necesitareis para programar en Orientación a Objetos. La función **de los patrones de creación será la de crear objetos de clases**, pero acorde a problemas de orientación a objetos más complejos, donde el tradicional new, no es solución **porque rompe la estructura de nuestro programa.**

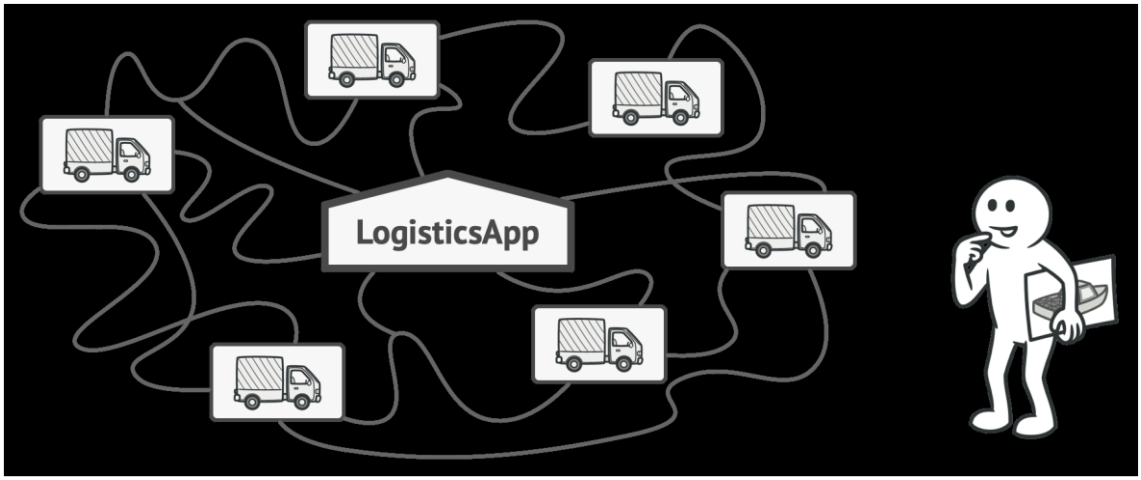
Una breve definición de la labor de un patrón de creación sería:

1. **Tratar con objetos que delegan responsabilidades a otros objetos**. Esto da como resultado n **una arquitectura en capas de componentes** con bajo grado de **acoplamiento**.
2. Facilitar **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**.
3. Proporcionar **formas de estructurar un objeto agregado** para que **se cree en su totalidad y recuperar los recursos** del sistema **de manera oportuna**.

Además, **admiten un mecanismo uniforme, simple y controlado** para crear objetos. **Permitir la encapsulación** de los **detalles** sobre qué clases se crean instancias y cómo se crean estas instancias. Fomentan el uso de interfaces, lo que reduce el acoplamiento. Comenzaremos con el **patrón Factory**.

4.1 Factory method o patrón factoria.

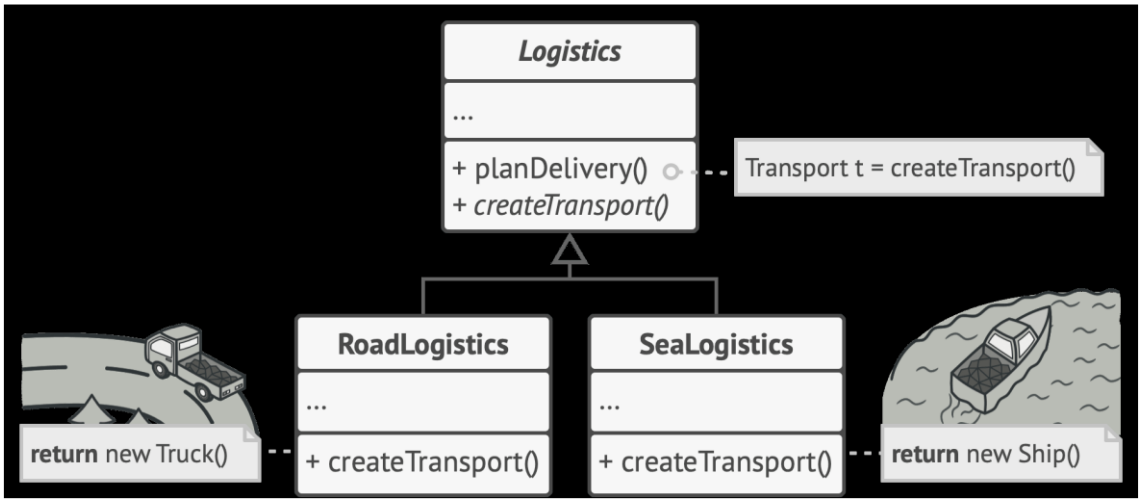
Factory Method es un patrón de diseño de creación que **proporciona una interfaz para crear objetos en una superclase**, pero **permite a las subclases modificar el tipo de objetos que se crearán**. Por ejemplo, tenemos una aplicación de logistica y realizamos envios en camiones. Tenemos la **Clase Truck**, pero ahora nuestro jefe quiere añadir barcos la ecuación, una **nueva Clase Ship**.



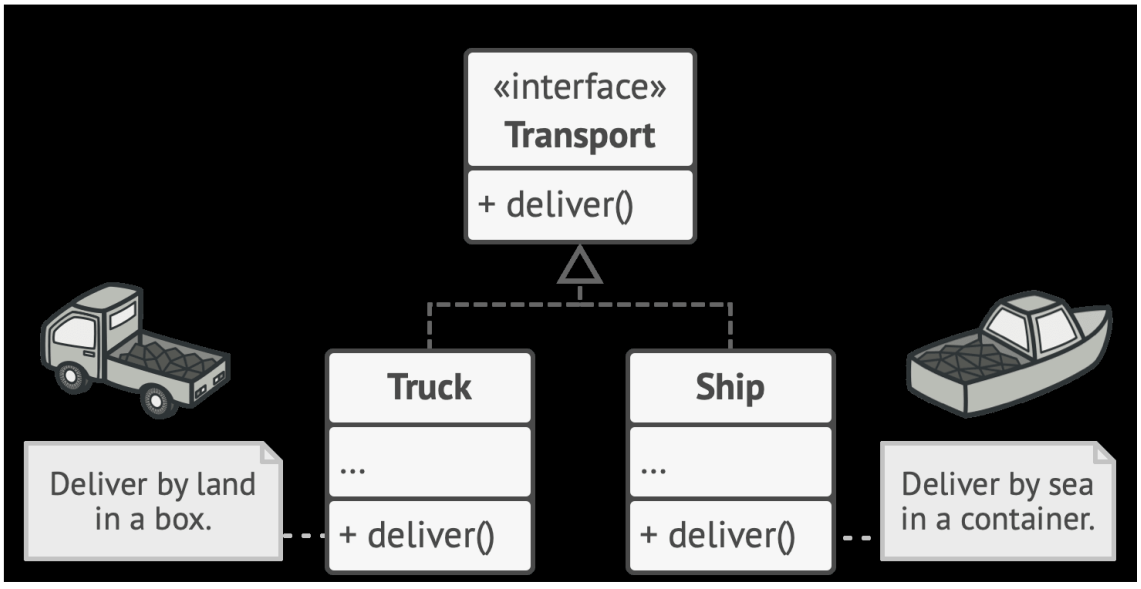
Si añadimos **la nueva clase Ship, vamos a tener que cambiar nuestro código completo** que sólo aceptaba camiones. Al final esto redunda en **un código lleno de ifs viendo** que **clase u objeto es el encargado** del reparto.

El **patrón factoría sugiere reemplazar llamadas de construcción de objetos** (utilizando el nuevo operador) con **llamadas a un método especial de la Factoría**. No te preocupes: los **objetos siguen creado a través del nuevo operador**, pero **está siendo llamado desde dentro el método Factory**. Los

objetos devueltos **por un método Factory** o se denominan **productos.**



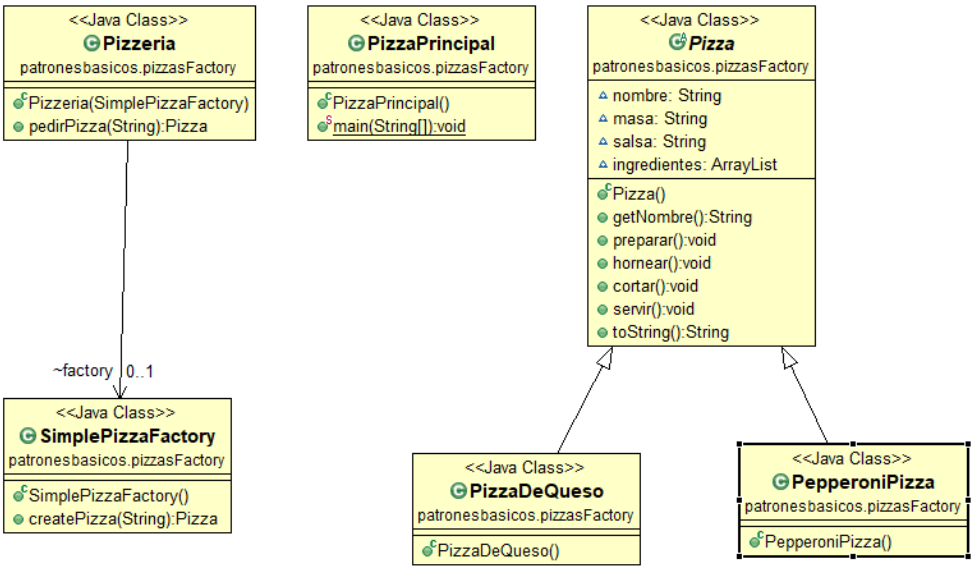
En **tiempo de ejecución se decide que objeto se construye**, un barco, un camión, **sin modificar apenas nuestro código**. En lugar de declarar una **variable Ship o Truck** vamos a declarar una variable de tipo **clase abstracta o interfaz** Transporte, que interiormente puede **ser una clase Ship o un clase Truck**.



Por ejemplo, **las clases Truck y Ship deben implementar la interfaz Transport**, que **declara un método llamado deliver()** (entregar). Cada clase implementa este método de manera diferente: Los **camiones entregan carga por tierra, los barcos entregan carga por mar.**
El **método de factoria de la clase RoadLogistics** devuelve **el objeto Truck**, mientras que **el método de factoria SeaLogistics clase devuelve Ship**.
El código que utiliza **el método factory** (a menudo llamado cliente código) **no ve una diferencia** entre **los productos reales devueltos por varias subclases**. El **cliente trata todos los productos** como **un transporte abstracto.**

4.1.1 Ejemplo 1. Pizzas. Practica guiada patrón factory

Factoria de pizzas. Vamos **a crear una factoria para realizar el trabajo de preparar, hornear, cortar y servir independientemente de la pizza que elijamos**, si es de queso o pepperoni. Este es el modelo del programa.
Creamos una factoria, **SimplePizzaFactory** para pizzas de distinto tipo. **PepperoniPizza** y de **PizzaDeQueso**. Tenemos una clase genérica Pizza, que define el comportamiento de las pizzas específicas. Desde **Pizzeria** usamos **la cocina, SimplePizzaFactory**. Y **PizzaPrincipal, nuestro programa y clase cliente** usa todo **nuestro modelo** o framework de clases para crear las pizzas, **a través de la Pizzeria**. Pizzeria es el **punto de entrada** a nuestro **modelo de datos**.



Empezamos **creando la clase abstracta pizza**, y **dos pizzas que heredan de ella.**

Pizza.java

```
import java.util.ArrayList;



abstract public class Pizza {
```

```
        String nombre;
        String masa;
        String salsa;
        ArrayList ingredientes = new ArrayList();

        public String getNombre() {
                return nombre;
        }

        public void preparar() {
                System.out.println("Preparar " + nombre);
        }

        public void hornear() {
                System.out.println("Hornear " + nombre);
        }

        public void cortar() {
                System.out.println("Cortar " + nombre);
        }

        public void servir() {
                System.out.println("Servir " + nombre);
        }

        public String toString() {

                StringBuffer display = new StringBuffer();
                display.append("---- " + nombre + " ----\n");
                display.append(masa + "\n");
                display.append(salsa + "\n");
                for (int i = 0; i < ingredientes.size(); i++) {
                        display.append((String )ingredientes.get(i) + "\n");
                }
                return display.toString();
        }
```

```
}
```

PizzaDeQueso.java

```
public class PizzaDeQueso extends Pizza {
        public PizzaDeQueso() {
                this.nombre = "Pizza de queso";
                this.masa = "Masa normal";
                this.salsa = "Salsa Marinara";
                this.ingredientes.add("Mozzarella");
                this.ingredientes.add("Parmesano");
        }
}
```

PepperoniPizza.java

```
public class PepperoniPizza extends Pizza {

        public PepperoniPizza() {
                this.nombre = "Pepperoni Pizza";
                this.masa = "crujiente";
                this.salsa = "Salsa marinada";
                ingredientes.add("Rodajas de pepperoni");
                ingredientes.add("Cebolla");
                ingredientes.add("Queso Parmesano gratinado");
        }
}
```

Como **veis usamos el patron padre abstracto** visto anteriormente. Las **pizzas heredan de la clase abstracta**. Hasta aquí **hemos hecho lo que antes**. Ahora viene **lo nuevo**

A nuestro programa principal **PizzaPrincipal.java** no le interesa saber que tipo de pizza es, **sólo hacer el trabajo de ordenar la pizza**. Ya tenemos a nuestro

cocinero, **SimplePizzaFactory preparando la pizza adecuada** en función del pedido. Esta clase es la **factoria que crea la pizza**, cuando llamamos al método **createPizza**(String c). Si el pedido **es de tipo "queso"** crearemos **una PizzaDeQueso**. Si el pedido es tipo **"pepperoni", creamos una PepperoniPizza**.

```
public Pizza createPizza(String tipo)
```

```
public class SimplePizzaFactory {

	public Pizza createPizza(String tipo) {
		Pizza pizza = null;

		if (tipo.equals("queso")) {
			pizza = new PizzaDeQueso();
		} else if (tipo.equals("pepperoni")) {
			pizza = new PepperoniPizza();
		}
		return pizza;
	}
}
```

En la **pizzeria atendemos las ordenes y se la mandamos al cocinero**, a la factoria. Hacemos el pedido de la pizza. **Creamos el** pedido que nos indica **en el parámetro tipo de la Pizza a través de la factoría.** Fijaos que la clase donde la almacenamos **es Pizza de tipo abstracto**. No sabemos qué tipo de pizza es ni nos importa. Es al cliente al que le importa. **La pizza se sirve en su caja** para llevar y no la vemos. Fijaos que tenemos 4 comportamientos para crear la pizza. Los métodos **preparar, hornear, cortar y servir**. Hay que hacer lo mismo con las dos pizzas. Con la de pepperoni y la de queso. Por eso **tenemos el código en la clase abstracta**. No **hace falta duplicarlo en las clases hijas**. Las **clases hijas lo heredan**.

```
public class Pizzeria {
	SimplePizzaFactory factory;

	public Pizzeria(SimplePizzaFactory factory) {
		this.factory = factory;
	}
```

```
        public Pizza pedirPizza(String tipo) {
        Pizza pizza;
         pizza = factory.createPizza(tipo);

                pizza.preparar();
                pizza.hornear();
                pizza.cortar();
                pizza.servir();

                return pizza;
        }

}
```

Y en el **programa principal lanzamos los pedidos**. Usamos la factoria SimplePizzaFactory y se la pasamos a la Pizzeria. Estamos indicando a la Pizzeria que tipo de factoria vamos a usar. Y en el pedido, indicamos el tipo que queremos. Estamos guardándolo **todo en una clase abstracta Pizza, porque a la pizzeria** le da igual que pizza pedir. Es el **cocinero, la factoria SimplePizzaFactory** quien hace el trabajo **y responde al pedido cocinando la pizza adecuada**. Poniendo **los ingredientes correctos** en función de tipo de pizza.

```
SimplePizzaFactory factory = new SimplePizzaFactory();
                Pizzeria pizzeria = new Pizzeria(factory);
```

PizzaPrincipal.java

```
public class PizzaPrincipal {


        public static void main(String[] args) {
                SimplePizzaFactory factory = new SimplePizzaFactory();
                Pizzeria pizzeria = new Pizzeria(factory);

                Pizza pizza = pizzeria.pedirPizza("queso");
                System.out.println("Hemos pedido una pizza" + pizza.getNombre
() + "\n");
```

```
                pizza = pizzeria.pedirPizza("pepperoni");
                System.out.println("Hemos pedido una pizza " + pizza.getNombr
e() + "\n");
        }
}
```

4.1.2 Ejercicios. Practica independiente patron factory.

1. Crear **dos nuevas pizzas PizzaFrutiDiMare, y PizzaBarbacoa**. Elegid vosotros los ingredientes.

2. **Cread un nuevo tipo de factoria llamada Kebab**. En esta se prepara, cocina y sirve **Durum Kebab y Galet Kebab que heredarán de la clase abstracta Kebab**

Kebab

- **tostarPan**
- **asarCarne**
- **anadirSalsas**
- **servir**

3. . Añadir **una nueva clase TiendaKebab** y **lanzar pedidos** desde la clase principal **PizzaPrincipal**.

4.2 Patrón Singleton

Este patrón nos va a **permitir crear objetos de una clase pero de una manera especial**. Sólo **va a permitir crear una sólo instancia u objeto de la clase objetivo**. Es muy utilizado para por ejemplo crear sesiones de usuario en las aplicaciones.

El patrón **Singleton** **es un patrón de diseño fácil de entender**. A veces, puede surgir la necesidad de **tener una sola instancia de una clase dada** durante **la vida útil de una aplicación**. Esto puede deberse a la necesidad o, más a menudo, a el hecho de que sólo una instancia de la clase es suficiente o necesaria. Por ejemplo, **podemos necesitar un único objeto de conexión** de **base de datos** en una aplicación. El **patrón Singleton es útil en estos casos porque asegura** que **existe uno y sólo una instancia de un objeto** en particular. Además, sugiere que los **objetos cliente ser capaz de acceder a la instancia única de una manera coherente**.

Tener una instancia de la clase en una variable global parece **una manera fácil** de mantener la instancia única. Todos los objetos de cliente pueden acceder a esta instancia de un manera consistente o coherente a través de esta variable global. **Pero esto no impide que los clientes desde la creación de otras instancias de la clase**. Para que este enfoque tenga éxito, todos los de los **objetos cliente tienen que ser responsables** de **controlar el número de instancias** de la clase. Esta responsabilidad ampliamente distribuida no es deseable porque **un cliente debe estar libre de cualquier detalle del proceso de creación de clase**. La **responsabilidad de asegurárse de que sólo hay una instancia de la clase** debe **pertenecer a la clase en** sí, dejando a los objetos de cliente libres de tener que controlar estos detalles.
Una clase que mantiene su naturaleza de instancia única por sí misma **se conoce como clase Singleton.**

La **implementación clásica de Singleton es la siguiente**. Se **crea un atributo estático de clase en donde se guarda el primer objeto** que **se crea de la clase se devuelve el objeto ya creado**, **si ha sido creado**, porque lo tenemos en la variable instanciaUnica.

```
if (instanciaUnica == null) {
                        instanciaUnica = new Singleton();
                }
                return instanciaUnica;
```

No podemos hacer un new del objeto. Sino **daríamos control al resto de usuarios**, para que lo hagan en otro sitio. Para que **esto sea posible hemos hecho que el constructor de la clase sea privado**, de manera que no podemos hacer un new desde fuera de la clase
`private Singleton() {}`

singleton.java

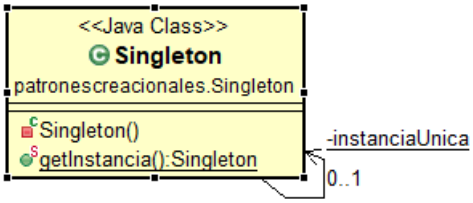
```
public class Singleton {
        private static Singleton instanciaUnica;



        private Singleton() {}


        public static Singleton getInstancia() {
                if (instanciaUnica == null) {
                        instanciaUnica = new Singleton();
                }
```

```
                return instanciaUnica;
        }


}
```

Es el método getInstancia() el que nos va a dar una **única instancia del objeto**. Para crear **al objeto Singleton debemos usar la clase estáticamente** y **llamar al método estático,** getInstancia() **Probad que no se puede hacer un new del objeto**. llamándolo desde una clase principal. Lo podéis ver en el siguiente ejemplo CocinaChocolate.

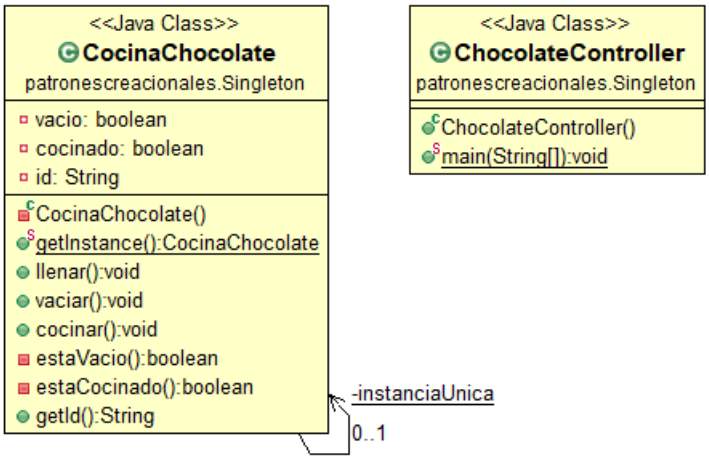
Para **probar crear el objeto Singleton** de **la siguiente manera**.

Singleton objeto = Singleton.getInstancia();



4.2.1 Ejemplo CocinaChocolate. Practica guiada patron singleton

Vamos a crear una herramienta para cocinar chocolate. **Cuando este vacia se llena, se cocina el chocolate y se vacía.** Queremos que la cocina sea única y la vamos a implementar con el patrón Singleton. Podéis ver el modelo de clases en el siguiente gráfico.



Será la clase que **implemente el patron singleton**, si os dais cuenta es igual al anterior. Usamos una **clase chocolateController** para usar este **objeto singleton**.

La **clase CocinaChocolate ofrece tres métodos**, tres servicios para cocinar el chocolate. **Llenar, cocinar y vaciar**. Es el **proceso completo** para **fabricar chocolate.**

Muy importante "public **static** CocinaChocolate **getInstance()**" es un método estático se usa a la clase y no a un objeto para llamarlo. Fijaos que **tiene el modificador static** delante.

CocinaChocolate.java

```
public class CocinaChocolate {
	private boolean vacio;
	private boolean cocinado;
	private String id="Soy el objeto unico. Id 1";
	private static CocinaChocolate instanciaUnica;

	private CocinaChocolate() {
		vacio = true;
		cocinado = false;
	}

	public static CocinaChocolate getInstance() {
		if (instanciaUnica == null) {
```

```
                        System.out.println("Creando unica instancia de choco
late");
                        instanciaUnica = new CocinaChocolate();
                }
                System.out.println("Devolviendo la instancia de chocolate");
                return instanciaUnica;
        }

        public void llenar() {
                if (estaVacio()) {
                        vacio = false;
                        cocinado = false;
                        // fill the boiler with a milk/chocolate mixture
                }
        }

        public void vaciar() {
                if (!estaVacio() && estaCocinado()) {
                        // drain the boiled milk and chocolate
                        vacio = true;
                }
        }

        public void cocinar() {
                if (!estaVacio() && !estaCocinado()) {
                        // bring the contents to a boil
                        cocinado = true;
                }
        }

        private boolean estaVacio() {
                return vacio;
        }

        private boolean estaCocinado () {
                return cocinado;
        }
```

```
        public String getId() {

                return this.id;
        }
}
```

ChocolateController.java

ChocolateController, es la clase **cliente que usa los servicios de la clase Singleton** para ir **fabricando chocolate** en diferentes pasos, ***llenar*, *cocinar y vaciar***.

Como veis aunque usar dos variables para crear el objeto CocinaChocalate el objeto siempre es el mismo.

Otra **vez fijaos en que estoy llamando a un método estático**, de la clase **CocinaChocolate**. Los **métodos estáticos o de clase**, están explicados en el tema anterior. Son **métodos que no se llaman sobre un objeto creado**, se **llaman sobre una clase. Tienen el modificador static,** para indicar que son métodos de clase.

```
public class ChocolateController {
        public static void main(String args[]) {
                CocinaChocolate cocinaChoc = CocinaChocolate.getInstance();
                cocinaChoc.llenar();
                cocinaChoc.cocinar();
                cocinaChoc.vaciar();

                System.out.println(" Soy siempre el mismo objeto " + cocinaCh
oc.getInstance());

                // will return the existing instance
                CocinaChocolate cocinaChoc2 = CocinaChocolate.getInstance();

                System.out.println(" Soy siempre el mismo objeto " + cocinaCh
oc2.getInstance());



                // no se puede  hacer un new
```

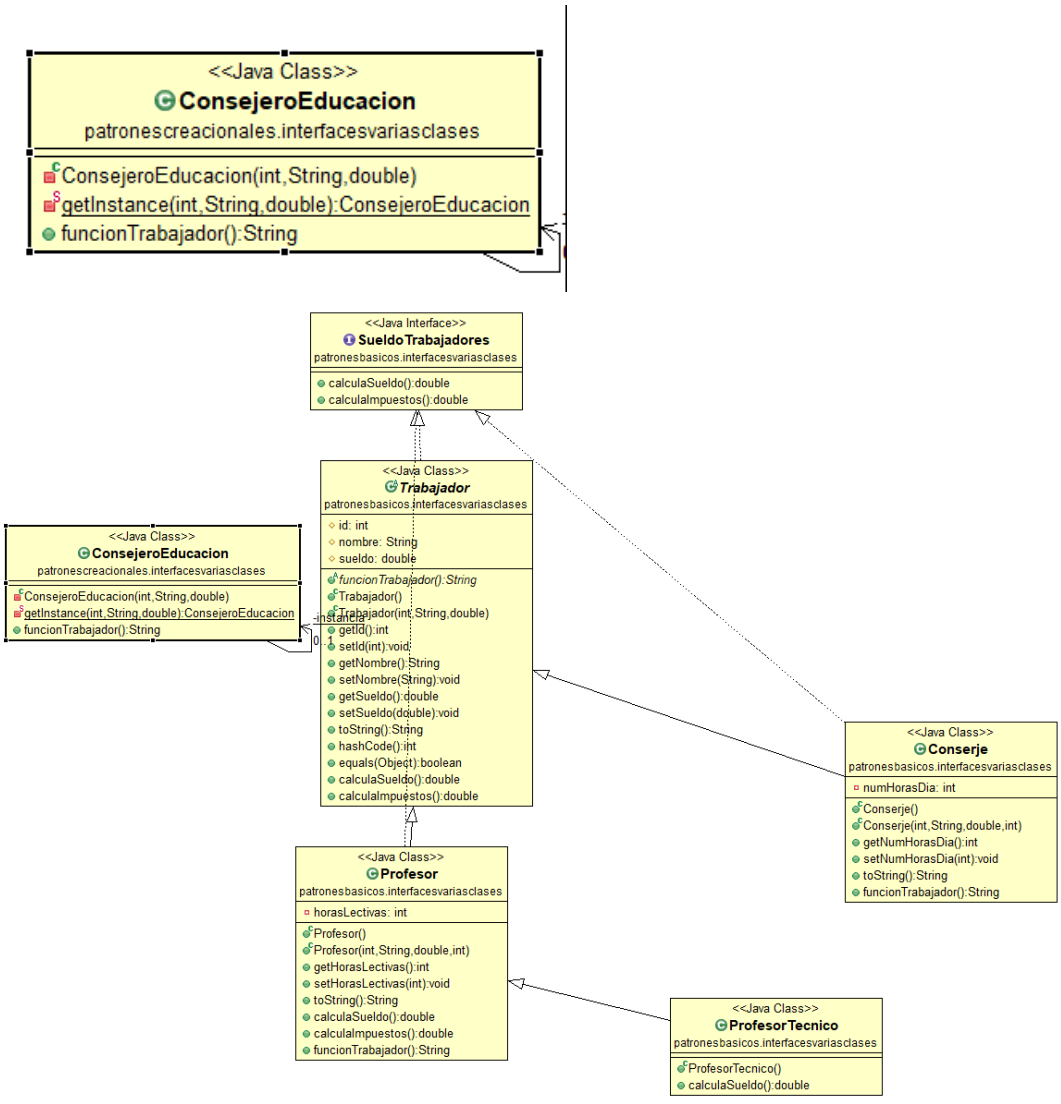
```
                //CocinaChocolate cocinaChoc2 = new CocinaCocholate();
        }
}
```

Esta comentado el código para hacer un new del objeto. Comprobar que no se puede.

4.2.2 Practica independiente patron Singleton.

Sobre el modelo de trabajadores de educación añadimos la clase ConsejeroDeEducacion.java.

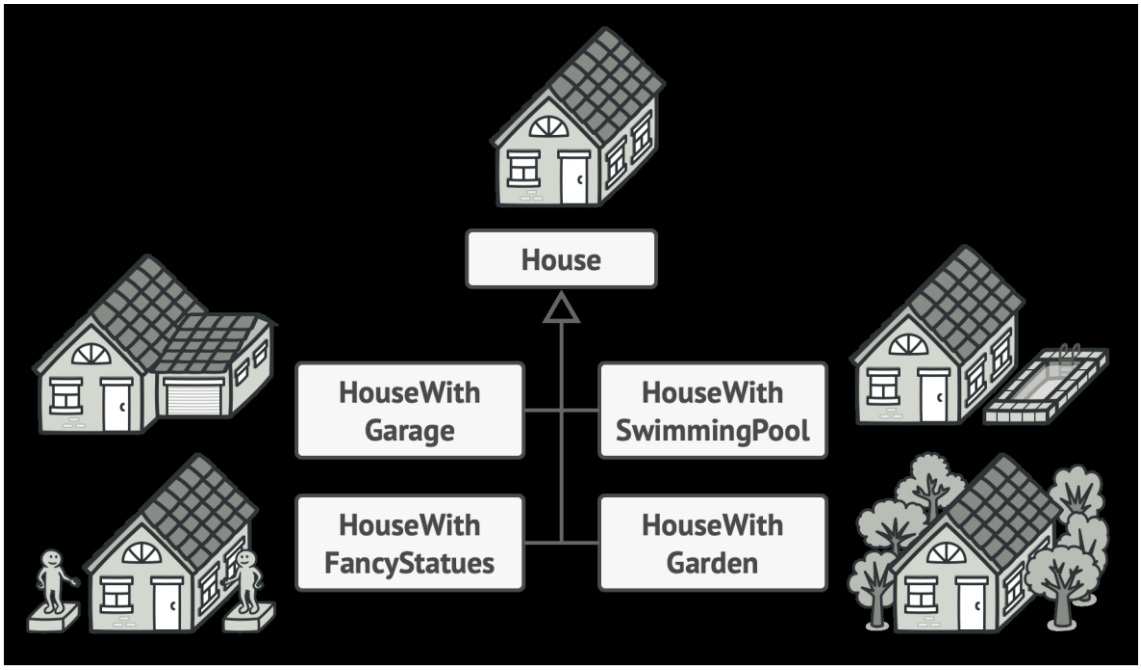
1. Heredara de la clase Trabajador
2. Se creará con un patrón Singleton como hemos visto en el ejemplo anterior. Porque consejero de educación solo puede haber uno.



4.3 Patron Builder

Builder **es un patrón de diseño de creación** que te **permite construir objetos complejos paso a paso**. El patrón **le permite producir diferentes tipos y representaciones de un objeto** utilizando **el mismo código de construcción.**

El problema es:
Imagina un **objeto complejo que requiera arduos pasos** de inicialización de muchos **campos y objetos anidados.** Dicha inicialización de código **es generalmente encerrado dentro de un constructor de monstruoso** tamaño con muchos parámetros. O peor aún: dispersos por todo el código de cliente.



Por ejemplo, **pensemos en cómo crear un objeto House**. Para **construir una casa sencilla, es necesario construir cuatro muros** y un **instalar una puerta, encajar un par de ventanas y construir un techo**. Pero ¿qué pasa **si quieres una casa más grande, más brillante**, con un **patio trasero y otras mercancías** (como un sistema de calefacción, fontanería y cableado)?

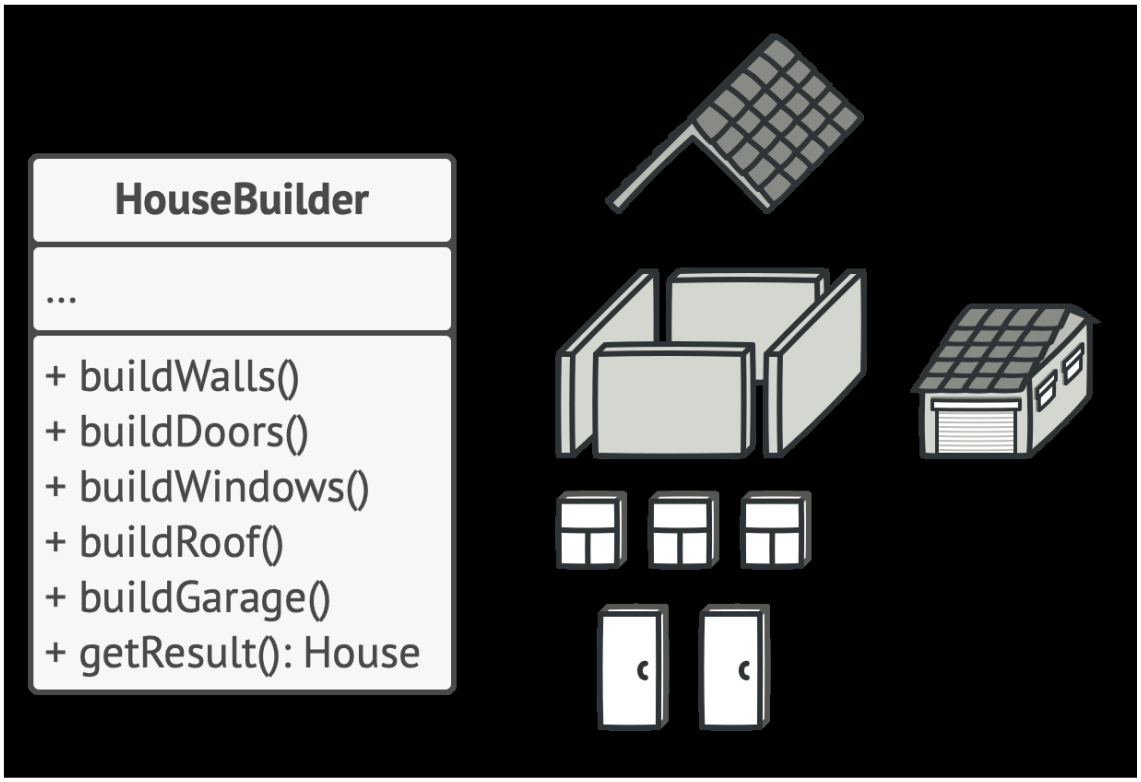
La **solución más sencilla es ampliar la clase base House** y crear **un conjunto de subclases para cubrir todas las combinaciones de la Parámetros**. Pero con el tiempo terminarás **con un considerable número de subclases**. Cualquier parámetro nuevo, **como el porche estilo, requerirá el crecimiento de esta jerarquía aún más**. Hay otro enfoque que no implica el uso extensivo de

subclases. Puedes **crear un constructor gigante justo en la base Clase House** con **todos los parámetros posibles que controlan el objeto House**. Si bien este enfoque **elimina de hecho la necesidad de subclases, crea otro problema.**

En la mayoría de los casos, **la mayoría de los parámetros no se utilizarán**, el **constructor al que se llama** queda bastante **feo**. Por ejemplo, sólo una fracción de las casas tienen piscinas, por lo que los parámetros relacionados con piscinas serán inútiles nueve veces de cada diez.

Solución

El patrón **Builder sugiere que extraigas la construcción del objeto código fuera** de su **propia clase** y moverlo a separar objetos llamados Builders.



El patrón organiza la construcción de objetos en un conjunto de pasos (buildWalls , buildDoor , etc.). Para crear un objeto, **se ejecuta una serie de estos pasos en un objeto de constructor**. Lo importante parte es que **no es necesario llamar a todos los pasos**. Puedes llamar **sólo los pasos que son necesarios** para producir una configuración de un objeto.

Algunos de los pasos de construcción podrían **requerir una implementación** diferente cuando se necesita **construir varias representaciones del producto**. Por ejemplo, las paredes de **una cabaña pueden madera, pero las paredes del castillo deben** ser construidas con piedra. En este caso, puede **crear varias clases de Builders diferentes que implementan el mismo conjunto de pasos** de construcción, pero de manera **diferente**. Entonces se puede **utilizar estos**

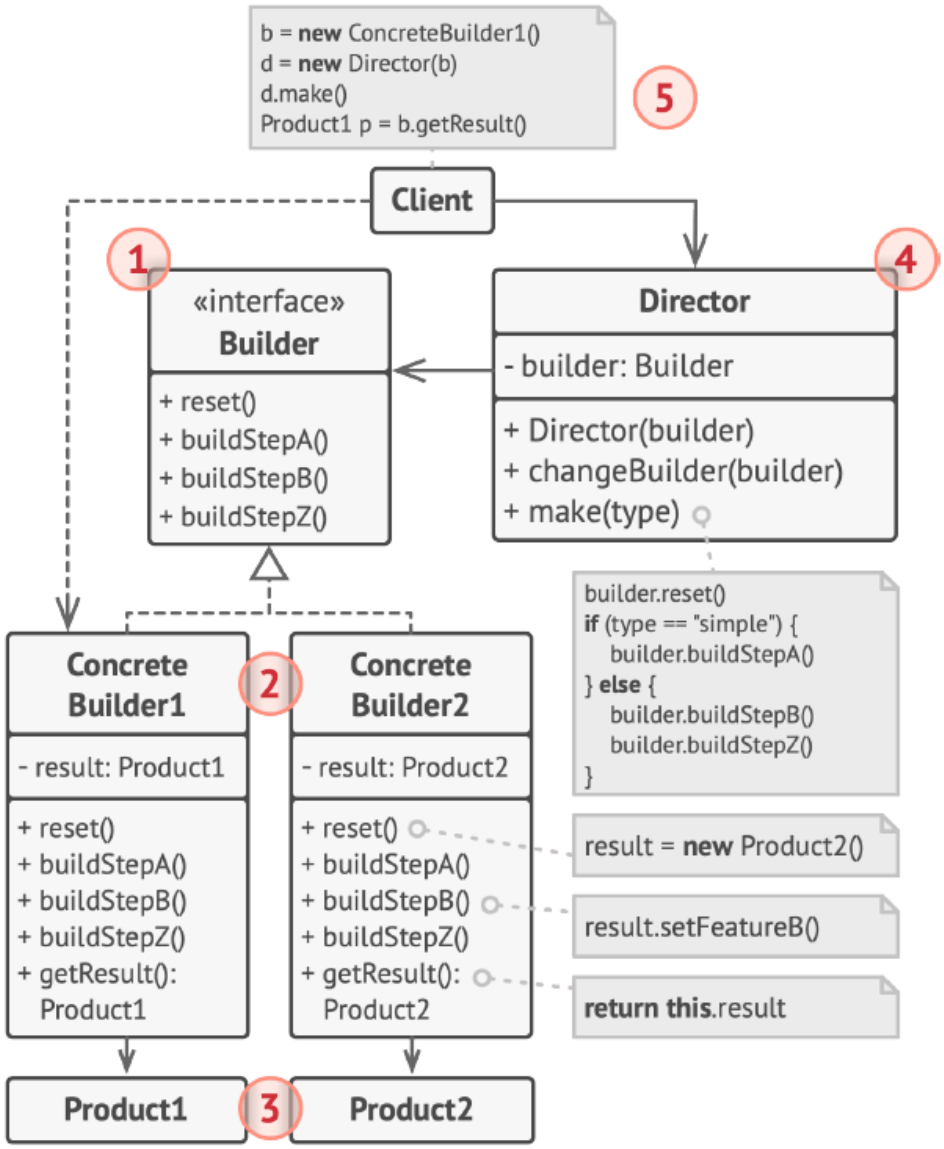
constructores en el proceso de construcción (es decir, un conjunto ordenado de llamadas a los pasos de construcción) para producir diferentes tipos de objetos.

Por ejemplo, imagina un constructor **que construya todo a partir de madera y vidrio**, **una segunda que construye todo con piedra y hierro y** una tercera que **utiliza oro y diamantes.** Llamando al mismo conjunto de pasos, se obtiene una casa normal del primer constructor, un pequeño castillo del segundo y un palacio de el tercero.

. **Director**

Puedes ir más allá y **extraer una serie de llamadas al constructor**, pasos que se utilizan para construir un producto, **en una clase separada llamado director**. La clase director **define el orden en que para ejecutar los pasos de construcción**, mientras que el constructor proporciona la implementación de esos pasos.

Además, **la clase director oculta por completo los detalles** de **la construcción del producto a partir del código del cliente**. El cliente **solamente necesita asociar a un constructor con un director**, lanzar **la construcción con el director**, y **obtener el resultado del constructor**.

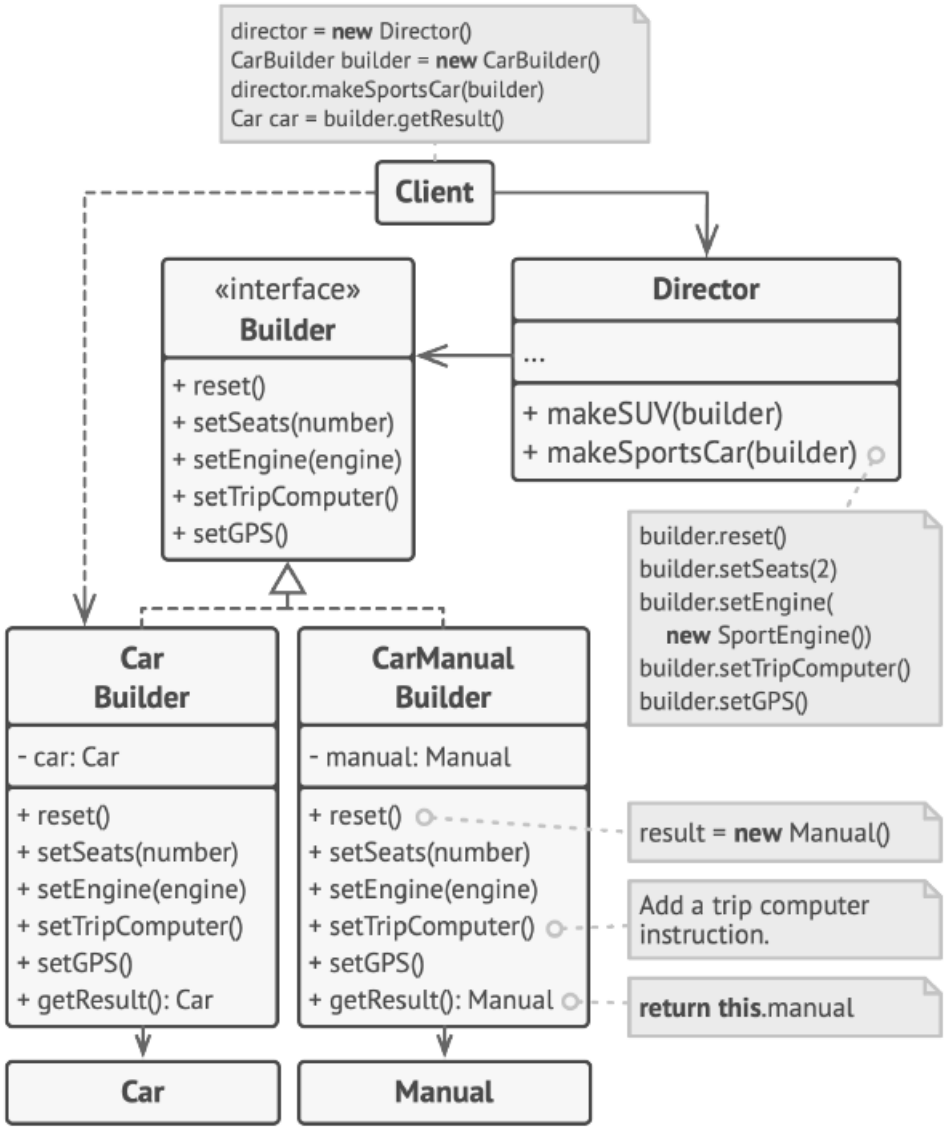


1. La **interfaz Builder** declara los pasos de construcción del producto que son comunes a todo tipo de constructores.
2. Los **constructores concretos** (**concrete builders**) proporcionan **diferentes implementaciones de la pasos de construcción**. Los constructores de hormigón pueden producir productos que no siguen la interfaz común. 3.
3. Los **productos (products) son objetos resultantes**, **productos construidos por diferentes constructores no tienen que pertenecer a la misma jerarquía** de clases o interfaz.
4. La **clase Director define el orden en que se debe llamar a la construcción pasos**, para que pueda **crear y reutilizar configuraciones específicas** de productos.
5. **El Cliente** (Client) **debe asociar uno de los objetos del constructor con el director**. Por lo general, **se hace una sola vez, a través de parámetros de constructor del director**. Entonces **el director usa ese constructor objeto para toda la construcción posterior.** Sin embargo,

hay una alternativa enfoque para cuando el cliente pasa el objeto constructor a el método de producción del director. En este caso, puede utilizar un constructor diferente cada vez que se produce algo con el Director.

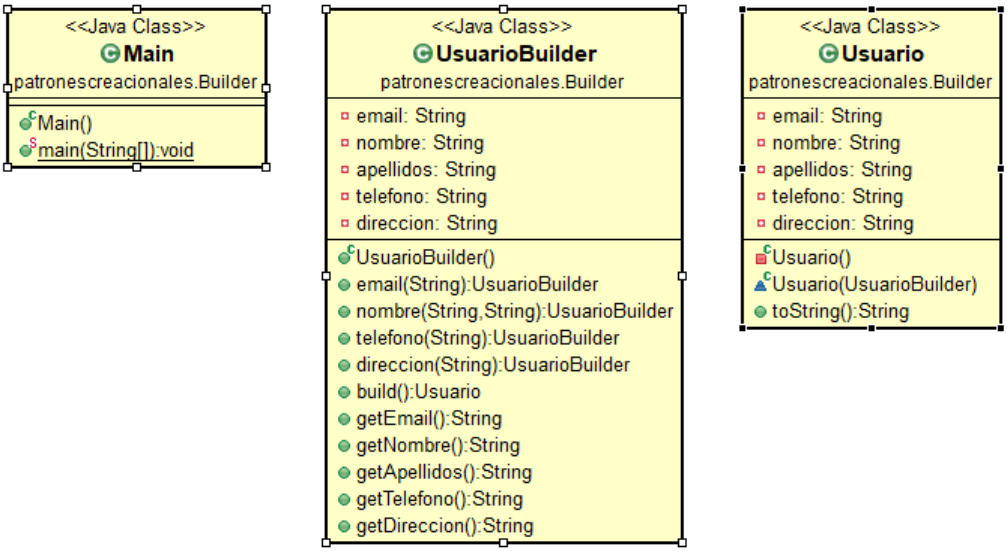
Seudocódigo

Este ejemplo **del patrón Builder ilustra cómo puede reutilizar el mismo código** de **construcción de objetos al construir diferentes tipos de productos**, como los **automóviles, y crear** los correspondientes **manuales para ellos.**



4.3.1 Ejemplo Usuario. Practica guiada patrón Builder

Vamos a **montar un ejemplo de builder para un usuario** con **muchos parámetros** y que pueda ser **construido poco a poco**, con el patrón builder. Con el siguiente diagrama de clases:



Fijaos en el constructor, es la clave para crear el Usuario con el Objeto Builder. **Definimos en Usuario un constructor privado**, y un **constructor que recibe como parámetro el objeto Builder**. De esta manera **es sólo posible crear el objeto Usuario** a **través del objeto Builder.**

```
Usuario(UsuarioBuilder builder) {
        if (builder.getEmail() == null) {
            throw new IllegalArgumentException("email es requerido");
        }
        this.email = builder.getEmail();
        this.nombre = builder.getNombre();
        this.apellidos = builder.getApellidos();
        this.telefono = builder.getTelefono();
        this.direccion = builder.getDireccion();
    }
```

Obligamos además al constructor Builder a que nos pase el email con **:**

```
if (builder.getEmail() == null) {
```

Mirad el **constructor de la clase Usuario**, nos obligar **a pasar un objeto Builder para ser construido. No se puede crear el objeto Usuario**, si **no hay un objeto**

Builder de por medio.

Clase Usuario.java

```
public class Usuario {

    private String email;
    private String nombre;
    private String apellidos;
    private String telefono;
    private String direccion;

    private Usuario() {
    }
  //Constructor especial
    Usuario(UsuarioBuilder builder) {
        if (builder.getEmail() == null) {
            throw new IllegalArgumentException("email es requerido");
        }
        this.email = builder.getEmail();
        this.nombre = builder.getNombre();
        this.apellidos = builder.getApellidos();
        this.telefono = builder.getTelefono();
        this.direccion = builder.getDireccion();
    }
public String toString() {

         return "usuario {email=" + this.email  +
        ",nombre= " + this.nombre +
        ",apellidos=" + this.apellidos +
        ",telefono=" + this.telefono +
        ",direccion ="+  this.direccion +"}";

    }

}
```

La **parte importante de la clase Builder es el build que nos construye al usuario**. El resto de **métodos nos rellenan parámetros del Usuario**. Recordar que el correo electrónico es necesario.

```
public Usuario build() {

        return new Usuario(this);

    }
```

En la clase UsuarioBuilder ofrecemos los **métodos email, nombre, dirección, teléfono,** para rellenar estos valores **antes de que el Builder** cree al usuario **con build.**

Build es el método que usamos para crear el objeto Usuario. Lo podemos hacer así por la definición anterior que tenemos del constructor Usuario, que tiene un objeto Builder como parámetro. Sólo podemos hacer un new del objeto desde Builder.

```
    public Usuario build() {
        return new Usuario(this);
    }
```

Clase UsuarioBuilder.java

```
public class UsuarioBuilder {


    private String email;
    private String nombre;
    private String apellidos;
    private String telefono;
    private String direccion;

    public UsuarioBuilder() {
    }

    public UsuarioBuilder email(String email) {
```

```
        this.email = email;
        return this;
    }

    public UsuarioBuilder nombre(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        return this;
    }

    public UsuarioBuilder telefono(String telefono) {
        this.telefono = telefono;
        return this;
    }

    public UsuarioBuilder direccion(String direccion) {
        this.direccion = direccion;
        return this;
    }

    public Usuario build() {
        return new Usuario(this);
    }

    // Getters
    public String getEmail() {
        return email;
    };

    public String getNombre() {
        return nombre;
    };

    public String getApellidos() {
        return apellidos;
    };
```

```
    public String getTelefono() {
        return telefono;
    };

    public String getDireccion() {
        return direccion;
    };
}
```

Para que la **construcción del usuario con el build sea efectiva** debemos asegurarnos que **pasamos un email**. Además **antes de llamar al build() llamamos a otros métodos anteriores** para **rellenar el objeto con los parámetros** que queremos

Clase **Main.java**

```
public class Main {


  public static void main(String[] args) {
    Usuario usuario = new UsuarioBuilder()
        .email("nombre.apellido@gmail.com")
        .nombre("Nombre", "Apellido")
        .telefono("555123456")
        .direccion("c\\ Rue el Percebe 13").build();

            Usuario usuario2 = new UsuarioBuilder()
                    .email("nombre2.apellido2@gmail.com")
                    .nombre("Nombre", "Apellido")
                    .telefono("555123456").build();



          System.out.println("usuario1" + usuario.toString());
          System.out.println("usuario2" +usuario2.toString());
  }
}
```

Hemos **creado dos usuarios diferentes**, **uno tiene dirección y otro no**. Esa es la clave del **Builder construimos sobre necesidad, y añadimos lo que nos hace falta** para cada objeto. Lo obligatorio, el correo, siempre debe estar.

4.3.2 Ejercicios. Practica independiente patron builder

1. **Añadir a usuario nombre como obligatorio**, y añadir el **atributo password al** ejemplo anterior. Que **password sea obligatorio también**.

2. Vamos a construir una estructura como la anterior con las siguientes clases. **Aplicando el patron Builder**, construir:

    a. Crear **una clase Autor** con **Nombre, Apellidos, Telefono, Direccion, Materia y Numero** de libros publicados.

    b. Crear todos los **getter y los setter** para los atributos. Añadir un método **toString**.

    c. Construir **un Builder** para construir **Autores, AutorBuilder**. Haced que **nombre de Autor sea obligatorio** en la construcción. El método que construye se llama **build**.

    d. Probadlos en una **clase Principal, AutoresPrincipal**, creando **dos autores**. Uno si teléfono, el otro sin dirección.

    e. Tipos de autor. Refactorizar el builder con tipos y herencia

        i. Normal
        ii. BestSeller -> numVentas
        iii. Premiado -> numVentas, numPremios

4.4 Patron Abstract Factory

Durante la discusión del patrón Factory vimos que en el contexto de **un Factory method o método Factory**, **existe una jerarquía de clases compuesta** de un conjunto de subclases (Pizza Queso, y Pepperoni Pizza) con una clase primaria común, en nuestro ejemplo, Pizza.

Un **Factory method** **se utiliza cuando un objeto de cliente** sabe cuándo **crear un instancia del tipo de clase primaria**, pero no sabe (o no debe saber) **exactamente de qué clase de entre el conjunto de subclases** (posiblemente la clase primaria) debe ser instanciada. Además de los **criterios de selección de clase**, un **Factory Method** también **oculta cualquier mecanismo especial**

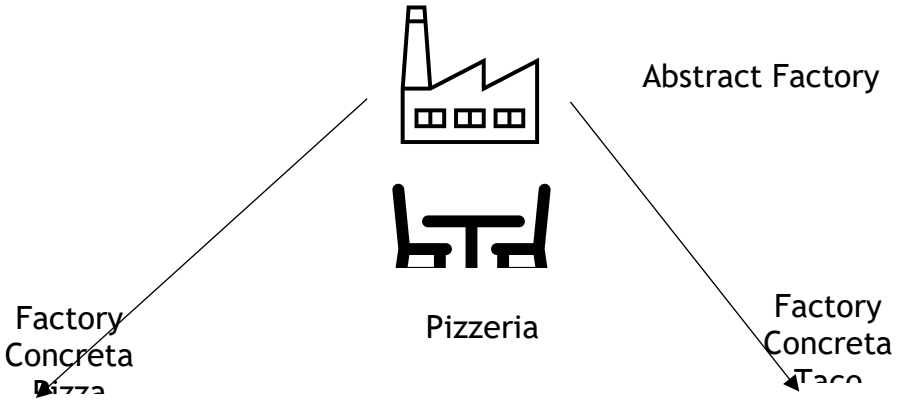
necesario para **crear instancias de la clase seleccionada**, por **ejemplo, que ingredientes lleva la pizza** de queso.

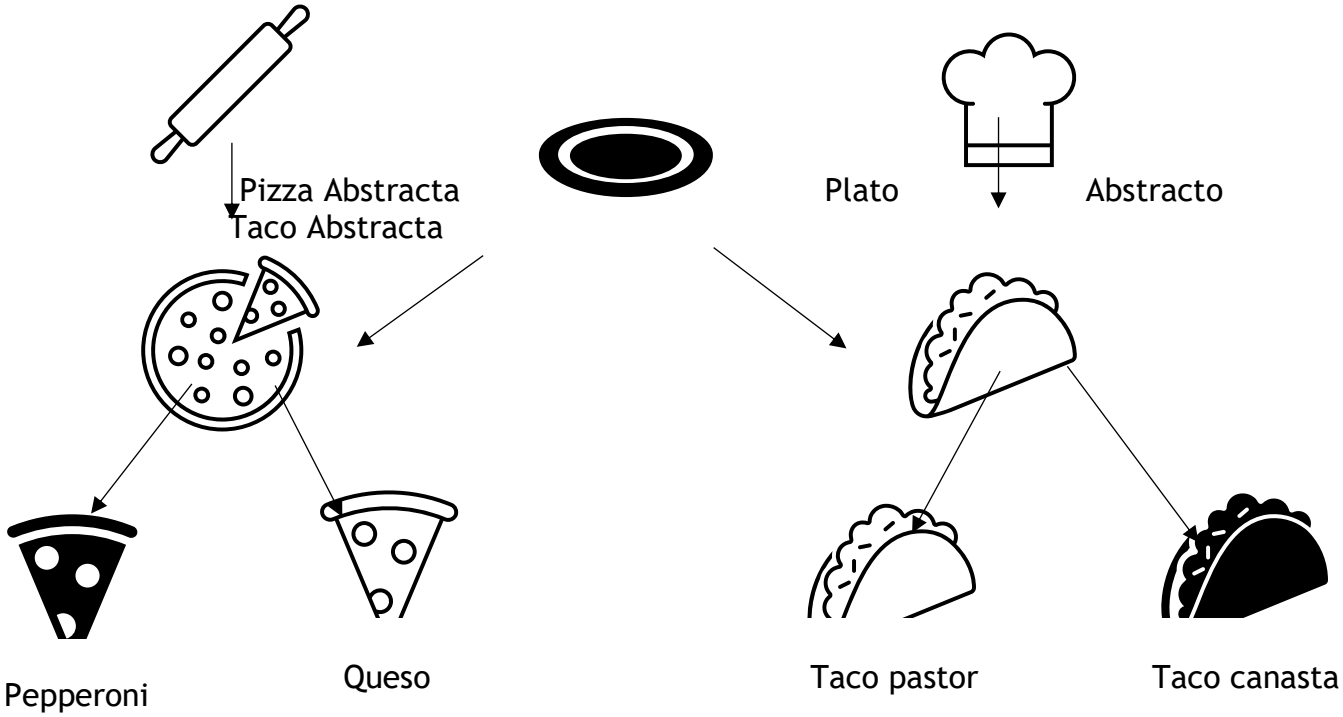
El **patrón Abstract Factory toma el mismo concepto al siguiente nivel**. En términos simples, **una fábrica abstracta es una clase que proporciona una interfaz** para producir una **familia de objetos,** platos de restaurante. En el **ejemplo de Factory Method** habíamos **creado una factoria para Pizzas**. Ahora vamos a usar una factoría extra, **vamos a añadir tacos a como platos** a nuestra **pizzeria**. Necesitamos una **Factoria para tacos**, y **una factoría abstracta que rija el comportamiento** de las factorías concretas **SimpleFactoryPizza,** y la nueva SimpleFactoryTacos.

En el **lenguaje de programación Java, se puede implementar ya sea como una interfaz** o como **una clase abstracta**. En otras palabras estamos **añadiendo variedad, en las factorías, tenemos una nueva,** la Factoria Taco, **para crear esta nueva familia de productos**. Usaremos un interfaz para crear la Factoria Abstracta

En el **contexto de una Abstract Factory existe: Suites o familias de clases dependientes relacionadas**. Un grupo de clases **de fábrica de platos que implementa la interfaz proporcionada** por la **clase Abstract Factory**. Cada una de estas fábricas controla **o proporciona acceso a un conjunto particular de objetos** e implementaciones relacionados **y dependientes la interfaz abstracta** de **la fábrica de una manera** que es **específica de la familia de clases que controla**.

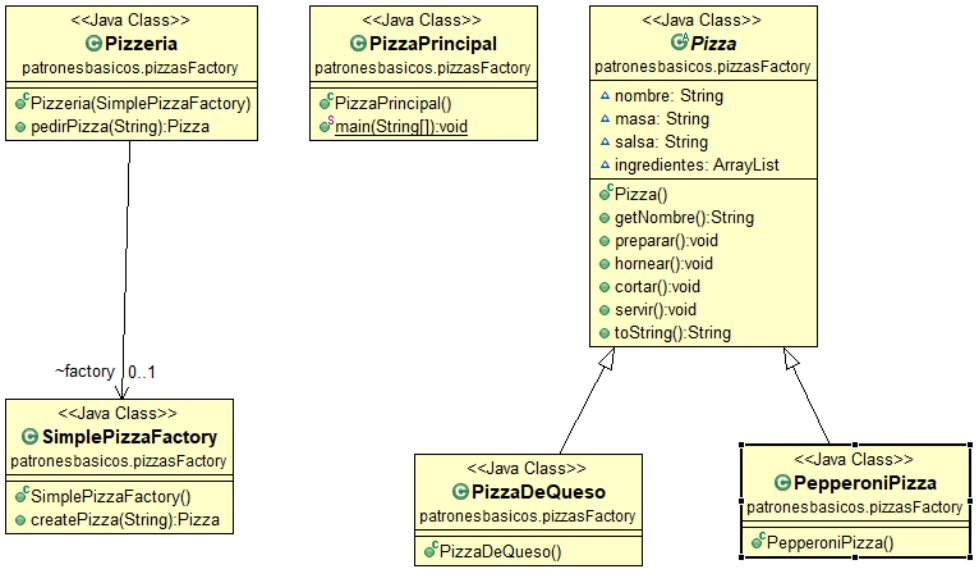
Diferentes cocinas de alimentos implementan **este Interfaz**. Los **objetos cliente, nuestro programa principal hacen uso de estas cocinas de pizzeria** para **crear objetos de dos familias diferentes: pizzas y tacos.** Por lo tanto, **no es necesario saber de qué clase concreta** se crea **realmente una instancia. Todas las instancias son Platos** para nuestro **programa principal y para la Pizzeria**. Lo podéis ver en el siguiente gráfico.





4.4.1 Ejemplo. Practica guiada de patrón abstract factory

Partimos de un modelo anterior con una factoria simple



Vamos a transformar este modelo Factory method con **una sola factoria**, en un **modelo AbstractFactory con varias factorías** y una **factoria abstracta que define** el **modelo de todas las factorías.**

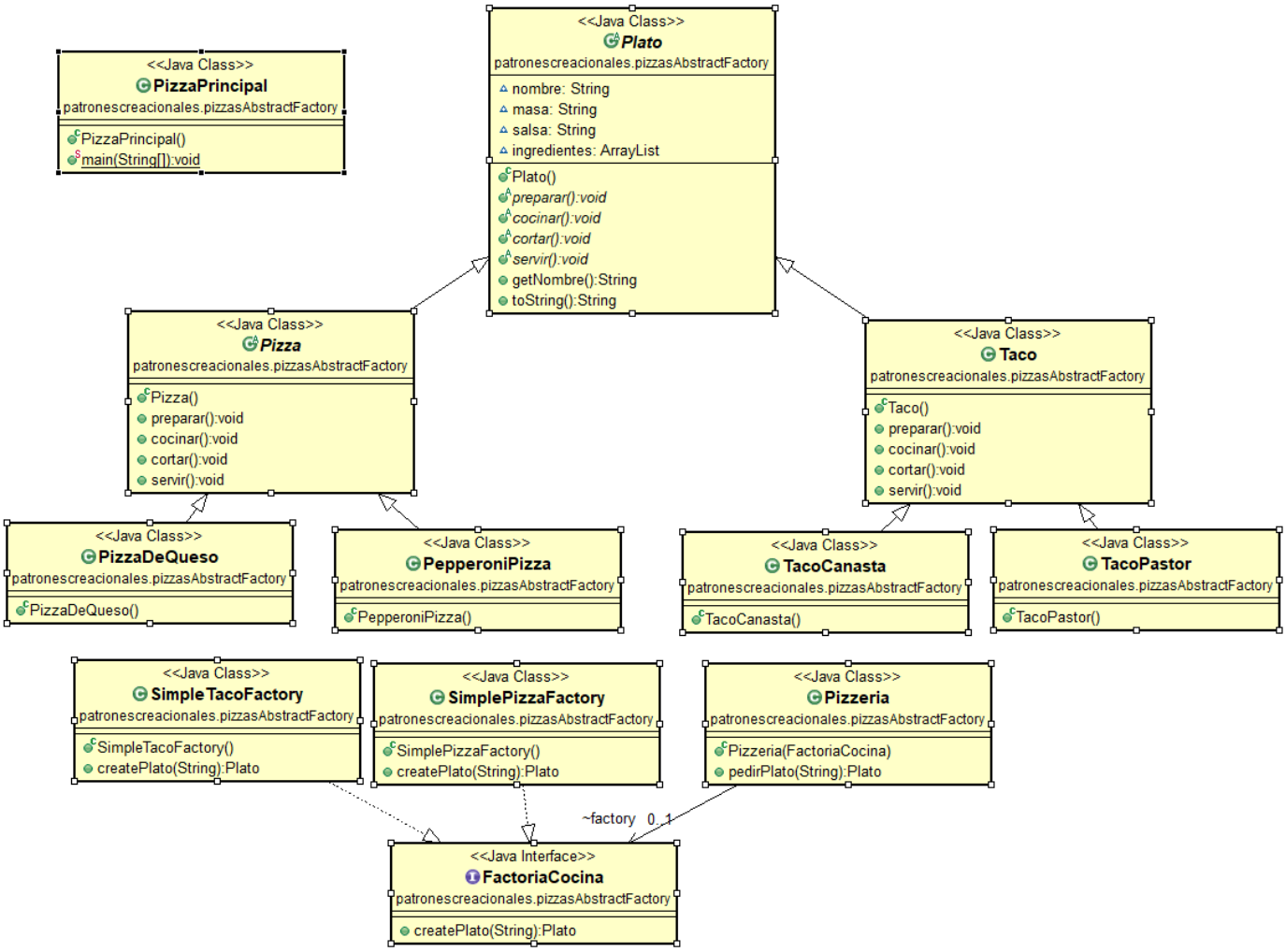
Introducimos **la SimpleTacoFactory como factoria concreta nueva.** Ahora tenemos **dos factorias, SimplePizzaFactory y SimpleTacoFactory.** La que define el **comportamiento de ambas** es la **nueva factoria abstracta**

FactoriaCocina. Lo podéis ver en el gráfico de la página siguiente. **FactoriaCocina es usado por Pizzeria, independientemente del plato** que **prepare** para **crear los platos**. **Dependiendo** del **pedido** se usará **una factoria concreta u otra para crear el plato**.

Igualmente hemos añadido **la clase abstracta Plato.** Ahora tenemos más variedad necesitamos recogerla con una clase abstracta genérica. Igualmente **conservamos nuestra Pizza abstracta** con dos subclases **PizzaQueso y PepperoniPizza**, lo conservamos del modelo anterior. Para finalizar añadimos un **nuevo tipo de plato Taco**, clase abstracta, con dos subclases, **TacoPastor y TacoCanasta**, dos platos específicos nuevos en el menú.

En resumen, podríamos hacer crecer el modelo infinitamente añadiendo **nuevos tipos de factorías, tipos de platos, y platos específicos**. Podríamos **añadir Kebab, Ensaladas, bocadillos y sándwiches**, etc.

Este es el modelo de orientación a objetos que iremos desgranando en los siguientes subapartados.

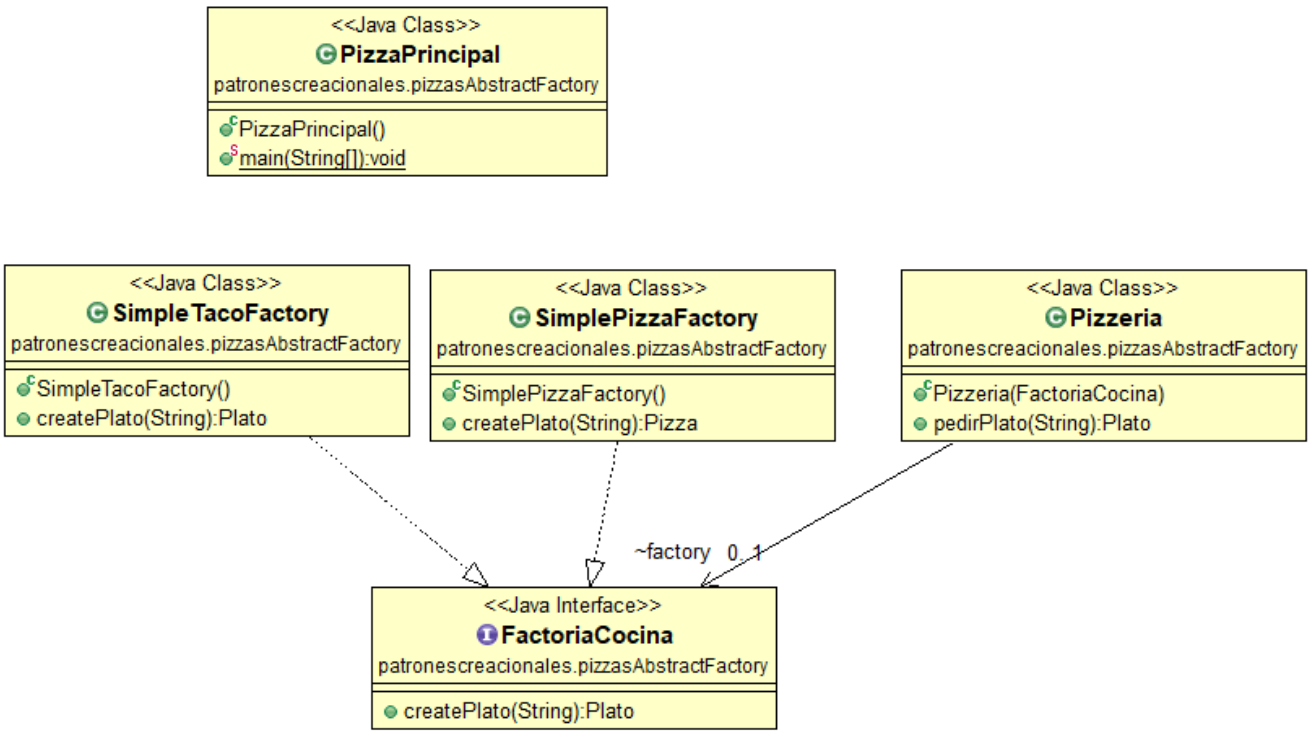


4.4.2 Pasos `para construir el nuevo modelo con Orientación a Objetos

4.4.2.1 Factorias y clases principales

En el modelo podéis **ver como la Pizzeria usa las factorías** concretas **para Taco y Pizza** a través de la **factoria abstracta cocina**. Esta es la **clave del diseño abstract Factory**. Puedo diferentes familias de productos desde una misma factoria abstracta común. Esto es así **porque el comportamiento de los familias de objetos Pizza y Taco** es **común**, debido a que todos son **platos que**

se piden y se preparan en una **cocina, factoria. Cambia el como se preparan.**



Paso 1. Añadimos una factoría abstracta con un interfaz. Cambiaremos el método **createPizza, por createPlato**, porque tenemos dos tipos de plato ahora, la pizza y el taco. Podéis verlo en el fichero **FactoriaCocina.java.**

```
public interface FactoriaCocina  {

        public Plato createPlato(String tipo);


}
```

Paso 2. En PizzaPrincipal indicamos que factoria vamos a usar para crear el plato si **la de pizzas o la de Tacos**. Pero la mantenemos sobre el mismo tipo de datos común FactoriaCocina.

Creo dos factorías concretas diferentes, pero fijaos que el tipo es la mismo, **`FactoriaCocina, en interfaz, abstracto.`**

```
FactoriaCocina factoryPizza =(FactoriaCocina) new SimplePizzaFactory();
                FactoriaCocina factoryTaco =(FactoriaCocina) new SimpleTacoFa
ctory();
```

Si quiero pedir Pizzas creo **una factoria específica para Pizzas.**

```
Pizzeria pizzeria = new Pizzeria(factoryPizza);
Plato plato = pizzeria.pedirPlato("queso");
```

Si quiero pedir Tacos creo una **factoria específica para Tacos.**

```
Pizzeria pizzeria2 = new Pizzeria(factoryTaco);
Plato plato2 = pizzeria2.pedirPlato("pastor");
```

Pero la manera de pedir el plato**, el comportamiento es el mismo,** `pedirPlato`**. Ya en el parámetro tipo de pedirPlato,** indicamos el plato en particular. La Factoria concreta lo resolverá y lo preparará.

PizzaPrincipal.java

```
public class PizzaPrincipal {


        public static void main(String[] args) {
                FactoriaCocina factoryPizza =(FactoriaCocina) new SimplePizza
Factory();
                FactoriaCocina factoryTaco =(FactoriaCocina) new SimpleTacoFa
ctory();


                Pizzeria pizzeria = new Pizzeria(factoryPizza);




                Plato plato = pizzeria.pedirPlato("queso");
                System.out.println("Hemos pedido un plato " + plato.toString(
) + "\n");


                plato = pizzeria.pedirPlato("pepperoni");
                System.out.println("Hemos pedido un plato " + plato.toString(
) + "\n");
```

```
                Pizzeria pizzeria2 = new Pizzeria(factoryTaco);

                Plato plato2 = pizzeria2.pedirPlato("pastor");
                System.out.println("Hemos pedido un plato" + plato2.toString(
) + "\n");


                plato2 = pizzeria2.pedirPlato("canasta");
                System.out.println("Hemos pedido un plato" + plato2.toString(
) + "\n");
        }
}
```

Paso 3. En pizzeria ahora usamos una factoria abstracta como atributo base FactoriaCocina. No sabemos que factoria es.

Como **veis a Pizzeria no le importa que tipo de factoria es**. Lo único que le importa es **crear el plato**. Ya **se encargarán los cocineros**, las **factorías de Pizza y Taco** de crear **el plato correcto.**

```
FactoriaCocina factory;


….
plato = factory.createPlato(tipo);
```

```
public class Pizzeria {
        FactoriaCocina factory;

        public Pizzeria(FactoriaCocina factory) {
                this.factory = factory;
        }

        public Plato pedirPlato(String tipo) {
                Plato plato;

                plato = factory.createPlato(tipo);
```

```
                plato.preparar();
                plato.cocinar();
                plato.cortar();
                plato.servir();
        return plato;
        }

}
```

Paso 4. Hacemos que la factoría concreta SimplePizzaFactory implemente el interfaz CocinaFactoria. Cambiamos créatePizza, por créatePlato para seguir el comportamiento marcado por CocinaFactoria.

SimplePizzaFactory.java

```
public class SimplePizzaFactory implements FactoriaCocina{

        public Plato createPlato(String tipo) {
                Pizza pizza = null;

                if (tipo.equals("queso")) {
                        pizza = new PizzaDeQueso();
                } else if (tipo.equals("pepperoni")) {
                        pizza = new PepperoniPizza();
                }

                return pizza;
        }
}
```

Paso 5. Añadimos la factoria para Tacos. Tenemos dos tipos de tacos, tacos pastor y tacos canasta. Empezamos con la factoria concreta para tacos. Observar que es muy parecida a la factoria para pizzas.

SimpleTacoFactory,java

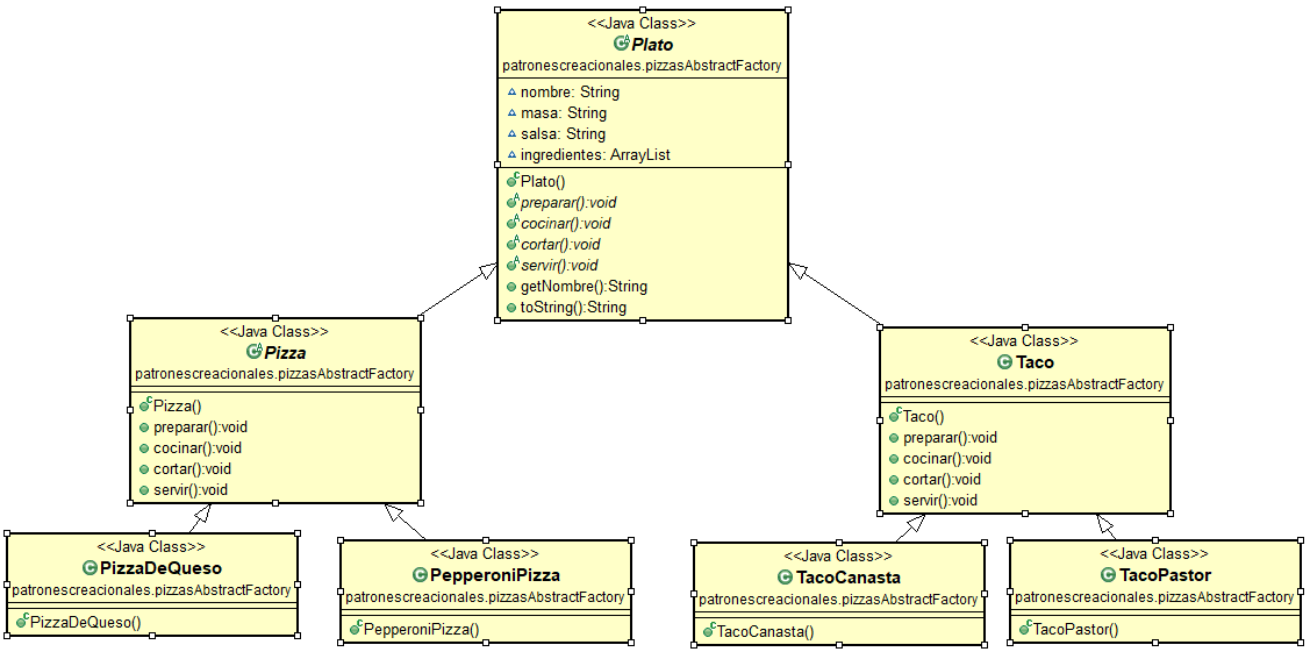
```
public class SimpleTacoFactory implements FactoriaCocina{
        public Plato createPlato(String tipo) {
```

```
                Plato plato = null;


                if (tipo.equals("pastor")) {
                        plato = new TacoPastor();
                } else if (tipo.equals("canasta")) {
                        plato = new TacoCanasta();
                }
                return plato;
        }


}
```

Paso 6. Por ultimo, nos queda añadir las nuevas familias de productos, y la clase abstracta Plato que define a las familias.

4.4.3 Nuevas familias de productos. Nuevas Herencias y clases.

Ahora tenemos un nuevo tipo de platos tenemos que integrar todo en nuestro modelo. Lo podéis ver en el siguiente diagrama de clases.



Para este **nuevo diseño añadimos un plato abstracto**, en **Plato.java. No todos los platos van a ser cortados**. Definimos **los métodos de comportamiento como abstracto**. Ahora tenemos dos familias, entonces cada familia de platos **se prepara, cocina, corta y sirve** de manera diferente.

```
import java.util.ArrayList;
public abstract class Plato {
        String nombre;
        String masa;
        String salsa;

        ArrayList ingredientes = new ArrayList<Object>();

        public abstract void preparar();


        public abstract void cocinar() ;


        public abstract void cortar();

        public abstract void servir();

        public String getNombre() {
                return nombre;
        }

public String toString() {

                StringBuffer display = new StringBuffer();
                display.append("---- " + nombre + " ----\n");
                display.append(masa + "\n");
                display.append(salsa + "\n");
                for (int i = 0; i < ingredientes.size(); i++) {
                        display.append((String )ingredientes.get(i) + "\n");
                }
```

```
                return display.toString();
        }


}
```

´De tal manera que **Pizza Abstracta y Taco Abstracto heredaran de Plato**. Ambas deben definir sus comportamientos específicos para **preparar, cocinar, cortar y servir**.

4.4.3.1 Familia Pizzas

Pizza hereda de plato ahora. Extendemos o heredamos de Plato que es la **clase abstracta** que nos marca un **comportamiento común. Sobrescribimos o implementamos esos métodos** para **asignarle el comportamiento** propio **de Pizza.**

Las **pizzas tienen su propia manera de preparar, cocinar, cortar y servir**. Es el comportamiento propio de clase, y se lo añadimos en este punto. **Concretamos familias de clases.**

Pizza.java

```
abstract public class Pizza extends Plato{


        public Pizza() {




        }
        public void preparar()
        {
                System.out.println("Hacemos la masa y añadimos los ingredient
es " + nombre);
        }


        public void cocinar() {
        System.out.println("Horneado " + nombre);
}


        public void cortar() {
```

```
                System.out.println("Cortar " + nombre);
        }

        public void servir() {
                System.out.println("Servir en una caja cerrada: " + nombre);
        }
}
```

Por último, las clases concretas **PizzaDeQueso y PepperoniPizza**, quedan exactamente igual. **No hay modificación en ese código.**

PizzaDeQueso.java

```
public class PizzaDeQueso extends Pizza {
        public PizzaDeQueso() {
                this.nombre = "Pizza de queso";
                this.masa = "Masa normal";
                this.salsa = "Salsa tomate";
                this.ingredientes.add("Mozzarella");
                this.ingredientes.add("Parmesano");
        }
}
```

PepperoniPizza.java

```
public class PepperoniPizza extends Pizza {
        public PepperoniPizza() {
                this.nombre = "Pepperoni Pizza";
                this.masa = "crujiente";
                this.salsa = "Salsa marinada";
                ingredientes.add("Rodajas de pepperoni");
                ingredientes.add("Cebolla");
                ingredientes.add("Queso Parmesano gratinado");
        }
}
```

4.4.3.2 Familia de clase Taco

Seguimos el mismo criterio para Tacos. **Extendemos o heredamos de Plato** que es la **clase abstracta** que nos marca un **comportamiento común**. **Sobreescribimos o implementamos esos métodos** para **asignarle el comportamiento de propio de Taco.**

Las **pizzas tienen su propia manera de preparar, cocinar, cortar y servir**. Es el comportamiento propio de clase, y se lo añadimos en este punto. **Concretamos familias de clases. En Taco de Taco.java, el plato no necesita ser cortado, dejamos como vacío, el método cortar().**

Taco.java

```
public class Taco extends Plato{

	public void preparar()
	{
		System.out.println("Añadimos a la barbacoa los ingredientes d
e " + nombre);
	}



	public void cocinar() {
	System.out.println("Asado " + nombre);
}

	public void cortar() {


	}

	public void servir() {
		System.out.println("Servir en un plato abierto " + nombre);
	}


}
```

Por último, nuestros productos concretos, los diferentes tacos que hemos añadido. Dos clases nuevas para implementar los platos concretos. Como veis ambas **heredan de Taco**.

TacoPastor.java

```
public class TacoPastor extends Taco{
        public TacoPastor() {
                this.nombre = "Taco Pastor";
                this.masa = "Masa harina";
                this.salsa = "Salsa Marinara";
                this.ingredientes.add("Carne");

                this.ingredientes.add("Perejil");
        }
}
```

TacoCanasta.java

```
public class TacoCanasta extends Taco{

        public TacoCanasta () {
        this.nombre = "Taco Canasta";
        this.masa = "Maiz";
        this.salsa = "Salsa Marinara";
        this.ingredientes.add("Carne");
        this.ingredientes.add("Limon");

        }
}
```

4.4.4 Practica Independiente AbstractFactory

En el ejemplo de patrón factory, creamos la factory kebab. Debéis añadirla a

la familia de productos de nuestro restaurante y nuestro modelo anterior usando el patron Abstract Factory

4.5 Patrón Prototipo

Como se ha comentado en capítulos anteriores, tanto el Factory Method como la Abstract Factory **permiten que un sistema sea independiente del proceso de creación de objetos**. En otras palabras, estos patrones **permiten a un objeto cliente crear una instancia de un clases** invocando **un método designado sin tener que especificar** la clase concreta exacta a instanciar**.** Al abordar el mismo problema el patrón Prototype ofrece otra manera de lograr el mismo resultado más flexible..

Otros usos del **patrón Prototipo incluyen**:

1. Cuando **un cliente necesita crear un conjunto de objetos** que son **iguales o difieren de unos a otros sólo en términos de su estado** y es caro crear tales en términos de **tiempo y procesamiento** involucrado.

2. Como **alternativa a la construcción de numerosas fábricas** que **reflejan las clases para se crean instancias** (como en el método Factory).

En tales casos, **el patrón Prototipo sugiere**:

1. Crear **un objeto y designarlo** como un **objeto prototipo**.
2. Cree **otros objetos simplemente haciendo una copia del objeto prototipo** y modificaciones necesarias.

En el mundo real, **usamos el patrón Prototipo** en muchas ocasiones para **reducir el tiempo y el esfuerzo dedicados a diferentes tareas**. Los siguientes son dos ejemplos de este tipo:

1. **Creación de nuevos programas de software**: los **desarrolladores generalmente tienden a la copia de un programa existente** con estructura **similar y modificarlo** para crear **nuevos programas**.

2. **Cartas de presentación** — Al solicitar puestos en diferentes organizaciones, un **solicitante no puede crear cartas de presentación** para cada organización individualmente desde cero. En su lugar, **el solicitante crearía una carta de presentación en el formato más atractivo**, hacer una copia de él y personalizarlo para cada organización.

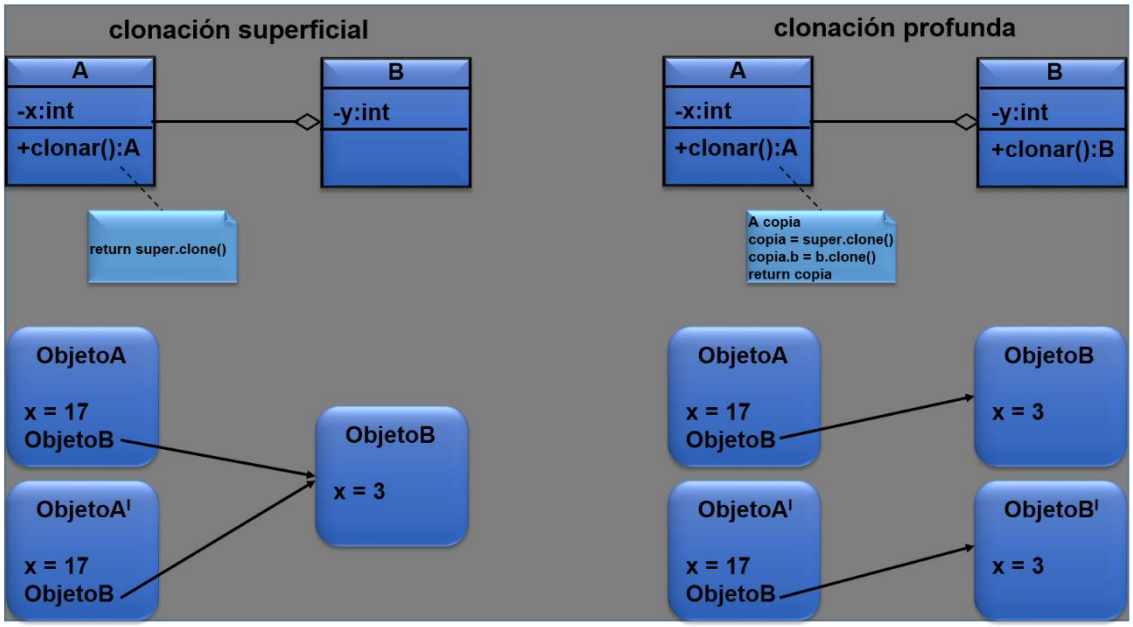
Como se puede ver en los ejemplos anteriores, **algunos de los objetos se crean desde** cero, **mientras que otros objetos se crean como copias de objetos existentes** y luego modificado. Pero **el sistema o el proceso que utiliza estos objetos no diferencia entre ellos** sobre la base de cómo se crean realmente. De una manera similar, cuando se utiliza **el patrón Prototipo, un sistema debe ser independiente** de los detalles de creación, composición y representación de los objetos que utiliza.

Uno de los **requisitos del objeto prototipo** es que **debe proporcionar una manera para que los clientes creen una copia de la misma**. De forma predeterminada, la clase clase java.lang.Object de java tiene el método clone(), pero es privado. Debemos implementar Cloneable para poder incorporar el método clone a nuestros objetos. Al incorporar el método clone() crea un clon del objeto original como una copia superficial. Para hacerla completa, se **necesita insertar código**.

4.5.1 Clonación superficial vs clonación completa

Cuando un objeto se clona como **una copia superficial**:
El **objeto de nivel superior original** y **todos sus miembros primitivos están duplicados**. Los objetos de nivel inferior, los objetos **que contiene por el patrón el objeto de nivel superior no se duplican**. **Solo se copian las referencias a estos objetos**. Esto da como **resultado tanto el original y el objeto clonado que hace referencia** a la **misma copia del objeto** de nivel inferior.

Por el contrario, **cuando un objeto se clona como una copia profunda**: **El objeto de nivel superior original y todos sus miembros primitivos están duplicados**. Los objetos de **nivel inferior que contiene el objeto de nivel superior también se duplican**. En este caso, tanto el objeto original como el objeto clonado se refieren a dos diferentes objetos de nivel inferior. La **figura siguiente se muestra estos comportamientos**.



Observar con en la copia superficial **los objetos A tienen y apuntan al mismo objeto B**. No se ha hecho una copia del Objeto B. Sin embargo, **en la clonación profunda se ha hecho una copia del objeto B también**.

4.5.2 Clonacíon en Java

Java tiene **definido en su método Object. Recordar que todas nuestras clases heredan** de **Object**. Pero **el método clone se define como privado en Object.**Para que nuestra clase pueda usar **el método clone debe implementar el interfaz Cloneable**. Este interfaz te obliga a sobreescribir el método clone.

```
public Object clone() throws CloneNotSupportedException;
```

Definiremos nuestras clases para clonar, con el interfaz Cloneable como podréis ver en el **siguiente ejemplo y sobreescribiremos el método clone(),** para hacer la copia de nuestros objetos. Es muy útil aplicar **este patrón clonación** en combinación con **el patrón objeto inmutable. Hacemos copias de objetos inmutables, no los modificamos.**

```
public class ClonacionEstudiante implements Cloneable {
```

```
@Override
public Object clone() throws CloneNotSupportedException {
```

```
            return super.clone();
        }
```

Todas **nuestras clases tienen el método clone(), lo podemos sobreescribir**. Cuando lo usamos **va a realizar una clonación**, una copia de todos **los tipos básicos y clases básicas predefinidas por java** como **String, Integer, etc**. Hay que tenerlo en cuenta para **realizar las clonaciones de vuestros objetos**. Las clases **que creemos nosotros el clone de Object no las va a clona**r, digamos que **clone() de la clase Object hace una clonación superficial**

4.5.3 Ejemplo de clonación superficial. Practica guiada de clonación en Java

En el **siguiente ejemplo clonamos la clase ClonacionEstudiante** usando **el método clone de la clase Object de java**. Hemos **definido para la Composición una clase interna propia Referencia que forma parte del Objeto EstudianteClonacion**. Vamos a demostrar **que mientras del objeto String nombre se hace una copia** con el **clone** de la clase Object. **Con el objeto Referencia no pasa.**

```
class Referencia {

        public Referencia() {

        }
}

public class ClonacionEstudiante implements Cloneable {
        private int estId;
        private String nombre;
        private Referencia referencia;
```

Lo **hemos sobreescrito en la clase EstudianteClonacion** para hacer la copia. Cuando **llamamos al super.clone()** accedemos al método clone de la **clase padre Object**.

Nos esta devolviendo una copia del objeto, rellenando todos los atributos básicos como **el id del estudiante de tipo int estId o el atributo nombre de tipo String.** De referencia **no va a hacer copia porque no es un tipo básico java.** Es un tipo **que hemos creado nosotros en la clase Referencia.**

```
public Object clone() throws CloneNotSupportedException {
		return super.clone();
	}
```

Copiamos el objeto en la función main. E imprimimos la dirección en memoria del objeto de tipo String nombre y del objeto referencia

```
ClonacionEstudiante s2 = (ClonacionEstudiante) s1.clone();
System.out.println(s1.getEstId() + " Objeto String nombre " + s1.getNombre().getBytes()+ " referencia : " + s1.getReferencia());
	System.out.println(s2.getEstId() + " Objeto String nombre " + s2.getNombre().getBytes()+ " referencia : " + s2.getReferencia());
```

Ejecutar este ejemplo. Veréis en la ejecución de **este ejemplo podéis ver que de la cadena nombre se ha hecho una copia distinta**, el **objeto referencia es el mismo.**

```
1 Objeto String nombre [B@5ca881b5 referencia : patronescreacionales.clonacion.Referencia@24d46ca6
1 Objeto String nombre [B@4517d9a3 referencia : patronescreacionales.clonacion.Referencia@24d46ca6
```

ClonacionEstudiante.java

```
package patronescreacionales.clonacion;
class Referencia {

	public Referencia() {

	}
```

```
}

public class ClonacionEstudiante implements Cloneable {
	private int estId;
	private String nombre;
	private Referencia referencia;


	public int getEstId() {
		return estId;
	}

	public void setEstId(int estId) {
		this.estId = estId;
	}

	public String getNombre() {
		return nombre;
	}

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

	public Referencia getReferencia() {
		return referencia;
	}

	public void setReferencia(Referencia referencia) {
		this.referencia = referencia;
	}

	ClonacionEstudiante(int rollno, String nombre) {
		this.estId = rollno;
		this.nombre = nombre;
		this.referencia = new Referencia();
```

```
        }
      @Override
        public Object clone() throws CloneNotSupportedException {
                return super.clone();
        }

        public static void main(String args[]) {
                try {
        ClonacionEstudiante s1 = new ClonacionEstudiante(1, "Carlos");

        ClonacionEstudiante s2 = (ClonacionEstudiante) s1.clone();

        System.out.println(s1.getEstId() + " Objeto String nombre " + s1.getN
ombre().getBytes()+ " referencia : " + s1.getReferencia());
        System.out.println(s2.getEstId() + " Objeto String nombre " + s2.getN
ombre().getBytes()+ " referencia : " + s2.getReferencia());



                } catch (CloneNotSupportedException c) {
                }

        }
}
```

4.5.4 Ejemplo Copia profunda

Este ejemplo es igual que el anterior. Solo que esta vez vamos a modificar la copia del objeto Referencia, para que veáis como clonar, copiando también nuestros propios objetos.

Hemos **introducido el método clone en Referencia2.**

```
class Referencia2 implements Cloneable{


      public Referencia2() {
```

```
	}
	public Object clone() throws CloneNotSupportedException {
		return super.clone();
	}

}
```

Hemos modificado el método clone de la clase principal **EstudianteClonacionProfunda** para que hagamos una copia también del objeto referencia.

En este punto copiamos el objeto con una copia superficial.
ClonacionEstudianteProfunda copia= (ClonacionEstudianteProfunda) super.clone();

Copiamos el objeto Referencia2 para que la copia sea profunda.
copia.setReferencia((Referencia2) this.referencia.clone());

```
public Object clone() throws CloneNotSupportedException {
		ClonacionEstudianteProfunda copia= (ClonacionEstudianteProfunda) super.clone();

		copia.setReferencia( (Referencia2) this.referencia.clone());
		return copia;
	}
```

Si **ejecutáis el ejemplo veréis** que los **objetos referencia en este caso son distintas**. Hemos realizado una **clonación profunda**.

```
1 Carlos referencia : patronescreacionales.clonacion.Referencia2@7a81197d
1 Carlos referencia : patronescreacionales.clonacion.Referencia2@5ca881b5
```

EstudianteClonacionProfunda.java

```
class Referencia2 implements Cloneable{

	public Referencia2() {

	}
	public Object clone() throws CloneNotSupportedException {
		return super.clone();
	}

}


public class ClonacionEstudianteProfunda  implements Cloneable {
	private int estId;
	private String nombre;
	private Referencia2 referencia;


	public int getEstId() {
		return estId;
	}

	public void setEstId(int estId) {
		this.estId = estId;
	}

	public String getNombre() {
		return nombre;
	}

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

	public Referencia2 getReferencia() {
```

```
                return referencia;
        }

        public void setReferencia(Referencia2 referencia) {
                this.referencia = referencia;
        }

        ClonacionEstudianteProfunda(int rollno, String nombre) {
                this.estId = rollno;
                this.nombre = nombre;
                this.referencia = new Referencia2();
        }

        public Object clone() throws CloneNotSupportedException {
                ClonacionEstudianteProfunda copia=  (ClonacionEstudianteProfu
nda) super.clone();

                copia.setReferencia( (Referencia2) this.referencia.clone());
                return copia;
        }

        public static void main(String args[]) {
                try {
                        ClonacionEstudianteProfunda s1 = new ClonacionEstudi
anteProfunda(1, "Carlos");

                        ClonacionEstudianteProfunda s2 = (ClonacionEstudiant
eProfunda) s1.clone();

                        System.out.println(s1.getEstId() + " " + s1.getNombr
e()+ " referencia : " + s1.getReferencia());
                        System.out.println(s2.getEstId() + " " + s2.getNombr
e()+ " referencia : " + s2.getReferencia());

                } catch (CloneNotSupportedException c) {
                }

        }
```

```
}
```

4.5.5 Ejercicio Clonación. Practica independiente

1. Del tema anterior debéis tener hecho la clase **TrabajadorInmutable**. Convertir **esta clase en Cloneable**, e **implementar una copia superficial** del mismo.

2. De la clase **Claustro** del tema anterior implementar una **copia profunda**.

5 Actividades de refuerzo

1. Implementar **un patrón factory para crear los diferentes objetos de Trabajador** en el modelo de educación.
2. **Implementar un patrón Builder para crear los diferentes objetos de Trabajador** en el **modelo de educación**. En este caso **eliminaremos todas las subclases menos consejero de educación** y crearemos una clase enumerada que contendrá los diferentes tipos de trabajador
3. Implementar cloneable para la clase Trabajador, con copia profunda.

6 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