Clase 2 - Patron Template Method

Status	Patrones
Assign	

Ejemplo: varias clases que hacen lo mismo, pero con algunas diferencias. El Patrón TM nos ayuda a reducirlo.

Los patrones de diseño: ayudan a resolver problemáticas comunes en el desarrollo.

Creacionales, estructurales y de comportamiento.

> 💡 Es un patrón de **comportamiento**, es decir que nos va a yudar a resolver problemas de interacción entre las clases y objetos.

Define un método esqueleto con el código que las clases tengan en común, permitiendo que algunas partes puedan ser modificadas por las subclases que implementen este método esqueleto. Ubica en un solo lugar el código repetido.

Propone **abstraer todo el comportamiento que comparten las entidades en una clase abstracta de la que se extiendan otras entidades**. Esta clase abstracta establecerá el esqueleto del método y delegará a las clases hijas la implementación de esos métodos.

`Ventajas`:

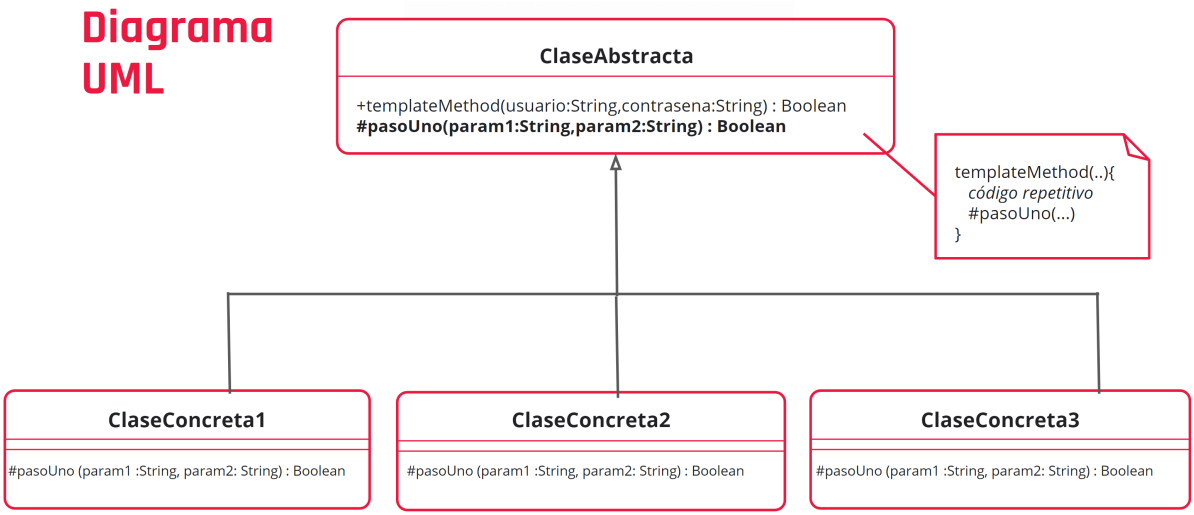
- Los clientes pueden sobrescribir ciertas partes de un algoritmo grande para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo.
- Se puede colocar el código duplicado dentro de una superclase.

`Desventajas`:

- Algunos clientes pueden verse limitados por el esqueleto proporcionado por el algoritmo.
- Los métodos “esqueleto” tienden a ser más difíciles de mantener cuantos más pasos a implementar tengan.

✅ Al eliminar el código repetido, nuestro código será más eficiente, legible y mantenible. Esto hará que sea más fácil de extender y mejorar.

Diagrama UML



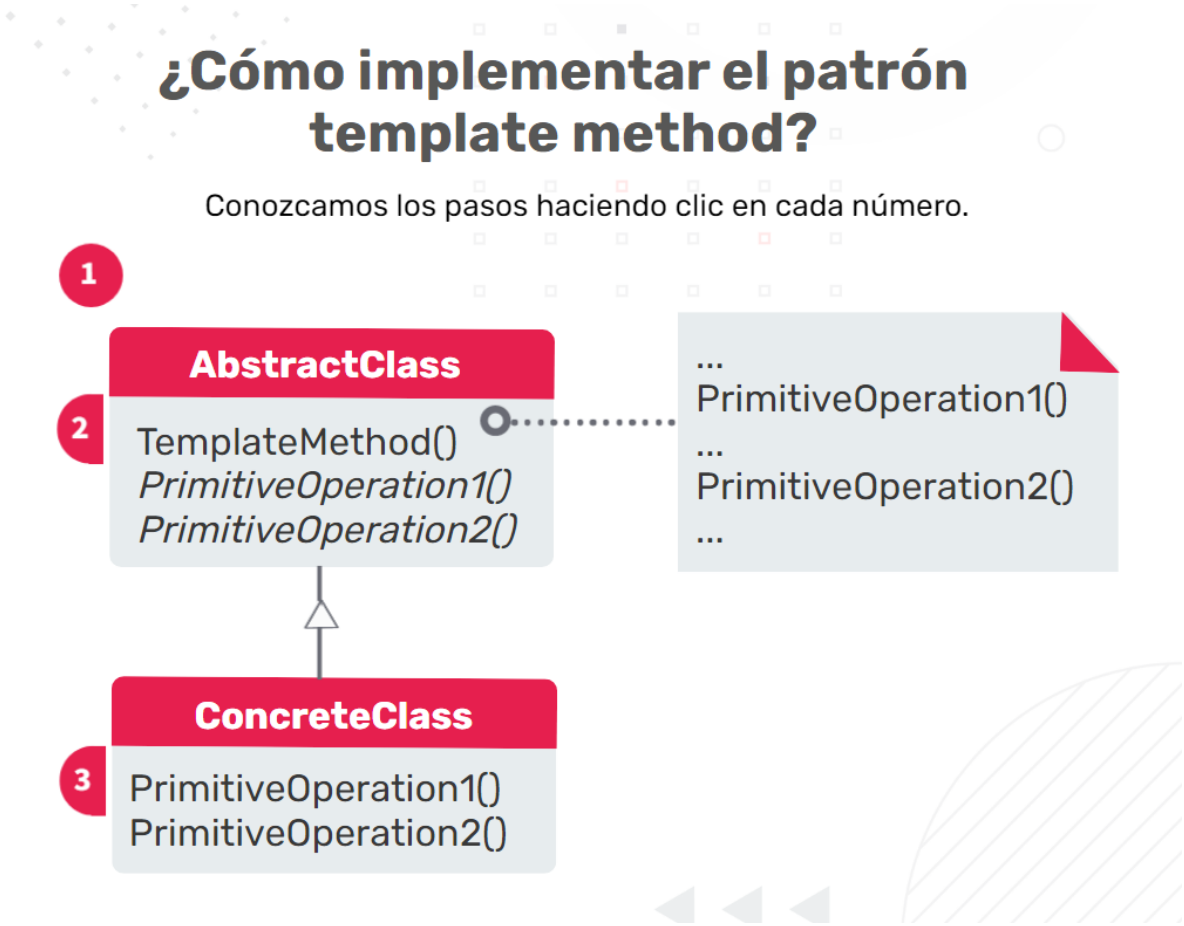
Propósito y solución

Propósito

Es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase, pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

Solución

El método esqueleto está conformado por el código que estas clases tienen en común, permitiendo que algunas partes puedan ser modificadas por la subclase que implemente el mismo, logrando ubicar en un solo lugar el código repetido.



1) Consideramos qué pasos son comunes a todas las subclases en las que vemos código repetido y cuáles siempre serán únicos.

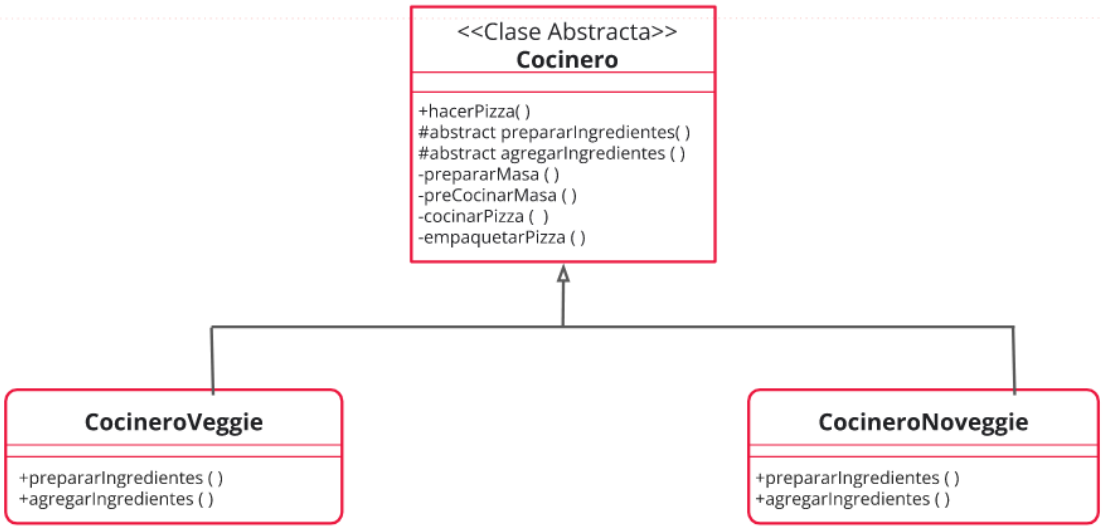
2) Creamos la clase base abstracta y declaramos el método esqueleto y un grupo de métodos abstractos que representan los pasos del algoritmo a implementar por las subclases.

3) Para cada variación del método, creamos una nueva subclase concreta.
Esta **debe**
implementar todos los pasos abstractos.

Ejemplo

Por cada variedad de pizza, los cocineros tienen que hacer todos esos pasos. Aplicando el patrón template method, podríamos crear el método esqueleto, con los pasos comunes a todas las pizzas y dejar que los cocineros solo se preocupen por los pasos que no son comunes a todas las pizzas, en este caso, **preparar los ingredientes y agregar los ingredientes.** Veamos cómo representamos esta solución en un diagrama UML.

Solución



Implementación de las cases del diagrama UML

```
public abstract class Cocinero {

    public void hacerPizza(){
        prepararMasa();
        preCocinarMasa();
        prepararIngredientes();
        agregarIngredientes();
        cocinarPizza();
        empaquetarPizza();

    }
```

Código de la clase abstracta que define el método esqueleto y los métodos abstractos a implementar por las subclases.

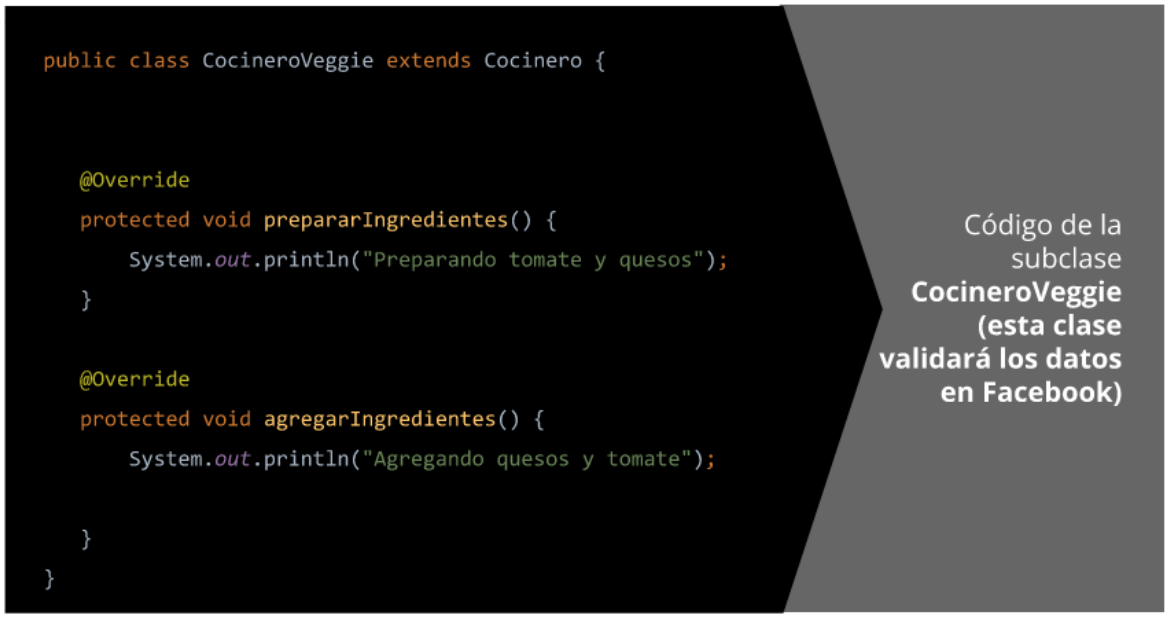
```
    protected abstract void prepararIngredientes();
    protected abstract void  agregarIngredientes();
    private void prepararMasa(){
        System.out.println("Preparando masa..");
    }
    private void preCocinarMasa(){
        System.out.println("Pre cocinando masa..");
    }
    private void cocinarPizza(){
        System.out.println("Enviando al horno la pizza");
    }
    private void empaquetarPizza(){
        System.out.println("Empaquetando pizza");
    }
}
```

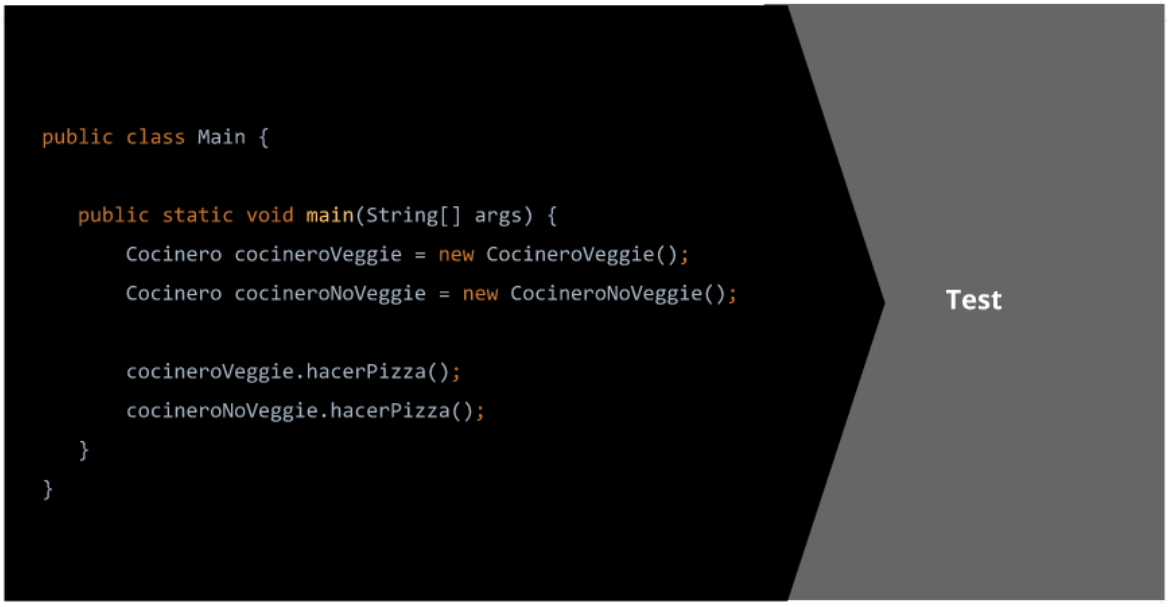
Código de la clase abstracta que define el método esqueleto y los métodos abstractos a implementar por las subclases.

```
public class CocineroNoVeggie extends Cocinero {
    @Override
    protected void prepararIngredientes() {
        System.out.println("Preparando queso y jamón,");
    }

    @Override
    protected void agregarIngredientes() {
        System.out.println("Agregando los ingredientes");
    }
}
```

Código de la subclase **CocineroNoVeggie**





Ejercicio Playground

Pensemos en un sistema para validar pagos, ya sea por tarjeta de crédito o de débito. Una tarjeta está compuesta por los **números del frente**, un **código de seguridad** y una **fecha de expiración**. A su vez, la tarjeta puede ser de **crédito o de débito**. Si es de crédito, tendrá un campo para el **límite** y otro para el **saldo utilizado**. Si es de débito, tendrá un único campo para el **saldo** disponible.

Para realizar un pago con tarjeta, es necesario recibir una autorización. Si la tarjeta es de débito, la autorización se produce si el saldo de la cuenta sobre la que se debita la tarjeta es suficiente. En el caso de crédito, se produce si el límite no ha

sido superado. Actualmente, contamos con dos servicios llamados: **ProcesadorCredito** y **ProcesadorDebito**. Ambos cuentan con un método para procesar el pago, pero antes, validamos que la fecha de expiración sea posterior a la fecha actual. Esa lógica es común, independientemente si el pago se está realizando con débito o crédito, es decir, tenemos un código que se repite en los dos métodos. Los métodos que utilizan los dos procesadores de pago reciben la tarjeta y el monto a cobrar.

Nos gustaría eliminar el código repetido. ¿Cómo lo resolverías aplicando el patrón template method? También nos gustaría poder ver por consola si el pago pudo ser autorizado. Te proponemos realizar el diagrama UML e implementación en JAVA.

Notas de la clase

Date: @August 10, 2021

Topic: Template Method

Notes

- Este patrón se usa para crear plantillas de clases en donde se respeta la herencia.

📌 **RESUMEN:**

Código en clase

Consigna

Para realizar la liquidación de sueldos es necesario realizar **tres pasos**. `Calcular el sueldo` correspondiente, imprimir o `generar recibos de sueldo` y `depositar` el importe correspondiente.

Hay distintos `tipos de empleados`. En el caso de los empleados `efectivos` o en “relación de dependencia”, **el cálculo es a partir de un sueldo básico y se agregan descuentos y premios**. Los empleados `contratados` trabajan una

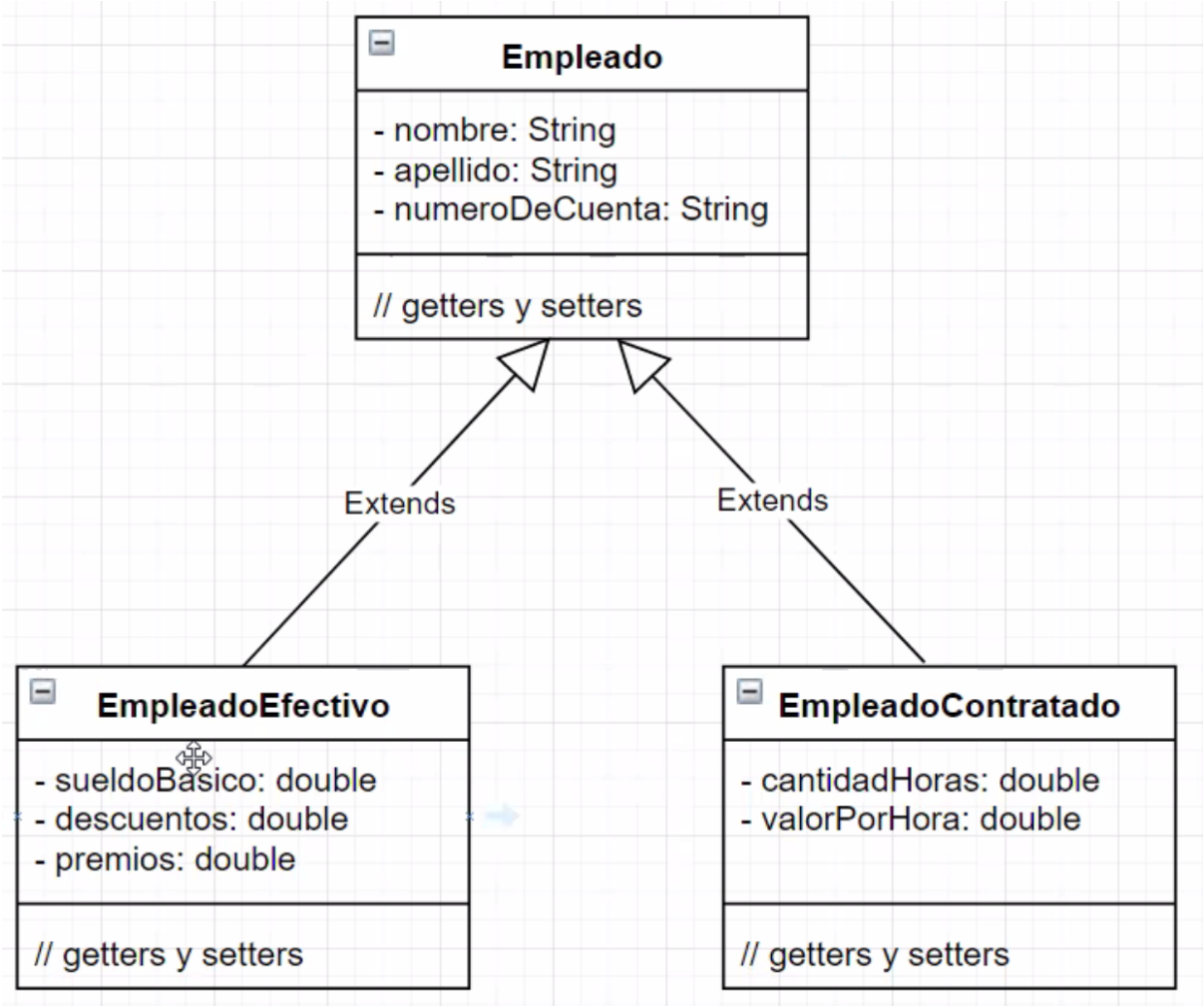
cantidad de horas, cada una con cierto **valor**. Con esta información se realiza el cálculo del sueldo a pagar y luego se imprime, por último se deposita.

El proceso que realizamos es `liquidacionSueldo`. Este lleva tres pasos: cálculo, impresión y depósito. En cuanto al `cálculo`, este se realiza según el tipo de empleado. La `impresión` puede ser digital, en el caso de los contratados, o en un recibo en papel en el caso de los efectivos, y por último el `depósito` en una cuenta bancaria.

Los empleados `efectivos` tienen como información, **sueldo básico** y un **valor fijo de descuentos** y otro de **premios**. Los empleados `contratados` tienen **cantidad de horas trabajadas** y **valor hora**. Ambos deben tener **nombre**, **apellido** y un **número de cuenta** donde se deposita el sueldo.

Para simplificar vamos a emitir un mensaje en la parte de `imprimir` el recibo que **informe si es un documento digital o un recibo impreso.**

UML



Código