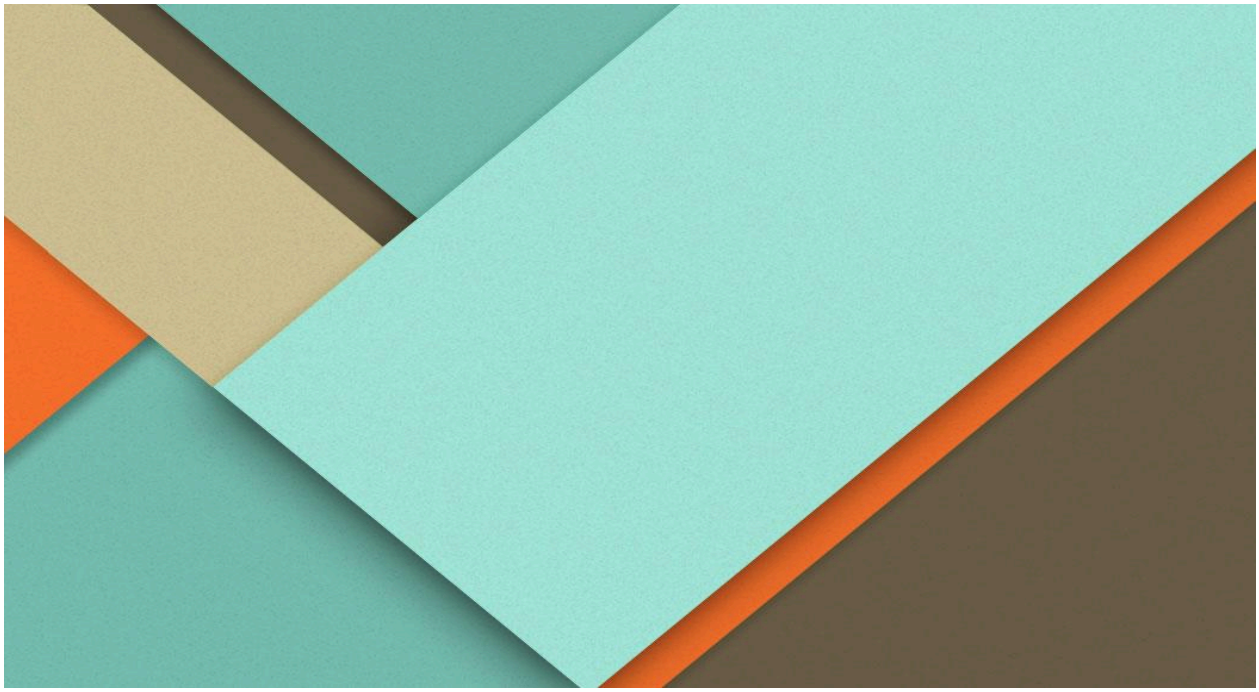


Proyecto Gestor de Pedidos

———————————————

Alejandra Ruiz Bermúdez

1º D.A.W.

Ciclo de Desarrollo de Aplicaciones Web (DAW)

Entornos de Desarrollo

I.E.S. Mar de Cádiz - El Puerto de Santa María

Índice

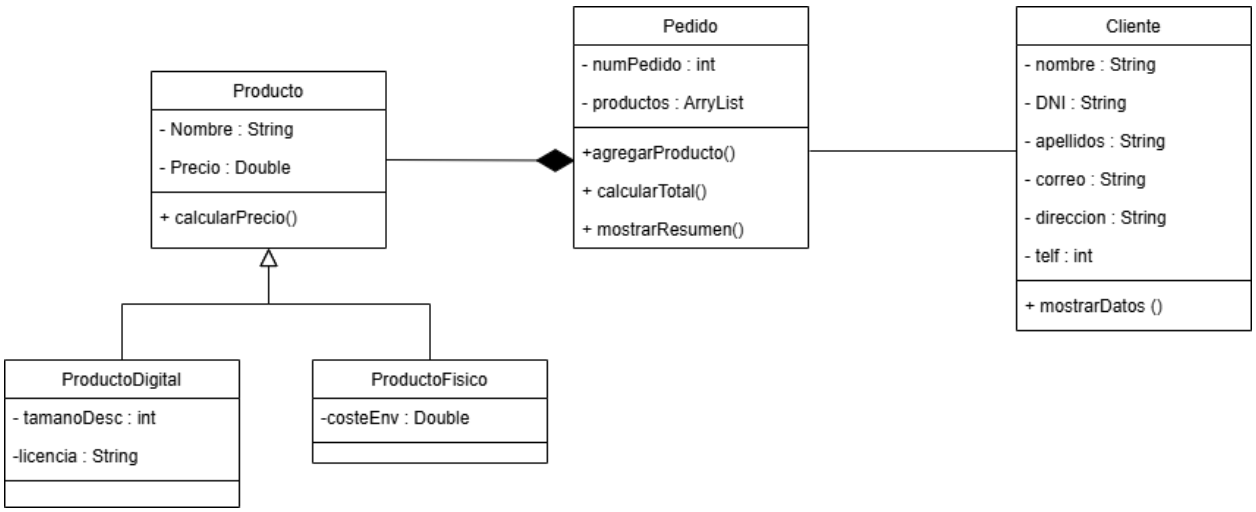
Introducción.

En este proyecto se pide crear un programa que ayude a gestionar los pedidos de una tienda. En este documento quedará reflejado el proceso de creación de este, pasando por el diseño del diagrama UML hasta la realización del código en java. En el enunciado se indica que se deberá hacer un diagrama UML utilizando ingeniería inversa y comparar ambos diagramas. Además, también se generará código a partir del primer diagrama UML, también para comparar.

Creación del Diagrama UML

Lo primero que se debe hacer es planificar cómo serán las clases y sus relaciones. En este caso las clases se definen en el enunciado, Cliente, Pedido, Producto, ProductoFísico y ProductoDigital. Las relaciones también vienen predefinidas, tenemos Herencia con Producto y sus hijos, Físico y Digital, tenemos Asociación con Cliente y Pedido, y por último, Composición o Agregación con Producto y Pedido. Con la relación Producto y Pedido creo que lo mejor será que tengan una relación de Composición dado a que los pedidos existan dependen de que haya productos que comprar en primer lugar.

El diagrama final queda tal que así :



Implementación de código

Una vez que el diagrama ya ha sido creado se puede empezar a crear el código. Lo primero es crear las diferentes clases una por una, exceptuando, Producto Digital y Producto Físico que ya se hablara de ellas más adelante. Por ejemplo:

```java
public class Cliente {      Ale, last week • new file:   src/Cliente.java …

    private String nombre;
    private String DNI;
    private String apellidos;
    private String correo;
    private String direccion;
    private int telf;

    //constructor
    public Cliente ( String nombre, String DNI, String apellidos, String correo, String direccion, int telf){
        this.nombre = nombre;
        this.DNI = DNI;
        this.apellidos = apellidos;
        this.correo = correo;
        this.direccion = direccion;
        this.telf = telf;
    }

    //metodos set y get
    public void setnombre(){ this.nombre = nombre;}
    public void setDNI(){ this.DNI = DNI;}
    public void setApellidos(){ this.apellidos = apellidos;}
    public void setCorreo(){ this.correo = correo;}
    public void setDireccion(){ this.direccion = direccion;}
    public void setTelf(){ this.telf = telf;}

    public String getNombre() {return this.nombre;}
    public String getDNI() {return this.DNI;}
    public String getApellidos() {return this.apellidos;}
    public String getCorreo() {return this.correo;}
    public String getDireccion() {return this.direccion;}
    public int getTelf() {return this.telf;}
}
```

En esta primera clase Cliente lo primero que definimos son sus atributos. Los atributos tienen la siguiente estructura:

```java
private String nombre;
private String DNI;
private String apellidos;
private String correo;
private String direccion;
private int telf;
```

Private indica el tipo de protección que tiene, en este caso no son accesibles de manera directa, la siguiente palabra indica el tipo de variable que va a ser el atributo, y, finalmente, el nombre del atributo.

Después podemos observar el constructor de la clase que sirve para inicializar objetos de una clase automáticamente al crearlos. Su principal función es establecer los valores iniciales de los atributos del objeto, preparándolo para su uso y asegurando que comience en un estado válido y bien definido.

```
String nombre - Cliente.Cliente(String, String, String, String, String, int)
//constructor
public Cliente ( String nombre, String DNI, String apellidos, String correo, String direccion, int telf){
    this.nombre = nombre;
    this.DNI = DNI;
    this.apellidos = apellidos;
    this.correo = correo;
    this.direccion = direccion;
    this.telf = telf;
}
```

Para terminar con lo básico que comparten todas las clases, tenemos los métodos Set y Get. Cuando ponemos private en un atributo hemos explicado que no se pueden acceder a los datos de manera directa, los métodos get y set se utilizan para definir, editar, borrar y ver los atributos.

```
//metodos set y get
public void setnombre(){ this.nombre = nombre;}
public void setDNI(){ this.DNI = DNI;}
public void setApellidos(){ this.apellidos = apellidos;}
public void setCorreo(){ this.correo = correo;}
public void setDireccion(){ this.direccion = direccion;}
public void setTelf(){ this.telf = telf;}

public String getNombre() {return this.nombre;}
public String getDNI() {return this.DNI;}
public String getApellidos() {return this.apellidos;}
public String getCorreo() {return this.correo;}
public String getDireccion() {return this.direccion;}
public int getTelf() {return this.telf;}
```

En producto tenemos una peculiaridad, tiene dos hijos, es decir, la clase Producto tiene una relación de herencia con otras dos clases que dependen de esta. Los tipos de Producto (Físico o Digital) heredan los atributos de su clase padre. En el código lo dejamos por escrito de la siguiente manera:

```
Ale, last week | 1 author (Ale)
public  class ProductoDigital extends Producto{
    private int tamanoDesc;
    private String licencia;
```

En este caso, la clase Producto es abstracta por el hecho de que existe para que sus hijos puedan heredar sus atributos.

```
abstract class Producto {

    private String nombreProd;
    private double precio;
```

También las clases pueden tener métodos propios, es el caso de la clase Pedido que nos ayuda a calcular el total del pedido, a mostrar un Resumen general del pedido, etc. Los métodos tienen la siguiente estructura:

```
// metodo calcularTotal() (vamos sumarle el IVA)
public double calcularTotal() {

    double suma = 0;

    for (Producto p : productos) {
        suma += p.calcularPrecio(p);
    }

    return suma * 1.21;
    // para calcular el IVA del 21%

}
```

Empezamos diciendo que es un método público para que todos los que tengan acceso al programa puedan utilizarlo, lo siguiente es poner el tipo de valor que va a devolver, y el nombre del método.

El uso del Main.java

El main.java sirve para realizar pruebas y asegurar que el código funcione correctamente, en este caso es imperativo crear un objeto de todos las clases e ir pasando por todos los métodos. Por ejemplo, así se vería la creación de un Cliente:

```
//crear cliente
Cliente cliente1 = new Cliente(
    "Juan",
    "12345678A",
    "García López",
    "juan.garcia@email.com",
    "Calle Mayor 123, Madrid",
    612345678
);
```

Lo primero que hacemos es llamar a la clase Cliente, y luego poner el nombre del nuevo cliente. Lo que hay después del = nos permite escribir uno por uno los atributos del cliente, (como nota, se puede poner null para dejarlo en blanco.)

```
// Probar cálculo de precio individual
System.out.println(x: "\n Precios individuales (con costes adicionales):");
System.out.println("- " + libro.getNombre() + ": " +
    String.format(format: "%.2f", libro.calcularPrecio(libro)) + "€ (incluye envío)");
System.out.println("- " + software.getNombre() + ": " +
    String.format(format: "%.2f", software.calcularPrecio(software)) + "€");
```

En la imagen superior se puede observar cómo se llama a diferentes métodos de Producto. Para llamar a un método ponemos el nombre del objeto y acto seguido ponemos el nombre del método con un " . " para separar los nombres.

Ingeniería Inversa

Creación de código a partir del UML

A partir de mi diseño del diagrama UML se generarán automáticamente las clases. En este caso, gracias a un código .py, (proporcionado por la profesora) he conseguido que saque un código a partir de este, observando las clases no hay muchas diferencias notables. Por ejemplo:

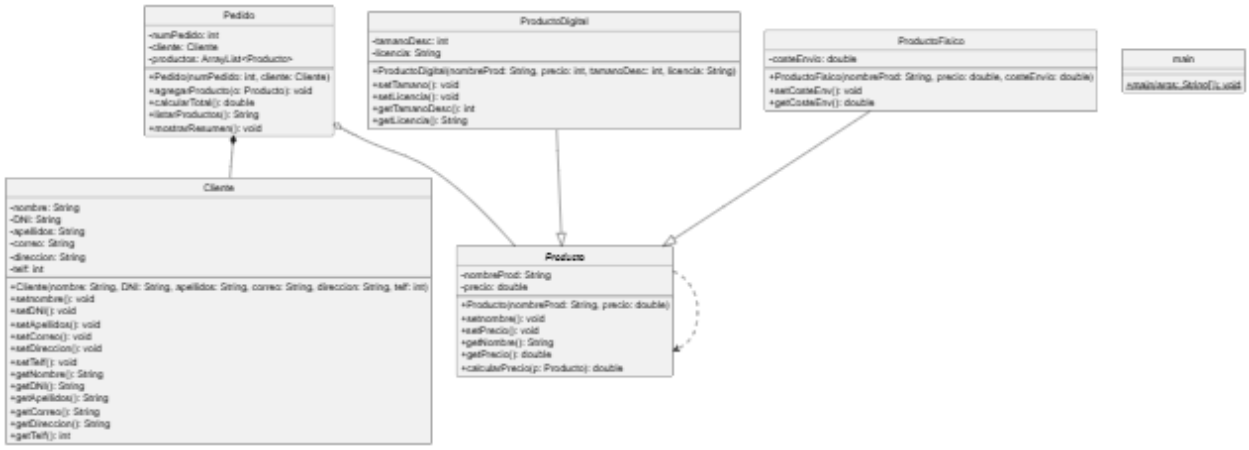
```java
You, 10 minutes ago | 1 author (You)
public class Cliente {      You, 10 minutes ago • new file:   Diagramas/output/cod_Generado/Clien…
    private String nombre;
    private String DNI;
    private String apellidos;
    private String correo;
    private String direccion;
    private int telf;
    public void Cliente(String nombre, String DNI, String apellidos, String correo, String direccion, int telf) {
        // TODO: Implementar el m□todo
    }
    public void setnombre() {
        // TODO: Implementar el m□todo
    }
    public void setDNI() {
        // TODO: Implementar el m□todo
    }
    public void setApellidos() {
        // TODO: Implementar el m□todo
    }
    public void setCorreo() {
        // TODO: Implementar el m□todo
    }
    public void setDireccion() {
        // TODO: Implementar el m□todo
    }
    public void setTelf() {
        // TODO: Implementar el m□todo
    }
```

Esta es la clase producto que se ha generado automáticamente, la diferencia entre ambos es que yo separo el listado de productos en un método aparte pero porque fue una decisión tomada en la implementación del código y no en el diseño.

Por lo demás es un programa bastante simple, lo suficiente como para que no haya diferencias grandes entre ambos códigos, sobre todo porque la gran mayoría de código son atributos y métodos get y set.

Creación de diagrama a partir del código

Gracias a programas como PlantUML podemos crear diagramas UML automáticamente de nuestro código, así es como están conectadas las clases en el proyecto mirando solo el código.



Diferencias Notables

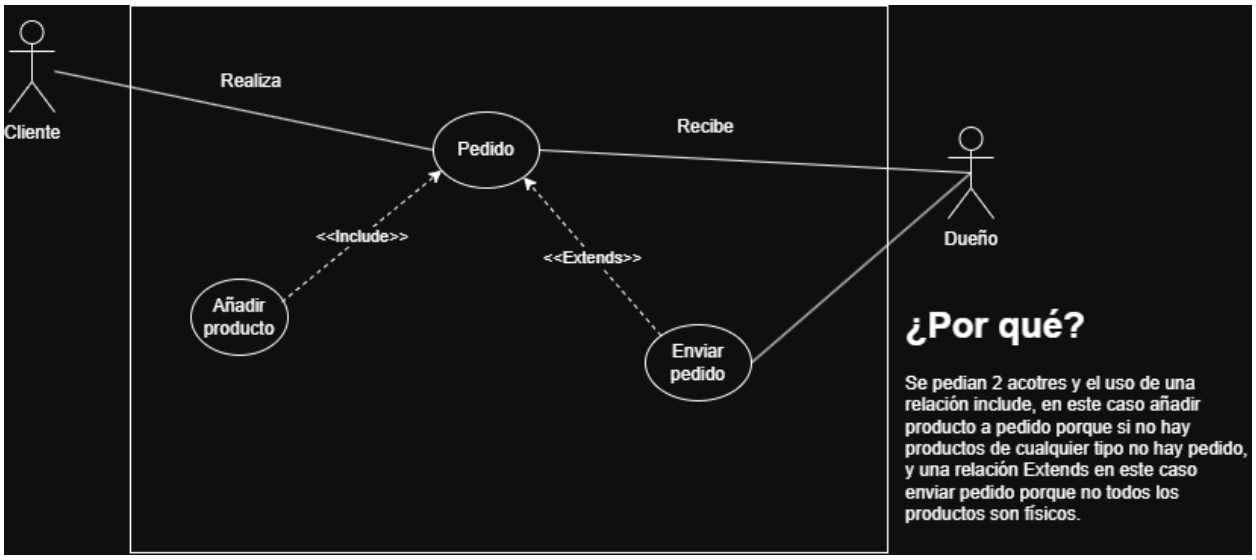
No se observan diferencias significativas entre mi diseño del diagrama y el diagrama generado automáticamente por el software, lo cual indica que la aplicación del modelo conceptual y las especificaciones de diseño se ha realizado de manera correcta y coherente.

Diagramas UML

En esta siguiente parte de las memorias se entrará en detalle sobre el desarrollo de diferentes Diagramas que han sido pedidos:

Diagramas de Casos

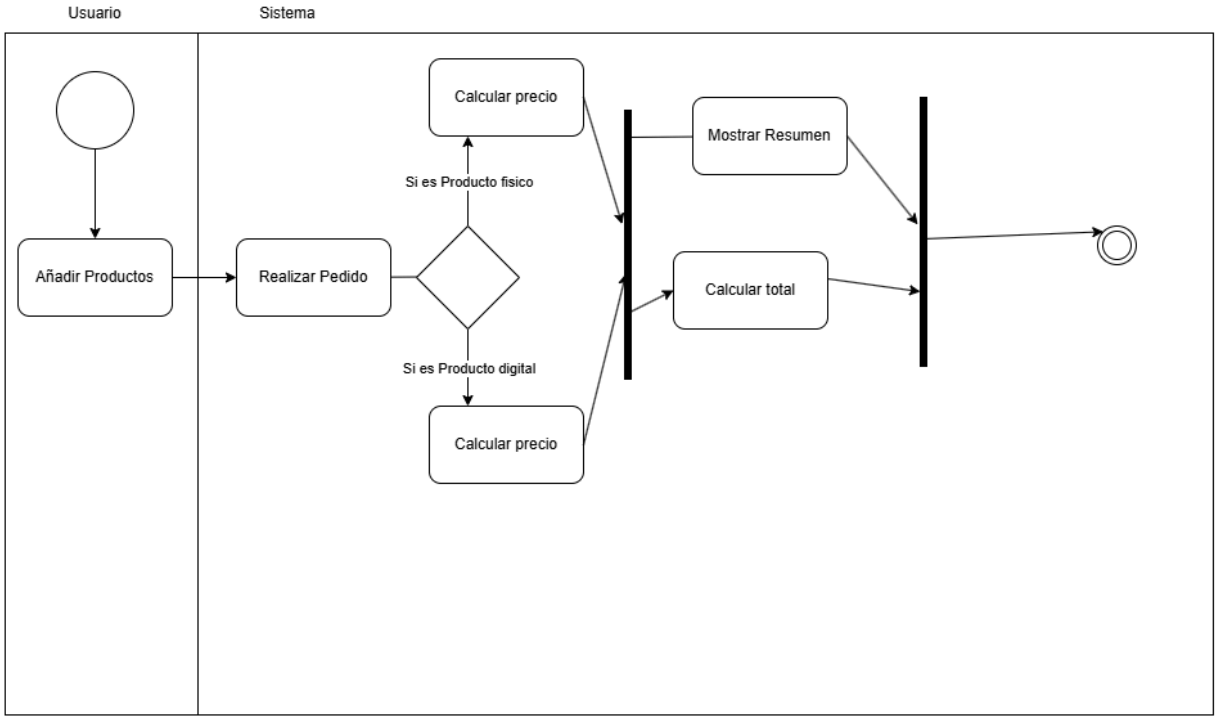
Un diagrama de casos de uso es un diagrama UML que muestra qué puede hacer un sistema y quién lo usa, sin entrar en detalles de cómo se implementa.



Como se ve en la imagen, se pedía que hubieran más de dos actores, una relación con la etiqueta <<Extends>> y otra con la etiqueta <<Include>>. En la relación include es de esta manera porque para que exista el pedido **siempre** se debe añadir productos antes. En el caso de la relación extends no siempre se va a realizar un envío de pedido dado a que no todos los productos son físicos.

Diagramas de Actividades

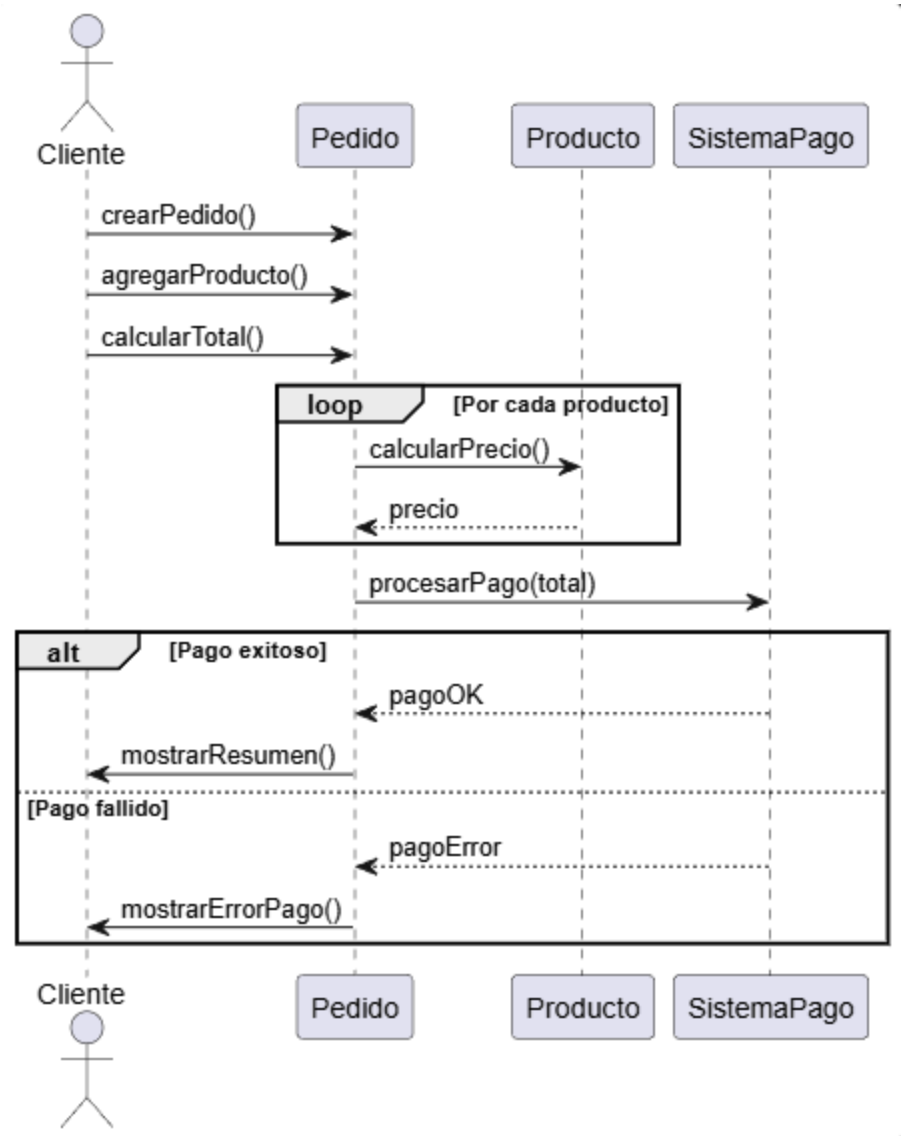
Un diagrama de actividades es un diagrama UML que representa el flujo de acciones de un proceso, mostrando qué se hace, en qué orden y bajo qué condiciones.



Se pedía específicamente que hubiera un Fork o un Join, en este caso he añadido los dos porque dependiendo de si es un producto físico o digital se calculan los totales de manera diferente luego se pasa a calcular el total y mostrar que se realizan simultáneamente.

Diagramas de Secuencia

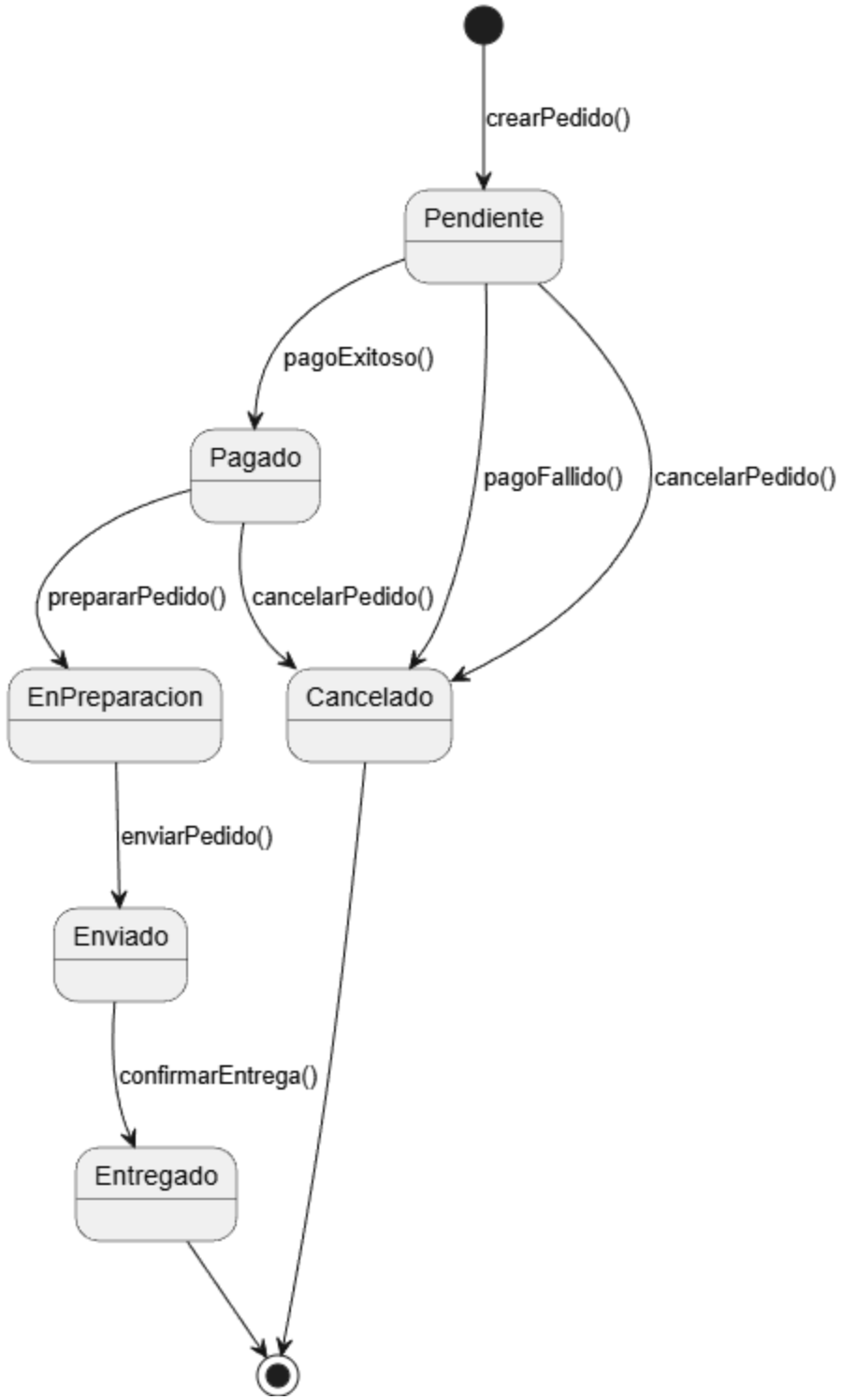
Un diagrama de secuencia es un diagrama UML que muestra el intercambio de mensajes entre los objetos de un sistema a lo largo del tiempo, en el orden en que ocurren.



Se muestran en el diagrama el paso de mensajes para el cálculo del total de un pedido.Utilizando fragmentos loop para recorrer productos y alt para gestionar el éxito/fallo del pago.

Diagramas de Estados

Un diagrama de estados es un diagrama UML que muestra los distintos estados por los que pasa un objeto durante su vida y qué eventos provocan los cambios entre esos estados.



El diagrama muestra los distintos estados de un Pedido y los eventos que provocan las transiciones entre ellos. El pedido inicia en Pendiente y puede finalizar en Entregado o Cancelado, según el resultado del pago y las acciones posteriores.

Conclusión

Como conclusión, el desarrollo de este proyecto ha permitido recorrer de forma completa el proceso de diseño y construcción de un sistema orientado a objetos, comenzando por la elaboración del diagrama UML y finalizando con la implementación en Java. A través de este flujo de trabajo, se ha demostrado la importancia de planificar adecuadamente la estructura del programa antes de escribir código, ya que el modelo conceptual facilitó la organización de las clases, sus atributos, sus métodos y las relaciones entre ellas.

La comparación entre el diagrama inicial y el código generado por ingeniería inversa confirma que el diseño planteado fue coherente y correctamente aplicado, puesto que ambos diagramas presentan una estructura prácticamente equivalente.

Aunque el código funciona correctamente y refleja bien el diseño inicial, existen diversas mejoras que se podrían implementar:

- Agregar una validación de datos, se podrían sanear las respuestas introducidas por el usuario para verificar si son válidas o no.
- Implementar manejo de excepciones, especialmente en las operaciones como los cálculos, creación de objetos o gestión de listas.

Bibliografía

Todo el proyecto se irá registrando en Github:

https://github.com/Alerb25/Sistema_GestionPedidos