PARCIAL GRUPAL: METODOLOGIA.

Tema: Todo lo visto en el cuatrimestre

Integrantes: Martinez Balian, Francisco; Landeira, Agustin; Gimenez, Magali Andrea.

Materia: Metodología de sistemas II.

Carrera: Tecnicatura Universitaria en Programación.

Docente: Castillo, Rocio.

Institución: UTN (Universidad Tecnologica Nacional).

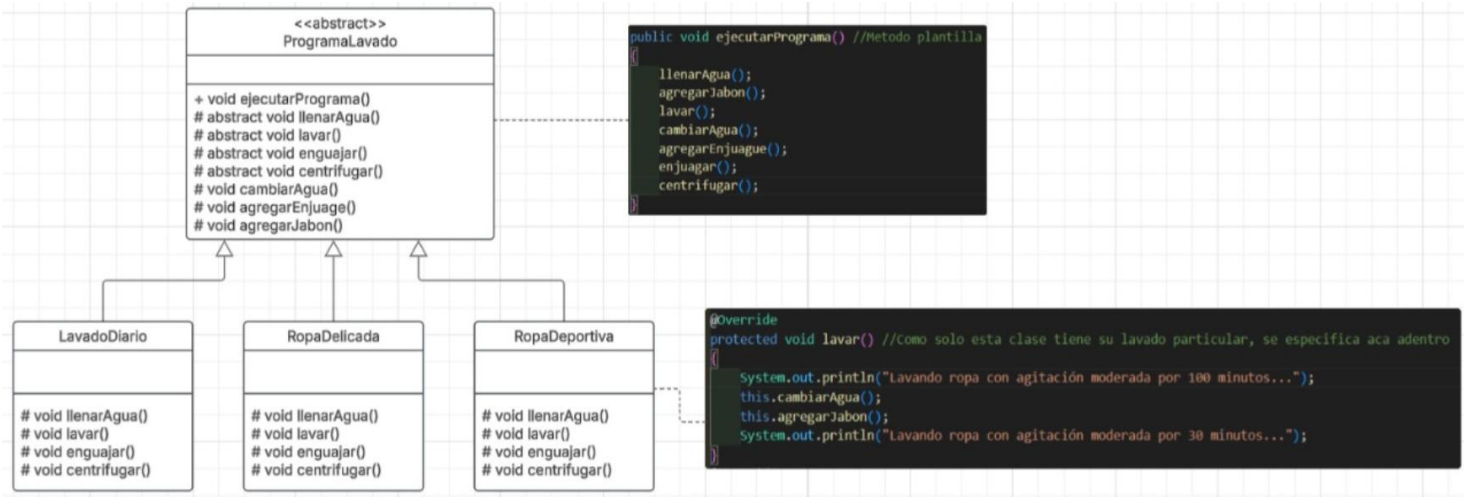
Fecha de entrega: 03/07/2025.

Índice.

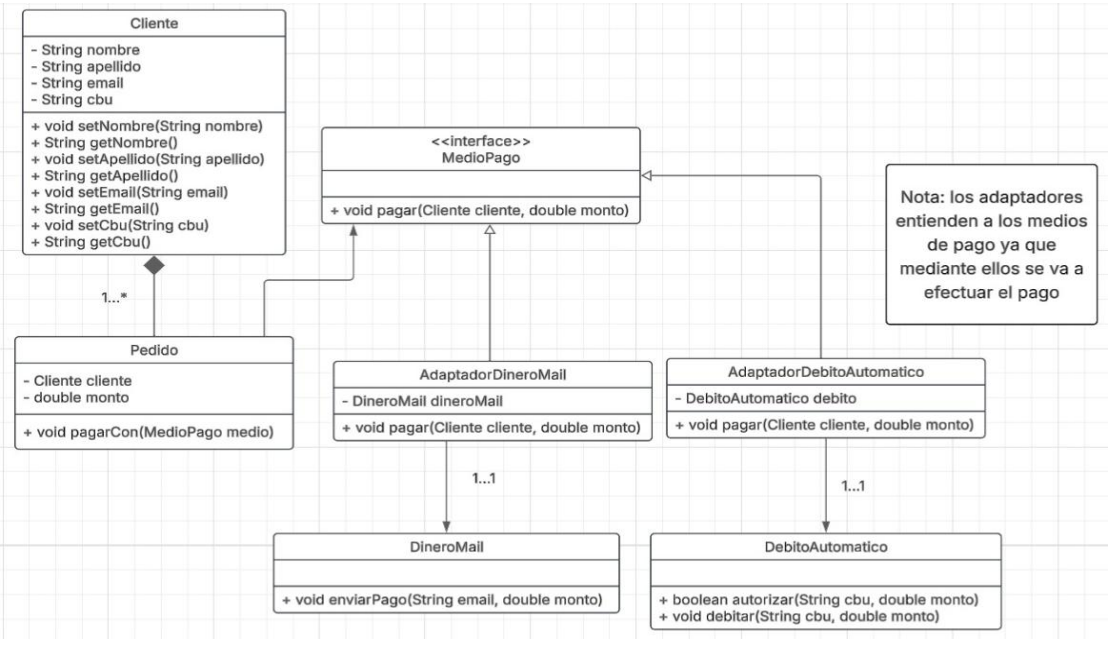
PATRONES DE DISEÑO:

1. En este caso se utiliza el patrón Template Method, ya que cada programa de lavado sigue una estructura general común, con pasos definidos como: llenar agua, agregar jabón, lavar, enjuagar, centrifugar, pero algunos pasos cambian según el tipo de ropa (por ejemplo, la cantidad de agua, los minutos o si se omite el centrifugado).

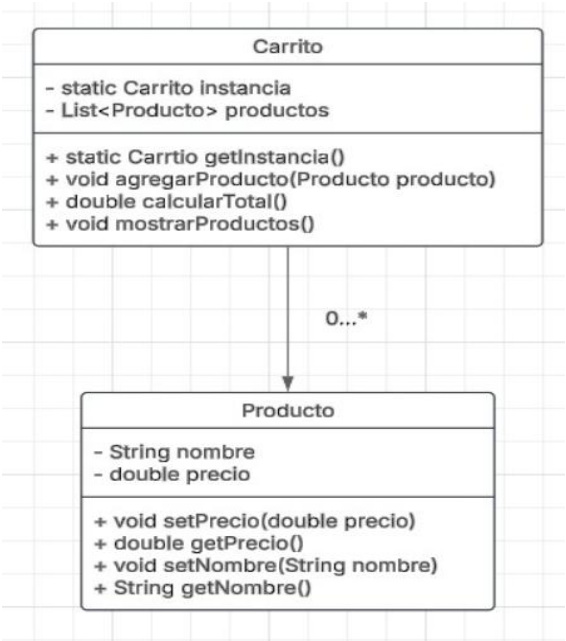
El Template Method permite definir un método base en una clase abstracta (por ejemplo, ejecutarPrograma()), que tiene la estructura del algoritmo, y permite que las subclases implementen los detalles específicos de cada paso.



2. En este ejercicio se usa el patron adapter porque las clases de medios de pago ya existen y no se pueden modificar, y la clase Pedido necesita usarlas de forma uniforme con un método común (pagarCon(medio)). El patrón Adapter permite envolver esas clases existentes y adaptarlas a la interfaz esperada, sin alterar su implementación interna.

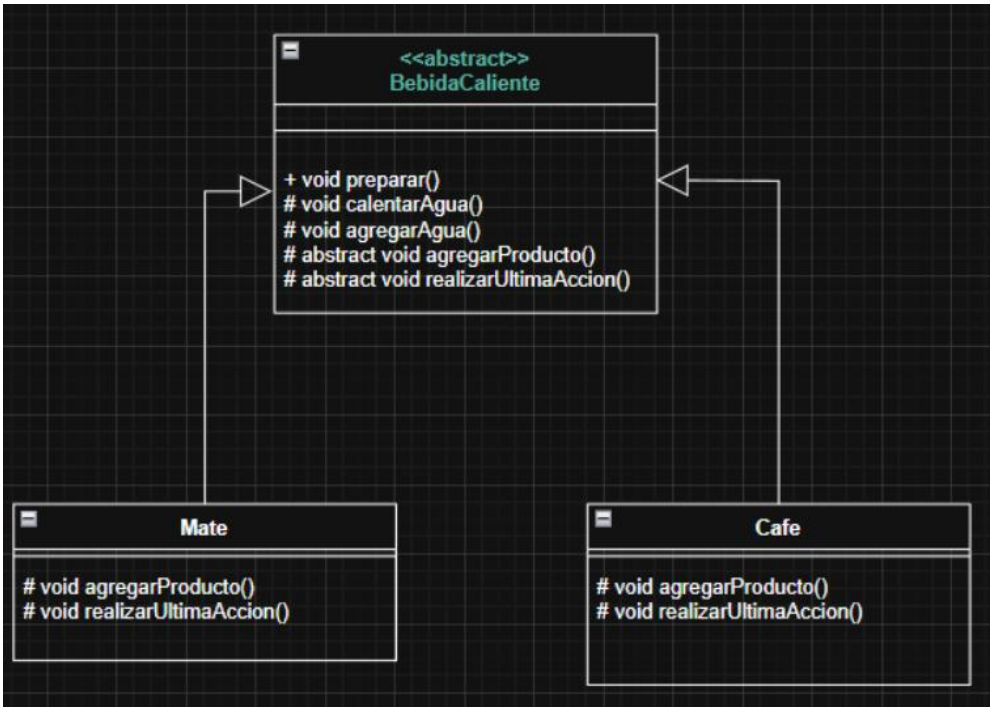


3. En este ejemplo se usa el patron singleton, debido a que en el enunciado dice que debe haber una única instancia del carrito durante toda la sesión (ademas, no tendria sentido crear otro carrito de compras). El patrón Singleton garantiza que una clase tenga una única instancia accesible globalmente y proporciona un punto de acceso centralizado a esa instancia.



4. Por ultimo, aca tambien se utiliza el patron Template Method, porque tanto al hacer un mate o un café, se siguen pasos comunes (mismo caso que en los programas del lavarropas).

A y B. (el codigo se encuentra en el github, el diagrama lo dejamos a continuacion)



C.

i. ¿Dónde se define el esqueleto del algoritmo?

El esqueleto del algoritmo se define en la clase abstracta llamada BebidaCaliente en el método preparar(), es donde se define los pasos generales de nuestro algoritmo.

ii. ¿Se puede redefinir el esqueleto?

No, no se puede redefinir, ya que, si se redefine, la estructura principal cambiaria, y ese cambio impactará en las clases que lo implementan

iii. ¿Qué es lo que se puede redefinir?

Se pueden redefinir los métodos abstractos que las subclases deben implementar.

iv. ¿Qué es inversión de control?

Es cuando la lógica general del flujo de control está en la superclase, es decir, en BebidaCaliente ya que la clase base llama a los métodos que están definidos en las subclases y no al revés.

REFACTORING:

1. Bad smell se encuentra en distintas partes en este código:

-En el primer método de la clase VideoJuego, hay un espacio entre la palabra personaje y ConMasDaño(), lo cual produce un error.

-Falta un corchete para cerrar el método imprimirInfo() y esto genera un error al compilar.

-Dentro del método imprimirInfo() hay código duplicado, ya que en vez de usar System.out.printLn se puede tener un método en cada subclase que retorne su descripción.

-double daño = p.getTipoHabilidad().calcularDaño(p.getDaño()) es una línea de código que presenta mucha confusión porque llama a muchos métodos a

la vez. Lo que se puede hacer, es crear un método en la clase personaje que realice la lógica interna de cuánto daño hace.

-En el método imprimirInfo() se puede extraer partes en métodos más pequeños ya que chequea tipos aparte de mostrar descripciones.

2. Creemos que los malos olores detectados representan un problema para la mantenibilidad y escalabilidad del software ya que:

-Hacen que el código sea difícil de entender debido a que, cada vez que se agrega un nuevo tipo de Habilidad, hay que modificar el if para contemplar ese nuevo tipo y eso va a terminar generando código repetido porque va a haber muchos else if.

-Es más difícil encontrar bugs en el programa ya que no va a tener una organización o estructura definida.

-Para las próximas personas que interactúen con el código, les va a ser muy difícil entenderlo y modificarlo.

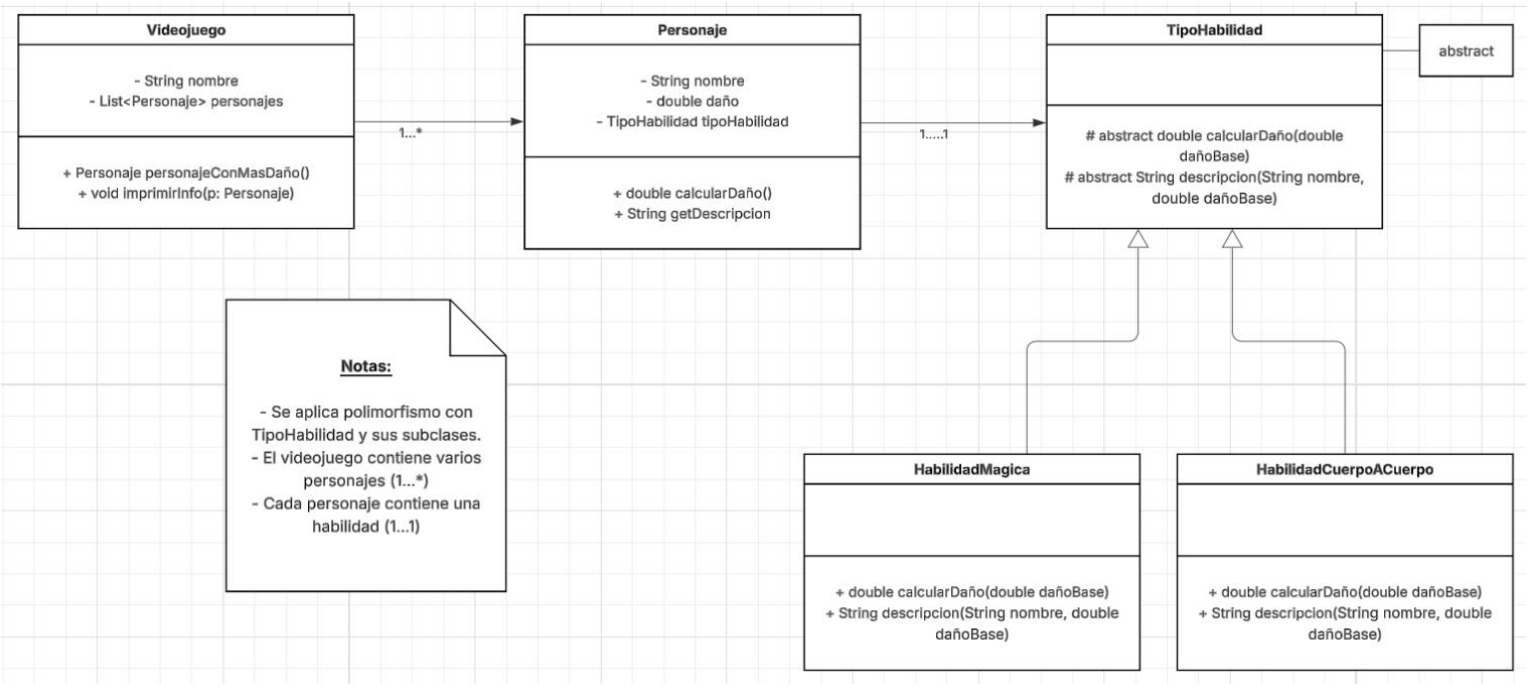
-El código mezcla responsabilidades (mostrar nombre, calcular danio, imprimir descripción). Esto hace que sea difícil de leer, mantener y extender.

3. El tipo de refactoring que se recomienda aplicar para corregir el mal olor es extract method ya que:

-Se puede extraer el código duplicado (por ejemplo, los printLn) en un método común dentro de cada clase correspondiente.

-Se puede dividir las múltiples responsabilidades que tiene un método en métodos más pequeños y comprensibles.

4. (El codigo corregido se encuentra en el github)



DOMAIN DRIVEN DESIGN:

1. ¿Qué es el lenguaje ubicuo y cómo contribuye a la alineación entre el equipo técnico y los expertos del dominio?

El **lenguaje ubicuo** es un lenguaje común, preciso, compartido y acotado del negocio que se construye entre los desarrolladores y los expertos del dominio. Hace referencia a un vocabulario que se utiliza en el código, en las reuniones, en la documentación y en las conversaciones del equipo.

Contribuye a la alineación entre el equipo técnico y los expertos del dominio de manera tal que:

- Reduce la ambigüedad: todos usan las mismas palabras con el mismo significado.
- Facilita la comunicación: se eliminan malentendidos entre los desarrolladores y los expertos del negocio.

Su objetivo es que no se presenten confusiones respecto de los términos que se utilicen, ya que una palabra puede ser percibida con distintos significados. Por ejemplo, si hablamos de transacciones, métodos de pagos, se hace referencia al pago, y este sería el lenguaje ubicuo porque se sabe de que se está hablando.

2. Diferenciá claramente una entidad de un value object en el modelo del dominio, ejemplifique.

Una **entidad** es un elemento que se caracteriza por ser identificable por un único valor o un conjunto de estos. Por ejemplo, una persona tiene dni, nombre, apellido, hobbies, etc (el dni sería un valor único). Su identidad no depende de sus atributos, ya que estos pueden cambiar pero sigue siendo la misma entidad (puede cambiar de nombre, de apellido, pero el número de dni siempre va a seguir siendo el mismo).

En cambio, un **value object** es un elemento que es identificable por el conjunto completo de su contenido y, en caso de modificar cualquier valor del mismo, se obtendría otro elemento distinto. Por ejemplo, una persona vive en una determinada dirección, la cual tiene una calle, ciudad, código postal, etc. La dirección, es definida por sus atributos y estos son inmutables (una vez creados, no se cambian). En el caso de que la persona se mude, no se cambiaría solo la calle, sino que se crea una instancia completamente nueva de su dirección.

3. Qué es un bounded context y explica por qué su correcta delimitación es clave para manejar la complejidad y la evolución de sistemas grandes.

Bounded context se refiere a los módulos donde se aplica un lenguaje ubicuo coherente. Las reglas y conceptos tienen un significado claro y consistente, pero solo dentro de ese módulo. Por ejemplo, la palabra “producto”, en el módulo de ventas, puede referirse a lo que el cliente ve y compra, o, en el módulo de logística, puede referirse a lo que se mueve dentro del almacén, o en el módulo de facturación, puede referirse a lo que se factura legalmente. El significado de esta palabra cambia según su contexto.

Su correcta delimitación es clave para manejar la complejidad y la evolución de sistemas grandes ya que:

- Reduce la complejidad del sistema (si surge un error, será en un módulo específico y no en todo el sistema, lo cual facilita encontrarlo y arreglarlo).
- Evita malentendidos.
- Organiza equipos y responsabilidades de forma clara.
- Hace que el sistema sea más escalable, mantenible y modular.

4. ¿Cómo ayudan los diagramas UML (como diagramas de clases, secuencia y casos de uso) a definir y comunicar el modelo del dominio? ¿De qué manera favorecen el lenguaje común entre desarrolladores y expertos del negocio?

Los diagramas UML ayudan a definir y comunicar el modelo de dominio de varias maneras:

- Clarifican conceptos y relaciones: representa visualmente la entidades del dominio, sus atributos, métodos y relaciones.Por ejemplo, es mucho más claro y preciso ver una relación de herencia en un diagrama de clases que explicarlo solo con texto.
- Prevén errores: ayudan a detectar confusiones tempranas (que luego se verían reflejadas en el código y su desarrollo).

Favorecen el lenguaje común entre desarrolladores y expertos del negocio de manera tal que los diagramas UML permiten comunicar ideas entre roles distintos; el equipo técnico (devs) y el funcional (negocio) pueden usar los mismo diagramas como base para las conversaciones. También estos se convierten en una herramienta visual del lenguaje ubicuo.