IIC 2143 - Ingeniería de Software

UML

M. Trinidad Vargas
mtvargas1@uc.cl

Diseño de Software

El diseño de software es una actividad creativa en la que identifica los componentes de software y sus relaciones, según los requerimientos del cliente.

Ian Sommerville,

Software Engineering, 10 ed.

Diseño de Software

Un **buen diseño** es

- fácil de entender
- fácil de modificar y extender
- fácil de reutilizar en otro problema
- fácil de testear la implementación
- fácil integrar las distintas unidades
- fácil de implementar (programar)

Unified Modeling Language UML

- Es un lenguaje de modelado de sistemas
- Permite visualizar, especificar, construir y documentar un sistema o proceso.
- Ofrece un estándar para describir un plano del sistema, de tal forma que sea comprensible para los desarrolladores, gestores de proyectos, etc.

UML

Ventajas

- Simplifica las complejidades.
- Facilita establecer líneas de comunicación.
- Facilita automatizar la producción de software y los procesos.
- Ayuda a resolver los problemas arquitectónicos constantes.
- Aumenta la calidad del trabajo.

UML

¿Para qué lo usamos actualmente?

- Diagramas de clase
- Diagramas de secuencia

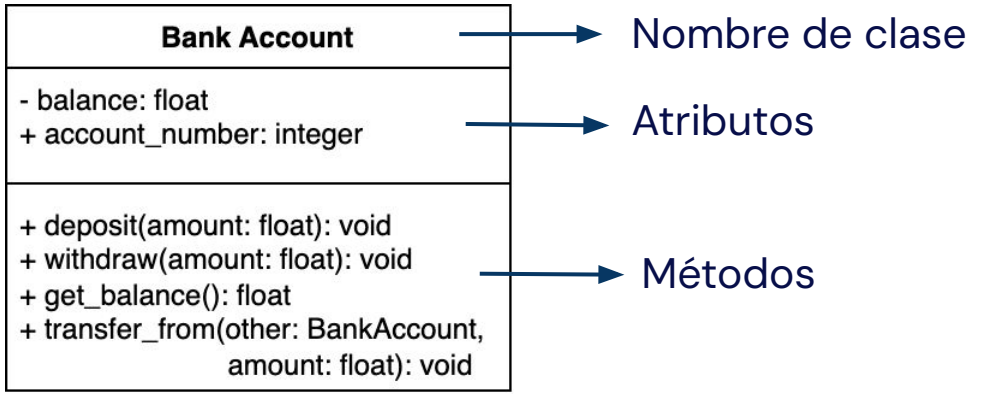
Más información en https://www.uml-diagrams.org/

Diagramas de clase

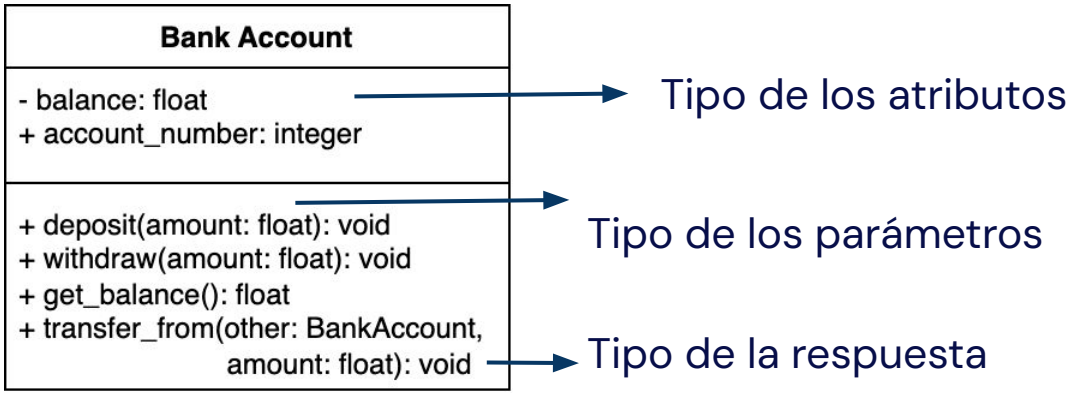
Representa el diseño lógico y físico de un sistema, y muestra sus clases.

Este diagrama ofrece una imagen de las diferentes clases y la forma en la que se interrelacionan.

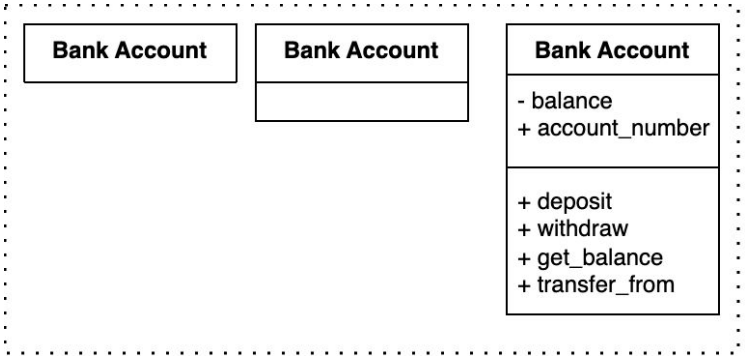
Las clases se representan con cuadros y sus asociaciones o dependencias con líneas.



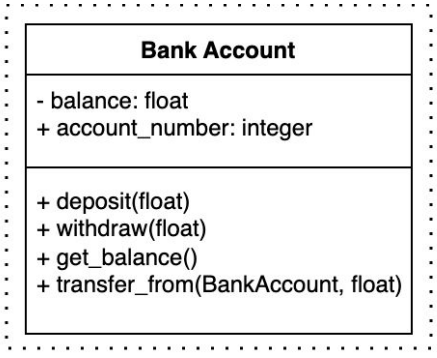
Diagramas de clase



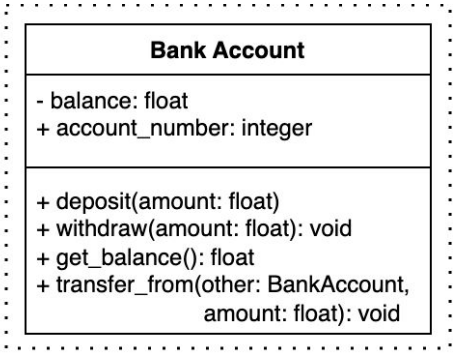
Niveles de detalle



Conceptual



Especificación

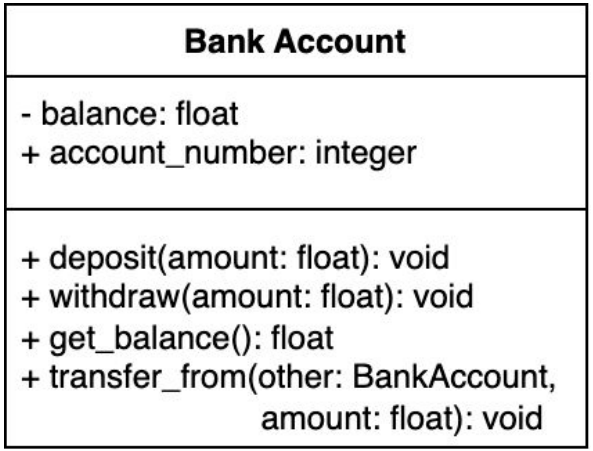


Implementación

Visibilidad

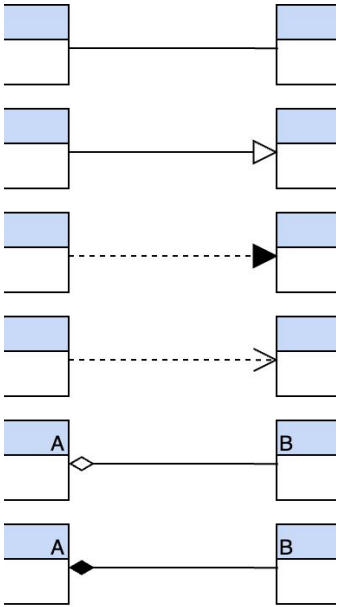
Se indica el control de acceso de atributos y métodos usando símbolos

- **público (+)** accedido desde cualquier clase del sistema
- **protegido (#)** solo accedido desde la misma clase y subclases
- **privado (-)** solo accedido desde la misma clase



Relaciones entre clases

A través de líneas representamos la forma en que se relacionan las clases



Asociación

Generalización (herencia)

Realización

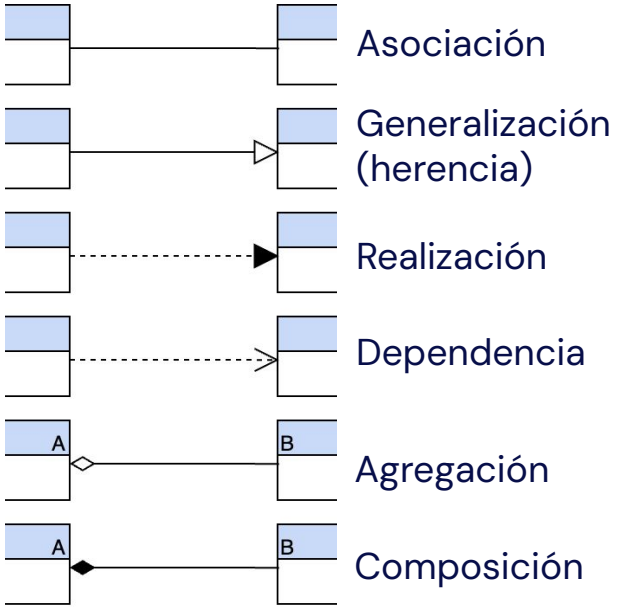
Dependencia

Agregación

Composición

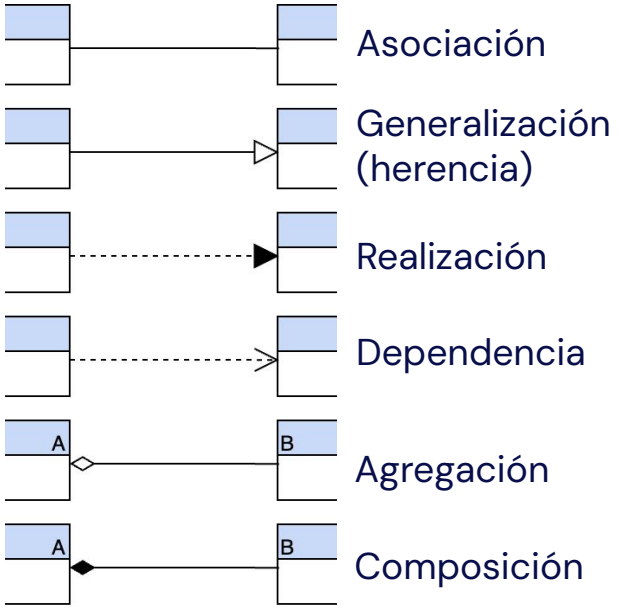
Relaciones entre clases

- **Association:** Un objeto de la clase tiene una referencia a un objeto de otra clase por mucho tiempo. Por ejemplo, cómo atributo.
- **Inheritance:** Una clase hereda de otra.
- **Realization:** Una clase implementa una interfaz o módulo (en Ruby).
- **Dependency:** Un objeto de la clase tiene una referencia a otro objeto de otra clase de forma temporal. Por ejemplo, como variable temporal o como argumento.

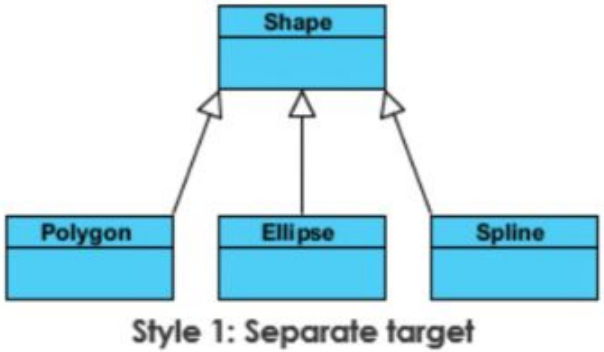


Relaciones entre clases

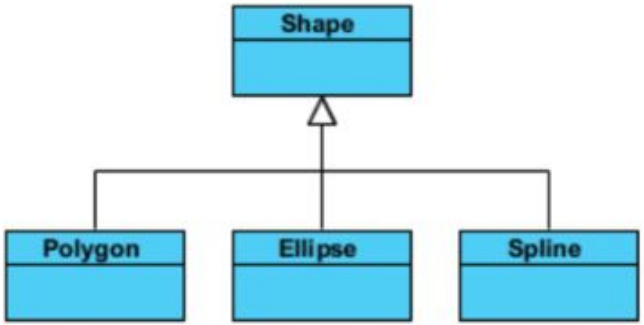
- **Aggregation:** Un objeto de una clase A está compuesto de objetos de otra clase B. Los objetos de B no son creados dentro de la clase A.
- **Composition:** Un objeto de la clase A está A compuesto de objetos de otra clase B. Los objetos de la clase B son creados dentro de A, causando una dependencia más fuerte.



Herencia



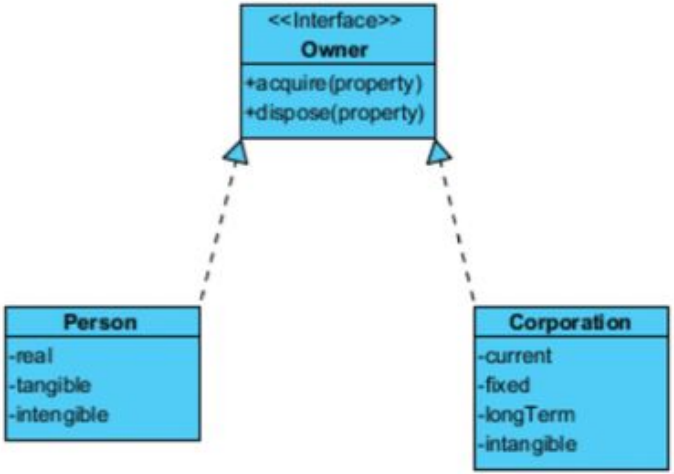
Style 1: Separate target



Style 2: Shared target

Se puede utilizar cualquiera de los dos estilos de forma indiferente.

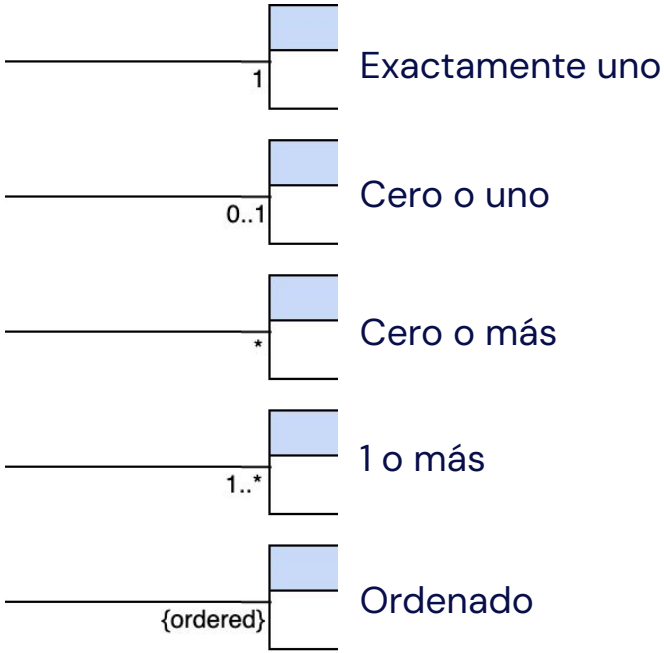
Realización



En Ruby, el equivalente a una interfaz es un módulo.

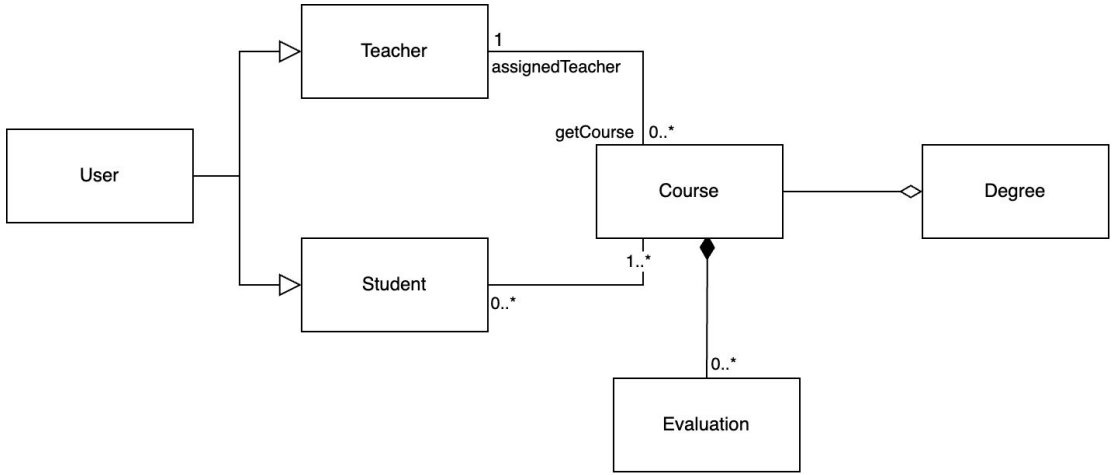
Cardinalidad

Es posible especificar la cardinalidad de la relación entre clases



Ejemplo diagrama de clases

- ¿Por qué tenemos asociaciones direccionales y bidireccionales?
- ¿Cuales son las cardinalidades faltantes?



Las relaciones dependen de los requerimientos

Diagrama de secuencia

Permite modelar las interacciones entre objetos de un sistema cronológicamente durante un caso de uso en particular.

Son útiles para observar el comportamiento de varios objetos y su interacción.

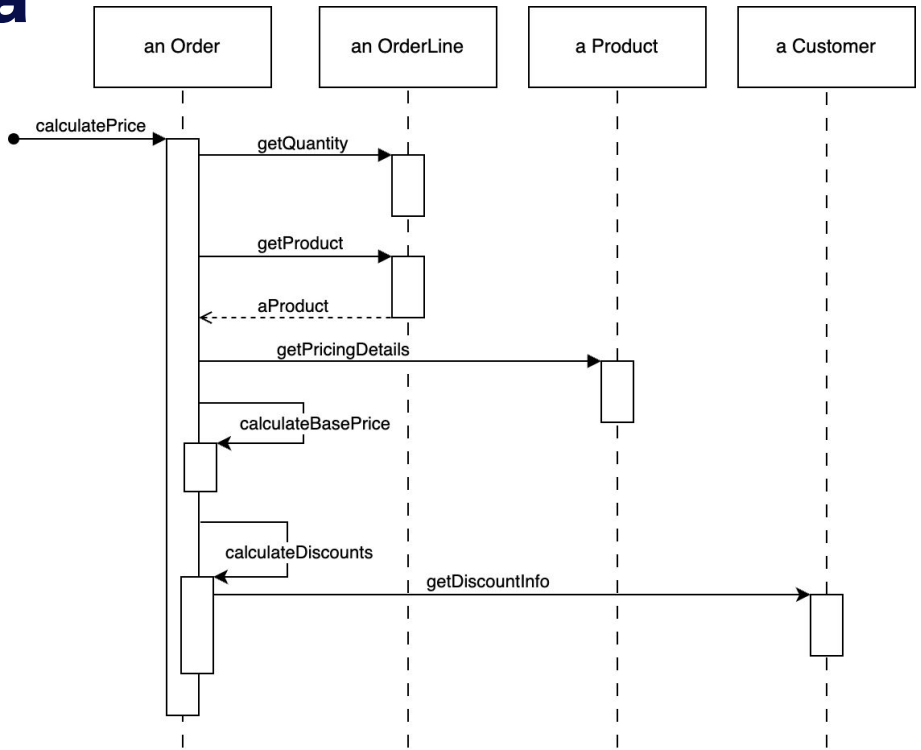


Diagrama de secuencia

- Cada caja es la instancia de una clase (objeto).
- La línea punteada es la línea de vida de un objeto.
- Se lee de arriba hacia abajo
- Las interacciones entre los objetos se indican con flechas dirigidas

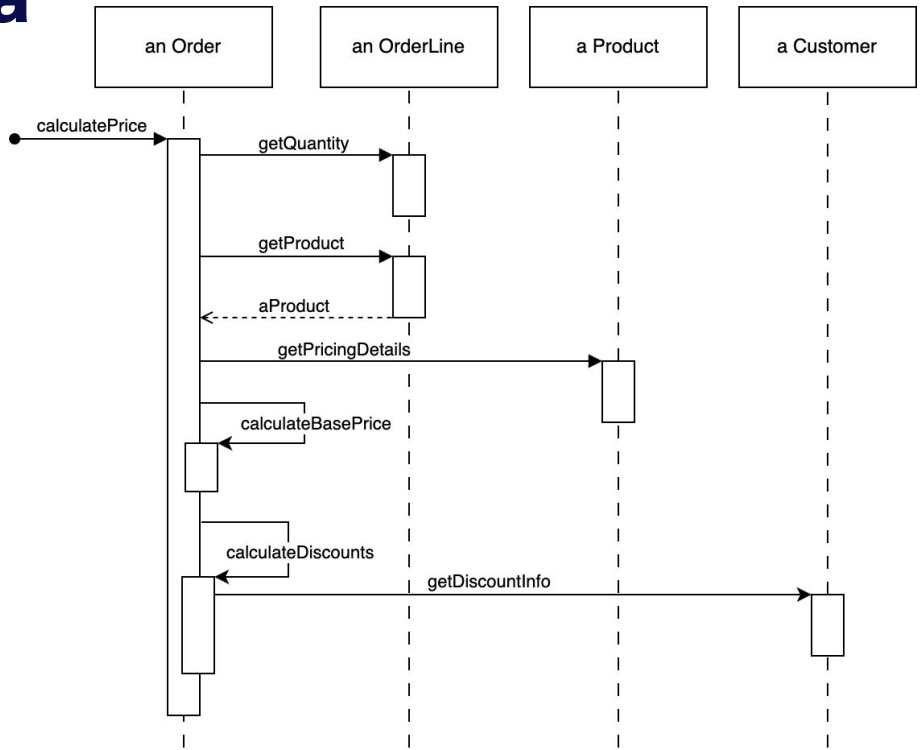


Diagrama de secuencia

```
class Order
 def calculatePrice()
   quantity = orderLine.getQuantity()
   product = orderLine.getProduct()
   product.getPricingDetails()
   self calculateBasePrice()
   self calculateDiscount()
   customer.getDiscountInfo
 end
end
```

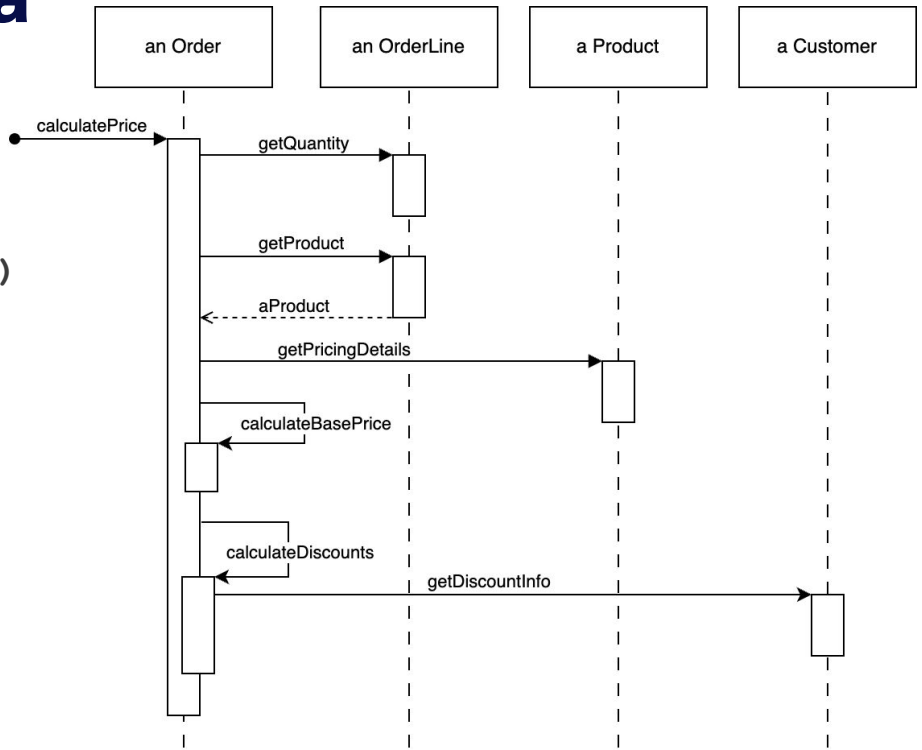
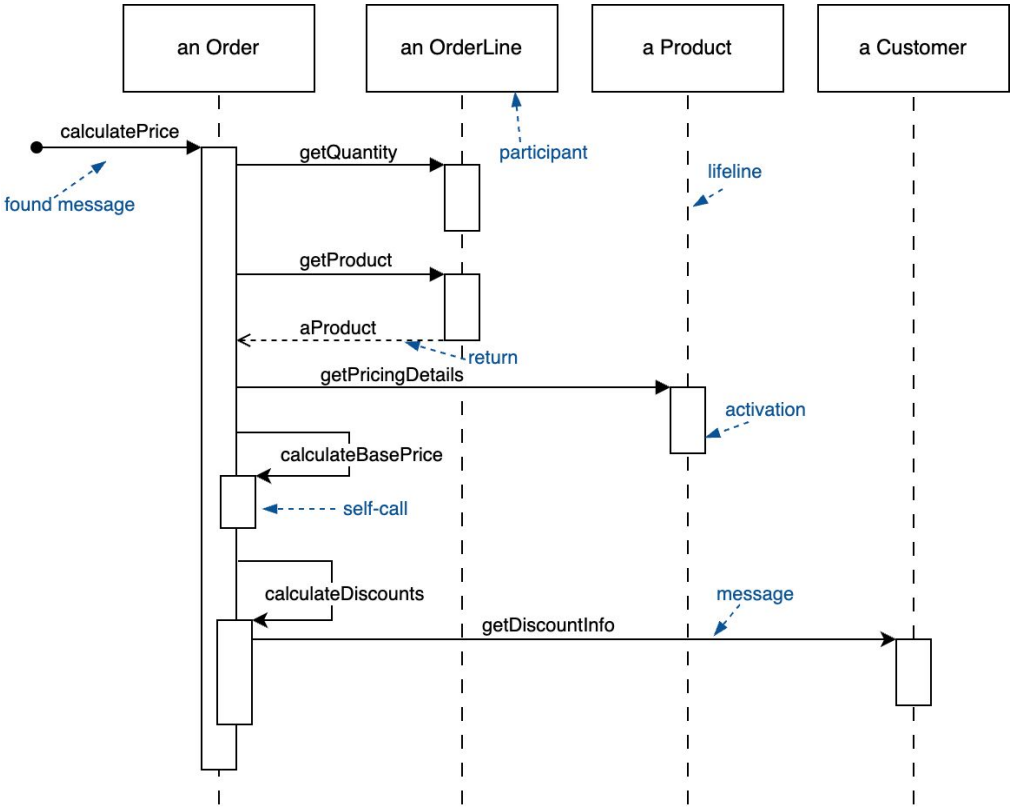
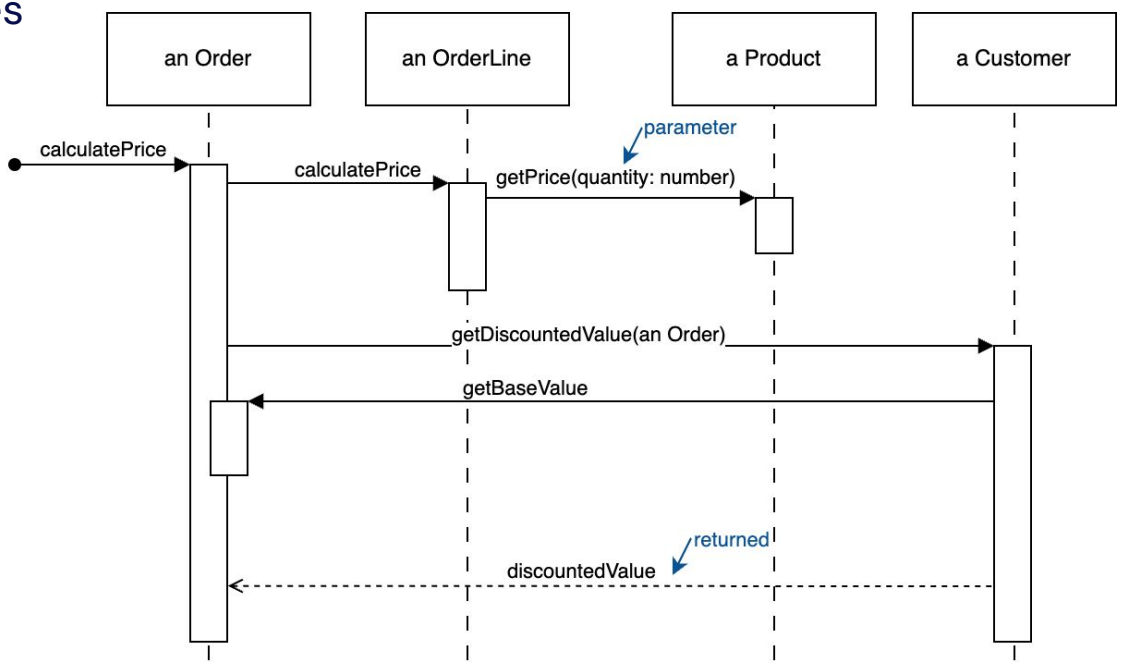


Diagrama de secuencia



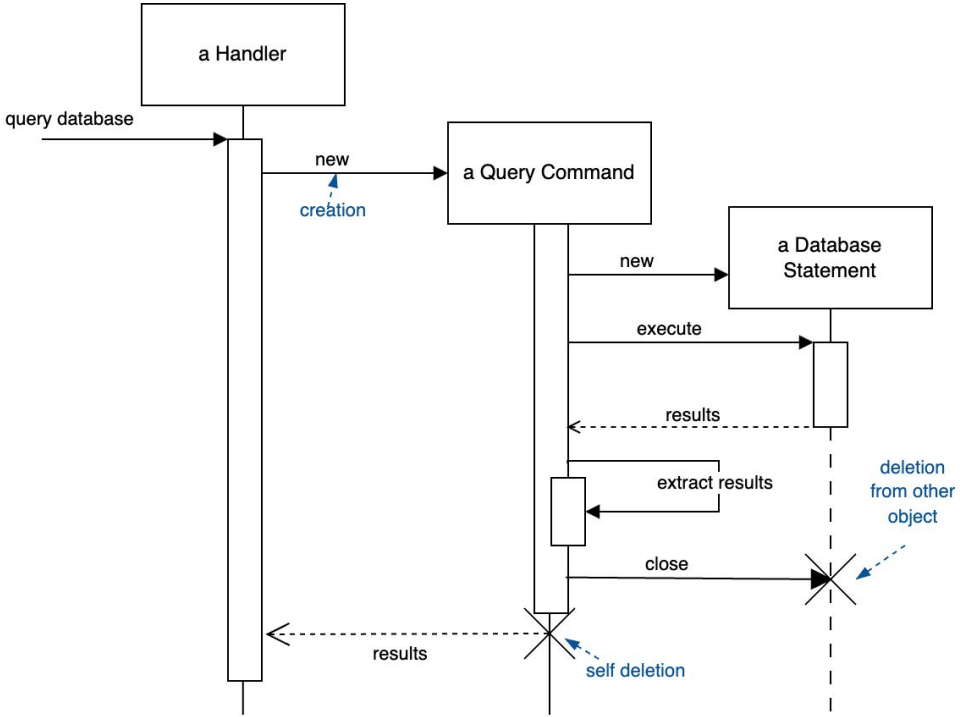
Argumentos y valores retornados

Es posible especificar los argumentos y valores retornados correspondiente a las interacciones



Creación y eliminación de objetos

Es posible especificar cuando se crean o eliminan algunos objetos del diagrama

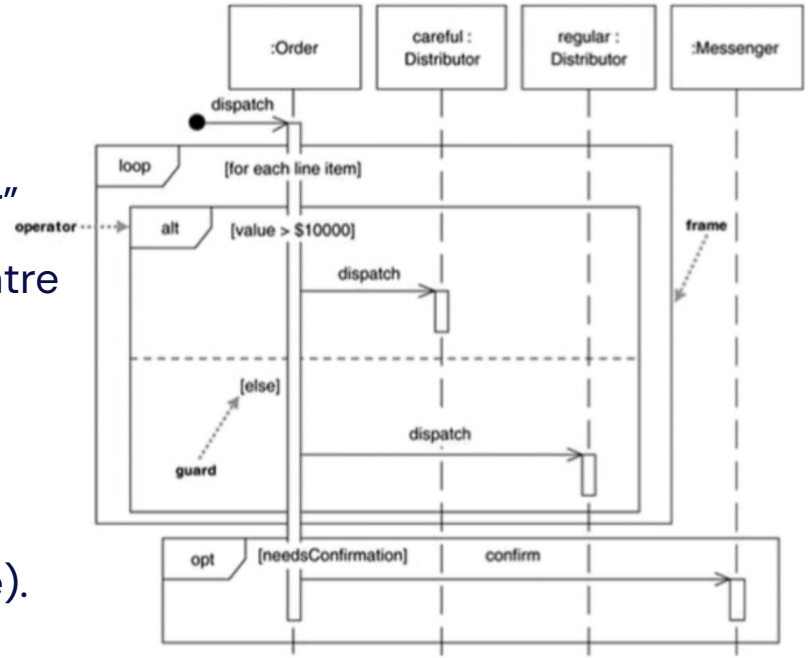


Iteraciones y condicionales

Se utilizan operadores para mostrar interacciones más complejas

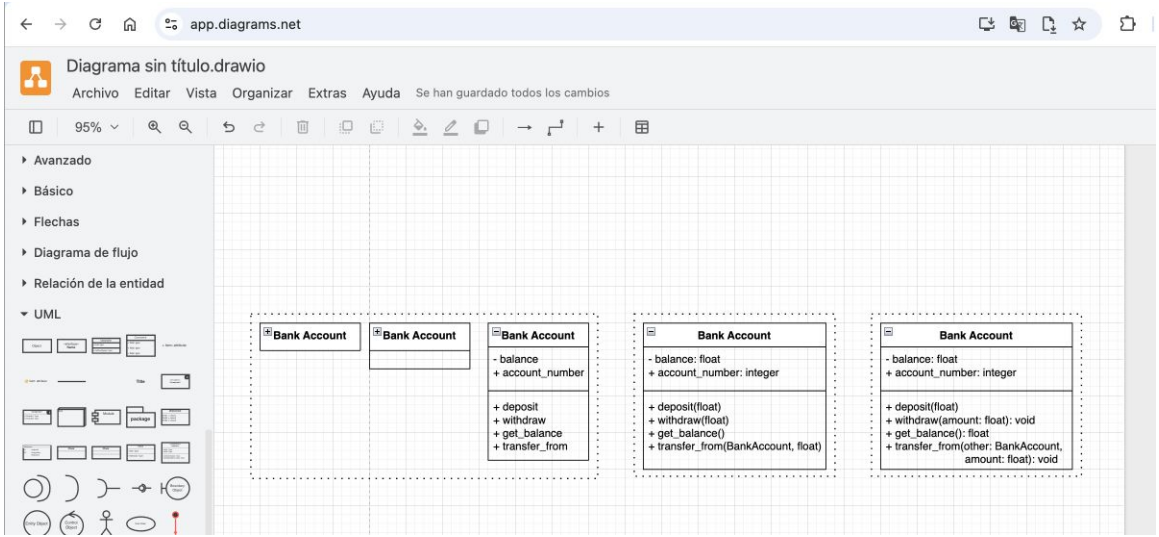
- es posible usar “an order” o “order1: Order”
- **alt:** Condicional, donde la condición va entre corchetes (if-then-else)
- **loop:** hace referencia a un for loop
- **opt:** Fragmento opcional que se ejecuta cuando la condición se cumple (if sin else).

Se usan marcos (frames) para denotar las interacciones dentro del if o for



Herramientas para diagramar

- https://www.drawio.com
- https://www.lucidchart.com/pages

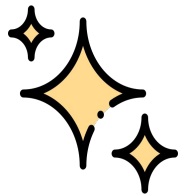


Ejercicio con décima

Realiza el diagrama **de clase y de secuencia** de alguno de estos problemas u otro de similar complejidad.

1. Inscripción de clases en un gimnasio (ClassPass)
2. Agregar un evento en un planificador de viajes
3. Transferir de una cuenta bancaria a otra

Recuerda agregar los métodos que validan los procesos



Ejercicio con décima

Realiza el diagrama **de clase y de secuencia** de alguno de estos problemas u otro de similar complejidad.

Entregable

Un archivo .pdf o .png legible

Se puede realizar solos o hasta 3 personas entregando **un solo archivo**

Pueden realizar el diagrama en draw.io