Orientación a objetos 1

ÍNDICE

MAEVEN

Es un arquetipo para transformar código en una aplicación. Un **arquetipo** define un modelo a partir del cual se crean otros elementos

PROGRAMACIÓN OO

En un **software construido con objetos,** un conjunto de objetos colabora enviándose mensajes. El éxito de este paradigma es poder agregar funcionalidades, reemplazar o modificar objetos sin que el sistema se rompa

Por lo tanto, los objetos deben conocer sus propiedades, conocer otros objetos y llevar a cabo acciones.

No hay un *main* ya que nadie es más importante que otro y cooperan entre sí

A tener en cuenta

- ✓ Mientras que la estructura sintáctica es "lineal", a la hora de ejecutarlo, no lo es
- ✓ Pensamos en qué cosas hay en nuestro software y cómo se van a comunicar entre sí
- ✓ La estructura general cambia, no hay una jerarquía de main, procedures, etc.

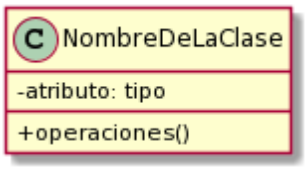
Cuando programo en objetos, pienso mi programa en base a conceptos del dominio de la aplicación, así se consiguen programas compuestos de módulos poco acopados y enfocados en un objetivo claro y esto resuelta en *programas más fáciles de entender, mantener y extender*

UML

El Lenguaje Unificado de Modelado (UML) muestra las clases del sistema y relaciones. Describe la vista estática de un sistema.

CLASES

Se denotan con 3 compartimientos, indicando: nombre, atributos y métodos



Nombre de la clase: debe tener un nombre en singular, en caso de ser una clase abstracta, se debe indicar antes del nombre la etiqueta <<abstract>>

Atributos: para cada atributo tengo que indicar su visibilidad (privada – o protegida #) y su tipo: integer, real, boolean, string

Operaciones: los parámetros deben tener nombre y tipo y deben separarse con comas, tengo que indicar el tipo de retorno y en caso que no devuelve nada, no especifico retorno

-En el caso que la operación retorne una colección, debe indicarse la multiplicidad de la forma: `+ obtenerOfertasDelDia (): Oferta [*]`

- En el caso que las operaciones sean abstractas, se deben anotar en cursiva o con el estereotipo <<abstract>>.

```
+ calcularSueldo (): Real
```

```
« abstract » + calcularSueldo (): Real
```

- Si se trata de un constructor, debe ser precedido por el estereotipo `<<create>> + <<create>> createCar(brand: String) : Car`

Visibilidad:

- ✓ Se utiliza visibilidad pública (+) cuando el miembro es accesible a todos los objetos del sistema.
- ✓ Se utiliza visibilidad protegida (#) cuando el miembro es accesible a las instancias de la clase que lo implementa y de sus subclases.

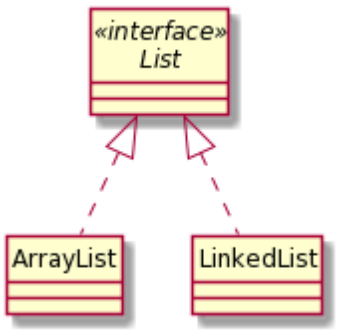
- ✓ Se utiliza visibilidad privada (-) cuando el miembro es sólo accesible a las instancias de la clase que lo implementa.

ASOCIACIONES

Si tenemos una asociación entre dos clases, se debe indicar en los extremos navegables:

- ✓ El nombre del rol.
- ✓ La multiplicidad, salvo que sea 1. Puede ser un número fijo, por ejemplo 4, o puede indicarse que se permiten varios elementos, en ese caso se indica como 0..*
- ✓ Si la asociación es navegable para uno de los extremos, se debe indicar con la punta de flecha, la clase destino.
- ✓ Si la asociación es navegable para los dos extremos, no se debe dibujar la punta de la flecha.

INTERFACES



La interfaz se denota con el estereotipo <<interface>>, el nombre debe denotarse en cursiva, los métodos de la interface son públicos y abstractos y la relación con la clase que la implementa, se representa con una flecha sin relleno y línea punteada, apuntando en dirección a la interfaz

OBJETOS

Un objeto se caracteriza por tener una **identidad** (se distingue de otro), **conocimiento** (en base a sus relaciones con otros objetos y estado interno), **comportamiento** (conjunto de mensajes que un objeto sabe responder

Los objetos se parecen a registros pero están encapsulados, contienen comportamiento (qué cosa se les puede pedir) y conocen a otros objetos mediante referencias y no mediante datos.

ESTADO INTERNO

El estado interno determina su conocimiento, el cual está dado por propiedades básicas del objeto y otros objetos con los cuales colabora para llevar a cabo sus responsabilidades

MÉTODOS

Indican qué sabe hacer el objeto y cuáles son sus responsabilidades.

La realización de cada mensaje, o sea la manera en que un objeto responde a un mensaje se especifica mediante un *método.* Entonces, cuando un objeto recibe un mensaje, responde activando el método asociado. El que envía el mensaje delega en el receptor la manera de resolverlo, que es privada del objeto

ENCAPSULAMIENTO

Cualidad de los objetos de ocultar los detalles de implementación y su estado interno del mundo exterior.

Es importante:

- ✓ Esconder detalles de implementación
- ✓ Proteger el estado interno de los objetos
- ✓ Un objeto solo muestra su cara visible por medio de su protocolo
- ✓ Los métodos y su estado quedan escondidos, el objeto decide qué se publica
- ✓ Facilidad de modularidad y reutilización

CLASES

Descripción abstracta de un conjunto de objetos que comparten los mismos atributos (estado interno de los objetos de la clase), operaciones (mensaje que responden los objetos), relaciones y semántica.

Cumplen 3 roles: agrupar el comportamiento común a sus instancias, definir la forma de sus instancias, crear objetos que son instancias de ellas

En consecuencia, todas las instancias de una clase se comportan de la misma manera. Cada instancia mantendrá su propio estado interno.

Instanciación: mecanismo de creación de objetos, la clase funciona como molde y un nuevo objeto es una instancia de una clase

Inicialización: para que un objeto esté listo para llevar a cabo sus responsabilidades, hay que inicializarlo, o sea darles valores a sus variables

Visibilidad de los métodos

- ✓ Publica (+) → atributo que puede ser visto y usado por fuera de la clase
- ✓ Privada (-) → atributo no puede ser accedido por otras clases
- ✓ Protegido (#) → privado pero visible a las subclases de la clase donde está declarado

HERENCIA

Mecanismo por el cual las subclases reúsan el comportamiento y estructura de su superclase. Permite crear una nueva clase como refinamiento de otra.

Herencia de comportamiento → una subclase hereda todos los métodos definidos en su superclase, a su vez las subclases pueden redefinir el comportamiento heredado mediante *@override*

Herencia de estructura → no hay forma de restringirla

CLASES ABSTRACTAS

Es una clase que no puede ser instanciada. Representan conceptos abstractos. Sirven para factorizar comportamiento común de sus subclases, las cuales completan las piezas faltantes o agregan variaciones

POLIMORFISMO

Un mismo mensaje puede ser enviado a diferentes objetos y distintos receptores pueden reaccionar diferente (tener diferentes métodos)

POLIMORFISMO + BINDING DINÁMICO

El binding es la asociación de operadores y operando. Se usa siempre al compilar el código. Tiene que ver con el código, la relación entre el mensaje y la variable, no se relaciona con la instancia.

Binding procedural/estático → se hace en tiempo de compilación y queda fijo porque es procedural y no hay otros.

Binding dinámico → no sabemos a qué clase va a pertenecer el objeto. Decimos que es dinámico porque recién en ejecución se determina cuál es el código que se ejecuta

INTERFACES

Medio común para que los objetos no relacionados se comuniquen entre sí. Estas son definiciones de métodos y valores sobre los cuales los objetos están de acuerdo para cooperar.

TIPOS

Tipo es un conjunto de firmas de operaciones/métodos (nombre, tipo de argumento, tipo de resultados). Decimos que un objeto es 'de un tipo' si ofrece el conjunto de operaciones definido por ese tipo

PRIMITIVOS

BYTE	INT	FLOAT	BOOLEAN	SHORT	LONG	DOUBLE	CHAR
0 como valor por defecto. Representa datos de 8 bits con signo, entonces almacena de -128 a 127	0 como valor por defecto. Representa 32 bits. EL valor mínimo es -2^{31} a 2^{31} -1	0.0f por defecto. Almacena números con coma flotante con precisión simple de 32 bits	False como valor por defecto. Ocupa 1 bit	0 por defecto. Representa datos de 16 bits con signo. Va del -32.768 a 32.767	Defecto 0L. Datos de 64 bits con signo. Desde el -2^{63} a 2^{63} -1	Defecto 0.0d Almacena números con coma flotante con doble precisión de 64 bits	U0000 por defecto. Representa un carácter Unicode sencillo de 16 bits

THIS

Permite que un objeto se envíe un mensaje a si mismo. La utilidad del *.this* es reutilizar comportamientos repetidos, aprovechar el comportamiento heredado y descomponer los métodos largos

MÉTODO LOOKUP

Cuándo this recibe un mensaje, la búsqueda de métodos comienza en la clase de la cuál this es instancia (sin importar dónde está el método en el que se le envía el mensaje)

SUPER

Cuándo un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje. En un lenguaje dinámico podría no encontrarlo (error en tiempo de ejecución), pero en un lenguaje con tipado estático sabemos que lo entenderá (aunque no sabemos qué hará)

Cuándo super recibe un mensaje, la búsqueda de métodos comienza en la clase inmediata superior a aquel dónde está definido el método que envía el mensaje (sin importar la clase del receptor)

COLECCIONES

Su rol principal es mantener relaciones entre objetos. De las cosas más importantes en las colecciones es que nunca debemos modificar una colección que obtenemos de otro objeto porque cada objeto es responsable de mantener las invariantes de sus colecciones. Solo el dueño de la colección puede modificarla.

LIBRERÍAS Y FRAMEWORKS DE COLECCIONES

Algunas operaciones que permiten los frameworks es: ordenar, recorrer, encontrar elemento, filtrar, recolectar y reducir

ORDENANDO COLECCIONES

Para ordenar colecciones tenemos un *comparador*. Usamos un objeto comparador, que implementa comparator, que es una interfaz de la librería de java. Los comparadores implementan un método principal, que es el compare, le damos un parámetro y nos devuelve un número (0 = son iguales , - = el primero es menor, + = el primer parámetro es mayor que el objeto 2).

RECORRIENDO COLECCIONES

Recorrer colecciones es algo frecuente, para lo cual si se hace así sin una operación especial involucra más de un ciclo for. El loop de control es un lugar más dónde cometer errores, además que el código es repetitivo y queda atado a la estructura o tipo de la colección (no se puede hacer en un set, porque es en algo indexado). Así como para ordenar las colecciones tenemos un comparador, para recorrerlas tenemos una herramienta: el iterador.

El iterador es una interface. Todas las colecciones entienden iterator() ,que encapsula: el cómo recorrer una colección particular y el estado de un recorrido. No nos interesa la clase de iterador, ya que son polimórficos.

TIPOS DE COLECCIONES

- ✓ List (java.util.List) → elementos indexados por enteros de 0 en adelante
- ✓ Set (java.util.Set) → no admite duplicados, sus elementos no están indexados y es ideal para chequear pertenencia
- ✓ Map (java.util.Map) → asocia objetos que actúan como claves a otros que actúan como valores
- ✓ Queue (java.util.Queue) → maneja el orden en que se recuperan los objetos (LIFO, FIFO, por prioridad, etc)

STREAMS

Encapsulan colecciones y nos dejan hacer operaciones con las mismas. Para esto vamos a usar: expresiones Lambda (métodos anónimos (no tienen nombre, no pertenecen a ninguna clase))

Los streams no almacenan si no que proveen acceso a una fuente, cada operación produce un resultado pero no modifica la fuente, potencialmente no tiene final y es consumible, o sea que cada elemento se visita una sola vez

ALGUNAS OPERACIONES CON STREAMS

- ✓ Filter(): retorna un nuevo stream que solo deja pasar los elementos que cumplen con cierta condición
- ✓ Map(): nos da un stream que transforma cada elemento de entrada aplicando una función que indiquemos
- ✓ Collect(): es una operación terminal

CRITERIOS DE DISEÑO

COMPROBACIONES BÁSICAS

- ✓ La sintaxis debe ser **objeto-mensaje** o sea que siempre hay un receptor del mensaje
- ✓ No olvidar retornar explícitamente un objeto cuando eso es lo que se espera
- ✓ No se puede acceder directamente a las variables de instancia de otros objetos ya que el ocultamiento de información es uno de los principios más importantes de la POO
- ✓ No confundir mensajes de clase con mensajes de instancia
- ✓ No se pueden acceder a las variables de instancia desde los métodos de clase

MALOS OLORES DE DISEÑO

Envidia de atributos → soy un objeto que le pido cosas a otros objetos para hacer algo, para evitar esto la tara la debe hacer el objeto que tiene las cosas que se necesitan

Clase Dios → una clase hace todo y las demás están anémicas, para evitarlo tengo que ver qué otros objetos podrían hacer aparecer, que se puedan encargar de alguna de las responsabilidades de este

Código duplicado → no tengo que hacer Ctrl+C Ctrl+V, tengo que pensar si es un método que puedo generalizar y heredarlo para reutilizarlo

Clase larga → tengo que pensar si la clase larga puede delegar cosas a otros objetos que conoce

Método largo → si tiene más de 10 líneas o debo incluir comentarios, es mala señal. Tengo que identificar dentro del método, partes que podría considerar comportamientos individuales, llevar cada parte a un nuevo método y cuando necesite llevar a cabo uno de esos comportamientos, enviar mensajes a self

Objetos que conocen el id de otro → nunca relacionar objetos por medio de claves, cuando un objeto se relaciona con otro, lo hace con una referencia

Switch statements → tengo que sentir mal olor cuando veo que se una un *if, case, switch* o *if anidados* para determinar de qué forma se resuelve algo, para evitarlo tengo que aplicar **polimorfismo**

Romper encapsulamiento → es algo muy malo porque hace perder las ventajas de la POO, entonces tengo que agregar setters y getters cuando sea necesario, nunca modificar una colección que no es nuestra y delegar tareas a los que tienen la información necesaria

Clase de datos o clase anémica → una clase que parece un registro de datos debería dar mala señal

No es-un → una relación de herencia siempre debe respetar el principio de *es-un*

No quiero mi herencia → cuando encontramos un método que redefine a uno heredado y hace algo totalmente diferente, hay que desconfiar

Programo un constructor usando varios setters → cuando programo el constructor, debo evitar usar un método setter por cada una de las variables de instancia que debo inicializar y en cambio programar un método de inicialización que reciba como parámetro todas las variables

ESTILO DE PROGRAMACIÓN

Algunos de los estilos y buenas prácticas recomendables para implementar

Inicializa los objetos de manera explícita→ siempre que sea posible incluir un método que inicialice aquellos atributos que pueden tomar valores preestablecidos

Ofrecer constructores

Nombre de mensaje que revela la intención → que el nombre del mensaje comunique lo que quiere hacer

Delegación a self → permite descomponer un método en partes que le mismo objeto resuelve. Cada método hace una cosa, su nombre indica lo que hace, quedan cortos y permite que una subclase redefina solo un paso

Métodos cortos→ es lo más preferible, para lograrlo se usa la delegación self

Cada cosa se hace una sola vez → hay que aprender el protocolo de colecciones y objetos frecuentemente usados

Los nombres de las variables deben indicar su rol → elegir los nombres de las variables para que quede claro qué rol cumplen en el método/clase, los cuales siempre empiezan con minúscula

Pensar bien los nombres de las clases → siempre iniciar con mayúscula y singular, pueden ser nombres largos y varias palabras

TESTS

Sirve para asegurarse que el programa está libre de errores, o sea que hace o que se espera. Se testea para encontrar errores antes que los encuentre el usuario. Es recomendable cuando antes posible hacerlo, y es una tarea correspondiente a los programadores y/o a las personas que testean

A veces podemos testear mal y es porque:

- ✓ Lo dejamos para el final
- ✓ Hay muchas combinaciones que considerar
- ✓ Requieren planificación, preparación y recursos adicionales
- ✓ Es una tarea repetitiva y poco interesante
- ✓ Creemos que es tarea de otro o que alcanza con “programar bien”

TIPOS

Test funcionales → qué hace el sistema

Test de unidad → aseguran que la unidad mínima de nuestro programa funciona correctamente y aislada de otras unidades

Test automatizados → se usa software para guiar la ejecución de los test y controlar los resultados. Para esto se usa jUnit, el cual es un framework de Java para automatizar los test de unidad

Algunos otros son test no funcionales (cómo lo hace el sistema), de integración (el sistema en general), de regresión, de punta a punta, de carga, de performance, de aceptación (involucra a los usuarios finales), de UI (involucra a los usuarios finales), de accesibilidad, Alpha y beta, test A/B

DISEÑANDO TESTS

Debemos elegir una estrategia y pensar qué podría variar y que pueda causar un error y luego elegir valores de prueba para maximizar las chances de encontrar errores haciendo la menor cantidad de pruebas posibles. Para eso, elegimos dos estrategias

TEST DE PARTICIONES EQUIVALENTES

Una partición equivalente es el conjunto de casos que prueban lo mismo o revelan el mismo bug. Asumo que si un ejemplo de una partición pasa el test, los otros también lo harán, por lo cual elijo uno.

TEST DE VALORES BORDES

Los errores ocurren con frecuencia en los límites y ahí es donde los vamos a buscar. Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores. Buscar los bordes en propiedades del etilo: velocidad, cantidad, posición, tamaño. Si se trata de valores en un rango, tomo un caso dentro del rango y uno por fuera de cada lado del rango. Si se trata de casos en un conjunto tomo un caso que pertenezca a dicho conjunto y otro que no.

CUESTIONARIOS DE PROMOCIÓN

PRIMER CUESTIONARIO

- ✓ En un programa construido con objetos no hay un objeto más importante que otros. El comienzo de una aplicación depende del flujo de control, de decisiones del desarrollador, del tipo de interacción, etc.
- ✓ Creamos clases para representar la estructura y comportamiento de todos los objetos que son instancias de la clase
- ✓ Para poder funcionar los objetos conocen a otros objetos a los que pueden enviarle mensajes usando el protocolo que dichos objetos exhiben
- ✓ Cuando desarrollamos software con el paradigma de objetos, nuestros programas se pueden ver como un conjunto de objetos que colaboran entre sí enviándose mensajes

SEGUNDO CUESTIONARIO

- ✓ Un tipo de lenguaje orientado a objetos es un conjunto de firmas de métodos
- ✓ Supongamos una jerarquía con clase raíz A, subclases B,C y D. Supongamos que B tiene subclases B1, B2 y C tiene C1, C2 y D tiene D1 y D2. Cuando un objeto o de la clase B1 recibe un mensaje m(), si no se encuentra el método m en su clase, entonces busca en la jerarquía de clases primero en B. Si lo encuentra lo ejecuta como si fuera propio. Si no lo encuentra lo busca en A. Si lo encuentra lo ejecuta como si fuera propio
- ✓ Cuando un objeto o recibe un mensaje m(), si encuentra el método m() correspondiente en su clase, lo ejecuta
- ✓ Decimos que en un lenguaje de programación orientado a objetos existe polimorfismo cuando el mensaje m() puede ser recibido por objetos de clases diferentes

TERCER CUESTIONARIO

- ✓ Para filtrar una colección en Java, es recomendable usar el protocolo de streams
- ✓ Es importante testear temprano, y tanto como sea el riesgo del artefacto a testear
- ✓ Para escribir tests con valores de borde, identifico los bordes en las particiones de equivalencia y uso valores en esos bordes para mis tests
- ✓ En Java, es recomendable que todos los objetos en una colección compartan un tipo

- ✓ Solo el objeto que es dueño de una colección puede modificarla porque si no lo hacemos así, estamos rompiendo el encapsulamiento

CUARTO CUESTIONARIO

- ✓ En el modelo conceptual o del dominio, para lograr un mejor diseño OO es aconsejable incluir todas las clases candidatas que identifiquemos a partir de los casos de uso y de la lista de categorías, luego pueden revisarse en el diagrama de clases
- ✓ En reúso de código por composición permite usar al objeto a través de su protocolo, sin necesidad de tener que conocer su implementación
- ✓ En reúso de código extendemos pensado en “herencia de comportamiento” para cumplir con “es-un”
- ✓ Los diagramas de secuencia del sistema (DSS) y los diagramas de secuencia de diseño (DSD) son ambos diagramas de secuencia UML que expresan interacción entre objetos, en distintas etapas del proceso de desarrollo OO
- ✓ En los contratos de operaciones, las postcondiciones describen el estado y cambios del sistema después de ejecutarse la operación, usando conceptos del modelo conceptual o del dominio

QUINTO CUESTIONARIO

- ✓ A diferencia de Java y Smalltalk, que son lenguajes orientados a objetos basados en clases, ECMAScript es basado en prototipos
- ✓ Smalltalk es un lenguaje dinámico, en el que no se indica explícitamente el tipo de las variables
- ✓ En ECMAScript, cada objeto hereda comportamiento y estado de su prototipo
- ✓ En Smalltalk todo se implementa con objetos y está abierto a modificación. Incluso lo que comúnmente conocemos como estructuras de control, se implementa como envíos de mensajes a objetos
- ✓ ECMAScript es un lenguaje dinámico, en el que no se indica el tipo de las variables