Introducción a Objetos 1

Un software orientado a objetos se construye como un conjunto de objetos que colaboran enviandose mensajes.
Un objeto es una abstracción de una entidad de nuestro dominio. La abstracción nos permite quedarnos solo con los datos que consideramos relevantes para nuestro sistema, para lo que queremos representar.
No vamos a tener un objeto Main, porque no le vamos a dar prioridad a un objeto por sobre otro. Vemos los sistemas como muchos objetos que están al mismo nivel, sin tener uno más importante que otro.

Cada objeto va a tener **identidad**, **conocimiento** y **comportamiento**.

Identidad: Para distinguirse de otro objeto
Conocimiento: Está expresado en base a sus atributos y a su relaciones con otros objetos, esto describe lo que el objeto sabe ya que se conoce a sí mismo y a otros objetos con los que interactúa
Comportamiento: Que cosas sabe hacer el objeto, a que mensajes sabe responder, que le podemos pedir. Esto es esencial. La realización de cada mensaje se hace a través de un método, que se activan con un mensaje.

Para que un objeto pueda mandarse un mensaje a sí mismo es obligatorio el uso del **this**, en la clase tenemos que poner **this.metodo()**
Para poder extender el comportamiento heredado (para re-implementar un método incluyendo el comportamiento que se hereda) usamos la pseudo-variable **super**, que hace referencia a la superClase inmediata superior de la clase que está ejecutando el método.

Herencia!!!

Cuando hacemos herencia debemos saber que heredamos todo el comportamiento de la clase padre y cualquier cambio que esta haga. Usualmente tenemos que redefinir métodos heredados, eso esta bien solo hasta cierto punto, no podemos hacer herencia solo porque nos sirven dos métodos, para hacer herencia se tiene que cumplir él “ES UN”

● Preferencias y cosas a tener en cuenta:

•Copy-paste es malísimo, no hay que repetir código!!!
•Nunca hacemos objeto.variableDeInstancia, NUNCA, nos rompe el encapsulamiento. Si necesitamos algo del objeto hay que comunicarnos por él mediante métodos, no accediendo a su implementación interna. Acordarse que si un objeto quiere cambiar toda su implementación no debe modificar a los objetos que le envían mensajes.
•Siempre que se quiera conocimiento que un objeto tiene, se le pide a través de mensajes, si el objeto no tiene mensajes para eso es porque el objeto no quiere compartirlo por ende no podemos acceder a él, mucho menos por las variables de instancia.
•De los objetos siempre se usa la interfaz pública.

•Evitar métodos largos, siempre hacer que cada método haga una única cosa y si es necesario hacer un método que requiera muchas cosas, hacer cada cosa en un método separado y mandarse mensaje a sí mismo para que ejecute esos métodos. Es **mala señal** si un método tiene más de **10 renglones**.
•Para hacer una jerarquía de clases (con superclase) se tiene que cumplir la relación **es-un** donde las subclases tienen que ser un/a superclase.
•Si el enunciado nos dice por ejemplo que un cliente podría pasar de normal a vip, significa que no pueda hacer jerarquía ya que el objeto no puede cambiar el tipo, por lo que habría que buscar otra forma de hacerlo.

Diagrama UML!!!

NombreDeClase
Variables de instancia **visibilidad** nombreVariable: **tipoDeDato**
Métodos de la clase **visibilidad** nombreMetodo (parametros): **tipoDeRetorno**

•Los **NombreDeClase** van siempre en singular y empiezan con mayúscula.
Para las variables:
La **visibilidad** de un atributo puede ser:

+ Para un atributo público (PROHIBIDO)
- Para un atributo privado (El que más vamos a usar)
Para un atributo protegido, es decir, es privado excepto por las subclases de la clase donde está declarado.

Para los métodos abstractos hay que anteponer <<abstract>> a la visibilidad.

Los **tipoDeDato y tipoDeRetorno** en UML son **Integer, Boolean, String, Real, Date**

También tenemos atributos de clase van Subrayados y significa que el valor de este atributo va a ser igual para cada una de las instancias de esta clase. (Por ejemplo, en la clase Ventas ventasRealizadas es un atributo que va a valer lo mismo en cada instancia)

Para los métodos:

La visibilidad es igual a la de las variables, solo que si se pueden usar métodos públicos porque van a ser los que nos permitan que otros objetos se puedan comunicar con el objeto que se está modelando.
Si el método no recibe parámetro los paréntesis son obligatorios de todas formas pero si recibe parámetros hay que especificar el tipo de cada uno y separarlos con coma. Los parámetros se definen como nombre: tipo. Su tipo puede ser otro objeto.
Si el método retorna algo al final hay que poner :TipoDeRetorno.

Para modelar constructores le agregamos <<Create>>
<<Create>>nombre(parámetros:tipo,otroParámetro:tipo)

Relaciones en UML!

Asociación (relación de que dos objetos se conocen)

Se representa solo con una línea que une dos clases a que esas clases se conocen. Si la relación tiene una flecha significa que el objeto que del que sale la fecha conoce al objeto que está apuntado. Hay que ponerle un nombre a la relación (En Java esta relación se representa mediante un atributo que tiene como nombre el nombre de la relación y como tipo el objeto apuntado).

Se la flecha tiene un * al final hace referencia a una colección.

Si tenemos una relación que tiene un rombo sin pintar en un extremo significa que los elementos de la relación no se necesitan mutuamente para existir.

Si tenemos el rombo pintado, significa que sus partes desaparecen si el todo desaparece.

Generalización (relación de herencia)

Se diseña con una flecha que apunta a un único lugar desde más de una clase, si la clase apuntada es abstracta se usa <<abstract>> arriba del nombre de la clase o el nombre de la clase se escribe en cursiva, pero es más difícil de notar cuando se diseña en papel.

Interfaz (relación entre la interfaz y la clase que lo implementa)

Se diseña con una línea punteada y una flecha (de la forma triángulo sin relleno).

Test

En objetos 1 vamos a ver: **Test de unidad** (testear métodos que son la unidad mínima del programa, son bien específicos nos dicen exactamente qué método falló), **test funcionales y test automatizados** (Se refiere a escribir programas que testeen nuestro programa, es decir un programa que se asegure que las clases y métodos funcionen bien. Por cada clase que escribimos vamos a tener una clase que la testee).

Anatomía de los test: Por cada clase que escribo escribo una clase de test y por cada método que podría fallar escribo un método de test, es decir se testean en función del riesgo de los métodos. Los test se ejecutan sin orden. Los métodos privados, constructores, getters y setters no se testean.

Los valores para testear tenemos que pensarlos como rangos, el rango “sin problemas” lo testeamos con un único valor, ya que al asegurarnos que uno ande todos los demás andarán, luego vamos a probar los dos extremos del rango, tanto los valores mínimos como máximos.Estos valores los denominamos valores de borde, siempre que nuestro programa falle lo hará en los bordes

Por ejemplo, si tenemos que hacer que nuestro programa solo haga lo que se le pide si hay energía, entonces solo va a funcionar si la energía es > 0, por lo que podemos probar con el 5 y el 0, ya que si funciona para 5 asumimos que funciona para cualquier entero positivo mayor que 0, y luego probaremos el 0 que es el límite de los valores de corte. Ya analizados todos los programas que hicimos hasta la fecha sabemos que en la parte que se rompe el código es en los límites, en vez de probar el 5 podemos probar el 1 ya que es el primer valor con el que la energía va a ser suficiente, entonces probamos el 0 y el 1, porque consideramos que si anda para esto valores va a andar para todos los demás.

Sintaxis: Se escriben los test como el nombre de la clase pero con la palabra test como sufijo. NombreClaseTest
Definimos una variable de instancia a la que seteamos con todos los datos que vamos a usar para nuestros test, para el método que va a setear nuestra variable de instancia para los test usamos @BeforeEach, luego para definir los test utilizamos @Test y a cada método le ponemos de prefijo la palabra test, testMétodo.

Coolecciones c:

•List (java.util.List): Admite duplicados y sus elementos son indexados por enteros de 0 en adelante (Depende de su posición). De List vamos a usar ArrayList, pero también tiene otras implementaciones como Vector y LinkedList.
•Set (java.util.Set): No admite duplicados y sus elementos no están indexados.
•Map (java.util.Map): Asocia objetos que actúan como claves a otros que actúan como valores (Hace hashing)
•Queue (java.util.Queue): Maneja el orden en que se recuperan los objetos (LIFO, FIFO, por prioridad, etc.)

REGLA: Nunca modificó la colección que me envió otro objeto, ya que solo el dueño de la colección puede modificarla. Si modificamos una colección desde otro lugar el objeto que es dueño de la colección no se entera, por ende si hay algún cálculo auxiliar, conteo, o lo que sea que hago el objeto dueño de la colección cada vez que agrega un elemento no lo va a hacer y por ende podría generar inconsistencia en los datos.

Stream y expresiones lambda.

Los streams son métodos anónimos que no pertenecen a ninguna clase y nos ayudan a hacer las operaciones que siempre (o casi) repetimos con las colecciones, estás son:
•Ordenar respecto a algún criterio •Recorrer y hacer algo con todos sus elementos
•Encontrar un elemento (max, min, DNI = xxx, etc.) •Filtrar para quedarme solo con algunos elementos •Recolectar algo de todos los elementos •Reducir (promedio, suma, etc.)

Para trabajar con stream necesitamos usar expresiones lambda, un ejemplo de que como recorrer una colección con una expresión lambda puede ser: (suponiendo que alumno es una colección que entiende el mensaje cursar()) es: alumnos.forEach(a -> a.cursar()); Esto se lee como recorre la colección alumnos y por cada uno de sus elementos denominados a, hace a.cursar(). La expresión lambda es lo que está entre paréntesis, es una función que en este caso tiene un solo parámetro a (si tuviera más van entre paréntesis separadas por comas) y ejecuta solo una linea (si tuviera más líneas van entre llaves {} separadas por punto y coma ;)

```
(parámetros, separados, por, coma) -> { cuerpo lambda }
```

Los stream nos proveen acceso a una fuente de información (en esta caso colecciones) sin modificarla permitiéndonos tratar la colección como un flujo de datos, donde cada dato se va a visitar una única vez. Para empezar a usar stream hay que usar el método stream() seguido de operaciones intermedias con su respectiva expresión lambda y al final siempre hay que poner una expresión terminal.

Operaciones intermedias
filter
map
limit
sorted

Operaciones terminales
count | sum
average
findAny | findFirst
collect
anyMatch | allMatch | noneMatch
min | max

La operación terminar puede ser collect que nos da muchísimas opciones, con el Collector.toList() genera una nueva lista con lo que quedó de las operaciones intermedias, con Collector.counting() contamos la cantidad de elementos que quedaron después de las operaciones intermedias. Otra operación final puede ser .sum(); que retorna la suma de todos los valores que quedaron después de las operaciones intermedias
.stream().filter(s -> s.esMayorDeEdad()).collect()

Modelado de clases y atributos

Para identificar clases y atributos vamos a buscar todos los sustantivos (o frases nominales, es decir frases que tienen como núcleo un sustantivo) y empezar a analizarlas, ya que todos los sustantivos que encontremos los vamos a empezar a pensar como clases candidatas. Para continuar los listamos, graficamos el modelo y comenzamos a analizar cuáles van a ser verdaderamente clases y que atributos van a tener, una vez hecho el gráfico vamos a analizar la relación entre las clases. Es obligatorio crear clases cada vez que necesitemos la especificación o descripción que necesitan ser preservadas en el tiempo de un servicio o artículo o si al eliminar la instancia que lo describe se pierde información (Por ejemplo cuando tenemos copias de libros hay que crear la clase copia).
Un tip para hacer esto último es imaginar de qué tipo son esas cosas que escribimos, si tenemos algo de tipo primitivo como un nombre (String), edad (int) estos serán atributos. Para saber si es una clase tenemos que notar que no lo podemos definir así ya que son compuestos, no hay ningún tipo ya hecho en Java que nos permita identificarlos, por ende serán una clase. Luego para analizar las relaciones lo que podemos hacer es ver si el atributo pertenece a la clase recién descubierta o de qué manera se comunican la clase del atributo con la clase recién descubierta, seguramente sea mediante el envío de mensajes por lo que vamos a tener que definir métodos.
La relación que hay entre un objeto y otro se llama rol (el nombre que ponemos en la flecha al UML).

Para empezar a agregar el comportamiento (métodos) vamos a usar HAR: Heurísticas para asignar responsabilidades. Es responsabilidad del objeto hacer cosas como cambiar su estado, iniciar una acción en otro objeto o controlar y coordinar actividades de otros objetos.

Tenemos heurísticas: Experto en la información, creador, controlador, bajo acoplamiento, alta cohesión, polimorfismo.
Experto en información: Hay que asignarle la responsabilidad al objeto que tiene todos los datos para resolverlo. Es decir, los objetos hacen cosas con la información que tienen. También puede pasar que para cumplir su responsabilidad el objeto necesite información que se encuentra dispersa en diferentes objetos, en este caso se dice que son expertos en información parcial.
Creador: Es quién tiene la responsabilidad de crear la clase. Esto se da cuando un objeto tiene/recibe todos los datos necesarios para inicializar a otro, cuando contiene objetos de dicho tipo y los usa de forma exclusiva.
Controlador: asigna la responsabilidad de manejar eventos a una clase que controla. El objetivo del controlador es encontrar manejadores manejadores de los eventos del sistema, sin recargar de responsabilidad a un único objeto y manteniendo alta cohesión.
Bajo acoplamiento: Asignar las responsabilidades de la manera que queden con la menor dependencia posible una clase de otra. Una clase tiene bajo complamiento cuando mantiene pocas relaciones con otros objetos. Esto se evita porque dificulta el entendimiento y complica la propagación de cambios en el diseño.
Alta Cohesión: Buscamos que el objeto no se cargue con muchas responsabilidades distintas, sino que haga más bien cosas puntuales y específicas. (El que mucho abarca poco aprieta) (Divide y vencerás, acá está bien delegar y hacer que otras clases hagan algo). Es mejor tener muchas clases que hacen pocas cosas que tener pocas clases que hagan muchas cosas.
Polimorfismo: Nos permite mediante interfaces hacer que objetos de diversas clases entiendan los mismos mensajes por ende darle más versatilidad al código.
No hables con extraños: Solo mandarles mensajes a los objetos que son this/super, a un parámetro que llega al método, un objeto que se tiene como variable de instancia, un objeto de su colección y un objeto creado dentro del método. Todos los demás objetos son extraños.

Heurísticas para diseño “ágil” Orientado a Objetos: S O L I D.
S → Single responsibility principle: Principio de responsabilidad única. Una clase debería ser responsable de una única tarea, causando una alta cohesión.

O → Open-Close principle: Principio que dice que la clase debe ser abierta para su extensión (ser capaz de agregar nuevas funcionalidades) pero cerrada para modificación (que las nuevas modificaciones no cambien el diseño existente).

L → Liskov susbtitution principle: Principio de sustitución. Esto quiere decir que si un programa usa una clase A y esta es extendida por B, C y D, el programa tiene que poder utilizar cualquiera de las subclases y seguir siendo válido, esto nos asegura el uso correcto de la herencia y del polimorfismo.

I → Interface segregation principle: Las clases que implementan interfaces muy voluminosas suelen no ser cohesivas, ya que las clases deben implementar todo el protocolo. Las clases no deberían verse forzadas a implementar métodos que no van a usar.

D → Dependency-Inversion principle: Los módulos de alto nivel no pueden depender de los de bajo nivel.

Malos olores en la programación

Envidia de atributos: soy un objeto que pide cosas a otros objetos para hacer algo yo mismo (por ejemplo un cálculo)
Para evitarlo: la tarea la debe hacer el objeto que tiene las cosas que se necesitan.

Clase Dios: Una clase que resuelve todo y las demás están anémicas. No cumple el principio de "una sola responsabilidad".
Para evitarlo: Ver qué otros objetos podría encargarse de alguna de las responsabilidades. Ver cuál de los objetos que este objeto conoce podría ser responsable de algo que ahora hace él.

Código duplicado: si hago Ctrl+C Ctrl+V (copiar y pegar) estoy metiendo la pata.
Para evitarlo: ¿No puedo generalizar ese comportamiento en una clase y heredarlo? ¿No puedo llevarlo a otro objeto y re-utilizarlo por composición? ¿No puedo extraerlo en un método en la misma clase y re-usarlo?

Clase larga: tengo una clase muy grande en comparación al resto.
Para evitarlo: ¿No será que esa clase puede delegar algo en otros objetos a los que conoce? ¿No será que esa clase modela más de una cosa? (Puedo pensarla como una composición de varios objetos).

Método largo: si un método tiene más de 10 renglones, es mala señal.
Para evitarlo: Identificar dentro del método largo, partes que podría considerar comportamientos individuales. Llevar cada parte a un nuevo método (con un buen nombre) y cuando necesite llevar a cabo uno de esos comportamientos, enviar mensajes a this.

Eso debería ser un objeto (obsesión por los primitivos): A veces modelamos como strings o números cosas que deberían ser objetos. Cuando hacemos eso, el comportamiento que debería tener ese objeto termina estando en un lugar que no corresponde.
Para evitarlo: Pensar si eso que estoy modelando con un string o número (un primitivo) no debería ser modelado con una clase específica.

Switch statements: Debería sentir mal olor cuando veo que se usa un ***if*** *(o un* ***case*** *o un* ***switch*** *o* ***ifs anidados****)* para determinar de qué forma se resuelve algo. Esto es más evidente si la variable que uso en el *if* tiene un nombre que suena a "tipo". Algo como "if (tipo = esto) entonces lo hago así. El caso más extremo es preguntar por la clase del objeto (si es de esta clase lo hago así, si no lo hago asá).
Para evitarlo: ¡Aplico adecuadamente el polimorfismo!

Variables de instancia que en realidad deberían ser temporales: Si una variable de instancia deja de tener sentido en algún momento de la vida del objeto, entonces es probable que sea temporal o que sea responsabilidad de otro.
Para evitarlo: pensar si esa variable es realmente un atributo del objeto, que lo acompaña siempre, o es algo que necesito temporalmente dentro de un método.

Romper encapsulamiento: No solo es preocupante acceder a variables de instancia de otros objetos, además podemos romper el encapsulamiento de manera más sutil. Por ejemplo, si automáticamente agregamos setters y getters para todas las variables de instancia de nuestros

objetos. Si modificamos una colección que no es nuestra (es de otro objeto) también atentamos contra el encapsulamiento.
Para evitarlo: sólo agregar getters y setters cuando es necesario; nunca modificar una colección que no es nuestra, delegar las tareas a los que tienen la información que se necesita.

Clase de datos o clase anémica: una clase que parece un registro de datos debería dar mala espina. A veces los enunciados son simplificaciones que hacen que algunas clases terminen siendo así, pero por lo general sospecho cuando una clase solo tiene datos y no tiene comportamiento.
Para evitarlo: asegurarse que no hay comportamiento en el sistema que debería estar haciendo esa clase y lo hace otro objeto (el cual seguramente muestre envidia de atributos).

No quiero mi herencia: cuando encontramos un método que redefine a uno heredado pero hace algo totalmente diferente, debemos desconfiar. Si no sirve el comportamiento heredado tal vez no se cumpla el principio "es-un". El caso extremo es redefinir un método heredado, para indicar un error o no hacer nada.
Para evitarlo: pensar si no puedo reorganizar la jerarquía de clases para que ninguna clase herede comportamiento que no quiere.

Estilo de programación

Nombre de mensaje que revela la intención: Que el nombre del mensaje comunique lo que se quiere hacer, no cómo.

Delegación a this: Permite descomponer un método en partes que el mismo objeto resuelve. Cada método hace una cosa. Permite que la subclase redefina/extienda solo un paso.

Métodos cortos: Siempre prefiere tener métodos cortos. Para que sean más fáciles de leer, utiliza nombres de mensajes que revelen la intención (servirán como documentación de lo que hace el código)

Cada cosa se hace una sola vez: Para ello es importante aprender el protocolo de colecciones y otros objetos frecuentemente utilizados. Es recomendable explorar el protocolo de los objetos que voy a utilizar antes de comenzar a programar.

Los nombres de las variables deben indicar su rol. Elige los nombres de las variables para que quede claro qué rol cumplen en el método/clase. Los nombres de variables siempre comienzan con minúscula. No temas a los nombres largos de las variables, con varias palabras y sintaxis de camello.

Piensa bien los nombres de las clases. Estos siempre inician con mayúscula y singular. No temas a los nombres de clase largos, con varias palabras y sintaxis de camello capitalizada. Si se puede, que el nombre de la subclase ayude a reconocer que es un caso particular de la superclase (por ejemplo, agregando alguna palabra al nombre de la superclase para definir un caso especial: EmpleadoDePlanta subclase de Empleado)

Operaciones y cómo manejar coolecciones!

Vamos a usar siempre ArrayList así que para crear una lista de alumnos hacemos:

```
List<Alumno> alumnos = new ArrayList<Alumno>();
```

`add(Alumno alu);` Agrega el Alumno alu en la última posición de la lista.

`add(Int 3, Alumno alum);` se agrega el Alumno alum en la lista en la posición ingresada. Recordar que las listas se empiezan a indexar en 0.

`get(Int 5);` Devuelve el elemento que se encuentra en la posición 5 (el 6to cargado)

`size();` Devuelve la cantidad de elemento que contiene la lista. El número será uno más que el índice del último elemento.

`contains(Alumno a);` Devuelve True si la lista contiene al Alumno a y False en caso contrario.

`remove(Int 8);` Elimina el elemento contenido en la posición indicada por Index y corre los elementos para no dejar el lugar vacío.

`clear();` Elimina todos los elementos de la lista dejándola vacía.

`isEmpty();` Devuelve True si la lista está vacía y False en caso contrario.