Programación Orientada a Objetos

75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

Elaboración: Lihuén Carranza y Luca Salluzzi
(en base a las presentaciones de Prof. Dr. Diego Corsi)

Índice

1. Introducción

Hay dos grandes grupos de paradigmas de programación, por un lado tenemos los **Imperativos** y por otro los **Declarativos**. Los **Imperativos** son más tradicionales, más cercanos semánticamente al funcionamiento del hardware, enfatizando en la ejecución de instrucciones. Dentro de este grupo se encuentran los **Procedimentales** (como C) y los **Orientados a Objetos** (como Smalltalk). Los paradigmas **Declarativos** hacen énfasis en la declaración de expresiones, abarcando los lenguajes **Funcionales** (como Haskell) y los **Lógicos** (como Prolog).

Debido a lo útil, simple y poderosa que es la programación funcional, ha logrado hacerse lugar en la mayoría de los lenguajes de la actualidad, esto incluye a Java, el cual es actualmente **multiparadigma** debido a que contiene aspectos de Programación Orientada a Objetos (OOP) y Funcional.

En esta cátedra vamos a centrarnos mayoritariamente en la parte de Java orientada a Objetos.

2. Sistemas Orientados a Objetos

> "En vez de un procesador de bits consumiendo estructuras de datos, tenemos un universo de objetos bien comportados, cada uno de ellos pidiéndole a otro que cortésmente le realice sus variados deseos". [1]

En la programación tradicional, los datos son pasivos y se les aplican procesos. En cambio, cuando se trabaja con objetos, son estos los que **actúan** para resolver un problema, respondiendo a estímulos (**mensajes o eventos**) del medio externo. En sistemas diseñados como un conjunto de servicios, por lo general se tendrá un disparador y luego **un objeto irá llamando a otro** hasta que todo el sistema se pone en movimiento.

2.1. Definicion de Objetos

Objeto: "una entidad con comportamiento" (Fontela, 2018, p. 27)
Está definido por:

- *Identidad* (lo que lo distingue de otros objetos de las mismas características).
- *Estado* (los valores de sus **atributos**).
- *Comportamiento* (sus reacciones ante mensajes recibidos y sus acciones en forma de mensajes enviados, implementados mediante **métodos**).

Un **modelo** es una representación de un sistema del mundo real que resulta de llevar a cabo el proceso de *abstracción*.

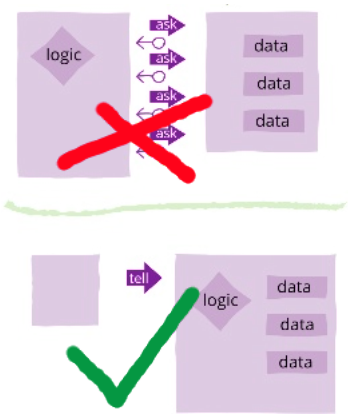
La **abstracción** es una simplificación que incluye solo aquellos detalles relevantes para determinado propósito y descarta los demás.

3. Principios de diseño

3.1. Tell, Don't Ask

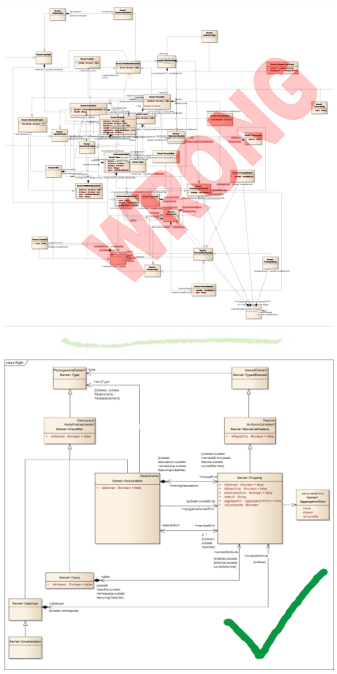
Principio de diseño orientado a objetos en el que se basa la **buena práctica** de evitar solicitarle a un objeto que indique su estado y luego llevar a cabo una acción basándose en este, en lugar de solicitarle al objeto que lleve a cabo la acción él mismo.

[1] Fuente: Ingalls, Daniel H. H. "Design Principles Behind Smalltalk". En: BYTE Magazine (Agosto, 1981). McGraw-Hill, Nueva York, p. 290



3.2. Less is More (Menos es Más)

Principio de diseño en el que se basa la **buena práctica** de dividir grandes diagramas en varios diagramas de menor tamaño y mejor enfocados en lo que se quiere comunicar.



Una frase para reflexionar:

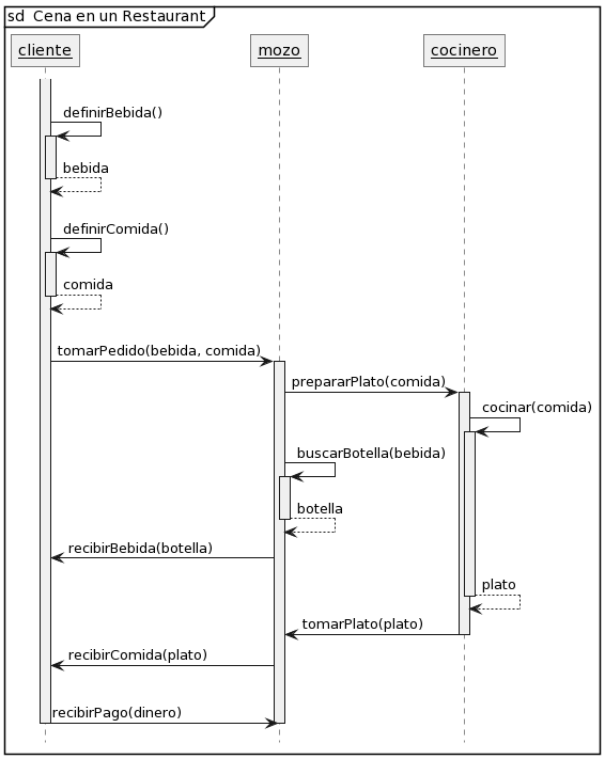
> Sometimes **the tool is talking** a subtle language. It slows down our performance to remind us we need to shape it better. I've lost count of how many times I've been presented **huge UML or entity-relationship diagrams** that require continuous scrolling, impairing my performance in using them. **The diagram was begging the designer to find a reasonable partitioning**, one that could fit on a single, if large, screen. Printing the diagram on even larger paper and working from there is a common way of shutting up our material. It works in the short term, and I use it on early diagrams of large systems, when the architecture isn't necessarily stable. But **I listen, and break large diagrams into subsystems** as soon as I can.

Fuente: Pescio, C. "Listen to your tools and materials". En: IEEE Software, sept./oct. 2006

4. Diagramas de Secuencia (UML)

Los diagramas de secuencia son **modelos dinámicos** que describen cómo un **grupo de objetos** colabora en determinado escenario a través del tiempo. En el diagrama de secuencia se representa **cada objeto como un rectángulo** arriba de su **línea de vida.**

Los **mensajes son flechas** entre objetos y el orden de los mensajes está dado por el eje vertical que se lee de arriba hacia abajo. De los mensajes se escribe por lo menos el nombre aunque también se pueden agregar los parámetros. También representan cierto control: iteraciones, condiciones, etc.

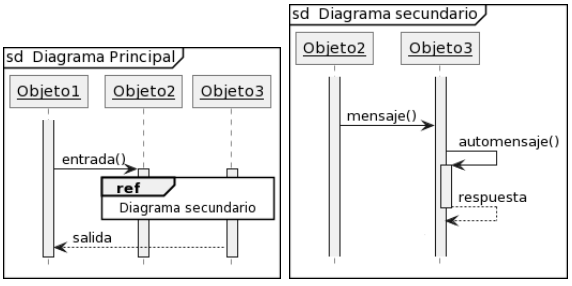


https://plantuml.com/es/
https://www.planttext.com/

4.1. UML: El uso de Interacción (Interaction Use)

Permite usar (o llamar) otra interacción, simplificando los **diagramas de secuencia** grandes y complejos. También es común reutilizar alguna interacción entre varias otras interacciones.

Una restricción impuesta por la especificación UML que a veces es difícil de seguir es que el **fragmento combinado ref** debe cubrir todas las *líneas de vida* involucradas representadas en la interacción envolvente. Esto significa que todas esas *líneas de vida* deben ubicarse cerca unas de otras. Si tenemos otro **fragmento combinado ref** en el mismo diagrama, podría ser muy complicado reorganizar todas las *líneas de vida* involucradas como lo requiere UML.

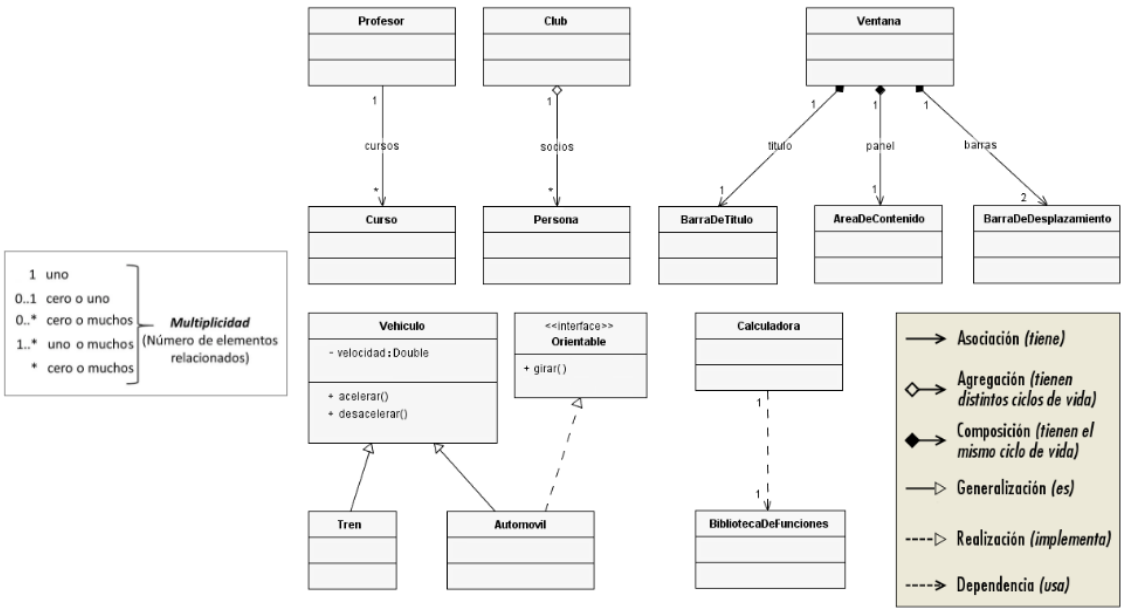


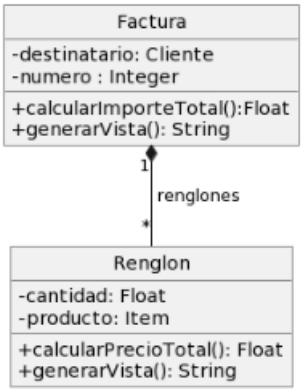
4.2. Conclusión

- Los Diagramas de Secuencia Ayudan a ver el flujo de control y el **ordenamiento temporal de los eventos**
- Los Diagramas de Secuencia ayudan a **encontrar los métodos** necesarios de los objetos
- Conviene colocar más a la izquierda los objetos que inician la **interacción** y los más importantes
- Como todo diagrama, su gran virtud es la **simplicidad** y el impacto visual: no pretenden mostrar toda la complejidad de un sistema.

5. Diagramas de clases (UML)

- Representan **relaciones estáticas** entre clases e interfaces (no cambian a través del tiempo)
- Una clase se representa con un **rectángulo con tres divisiones**: una para el *nombre*, otra para los *atributos* y otra para los *métodos*.
- En ocasiones no usamos las tres divisiones, sino solo dos (eliminando los atributos) o una (solo con el nombre de la clase)
- Los atributos y métodos **privados** se pueden indicar precedidos de un signo -, los **públicos** del signo + y los **protegidos** del signo #
- No es conveniente utilizar aspectos de la sintaxis que tengan un significado solo en determinado lenguaje



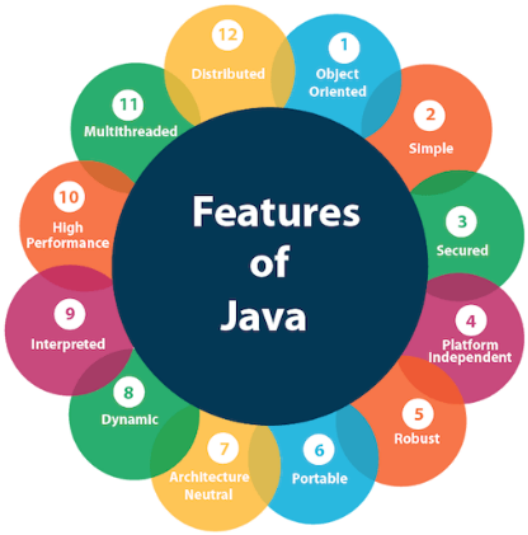


5.1. Conclusión

- No es necesario representar todas las clases con todos sus detalles.
- Conviene representar atributos, asociaciones y generalización (la visibilidad, las dependencias y otras propiedades son necesarias solo en algunas ocasiones especiales).
- Conviene dibujar modelos solo de las partes importantes del sistema.
- Son herramientas, por lo que el grado de detalle hay que manejarlo desde el punto de vista de la practicidad.
- Conviene ordenar los elementos del diagrama para minimizar el cruce de líneas y para que las cosas que son cercanas semánticamente queden cerca físicamente.

6. JAVA

1. Orientado a objetos
2. Simple
3. Seguro
4. Independiente de la plataforma
5. Robusto
6. Portable
7. Arquitectónicamente neutro
8. Dinámico
9. Interpretado
10. Alto desempeño
11. Multihilo
12. Distribuido



6.1. Estado de los objetos

Los objetos que interactúan en un sistema orientado a objetos tienen un **estado** que está dado por los valores de sus **atributos** (*variables de instancia*) en determinado momento. Generalmente, el estado de un objeto evoluciona en el tiempo, ya que desde sus métodos es posible acceder a sus atributos y modificarles el valor. La excepción son los atributos calificados como final, ya que estos permanecen *constantes*. Por convención, en Java los nombres de los atributos empiezan en minúsculas. La excepción son los atributos calificados como final, que se escriben totalmente en mayúsculas.
Los atributos de un objeto no deberían ser manipulables directamente por el resto de los objetos del sistema, por eso casi siempre se los califica como `private` y se accede a ellos mediante métodos. Se recomienda *declarar* los atributos (es decir, indicar su tipo y su nombre) antes de los constructores, y utilizar éstos para *inicializarlos* (es decir, asignarles su primer valor).

6.1.1. Atributos de tipos primitivos

En Java, cualquier variable declarada usando un *tipo primitivo* contendrá directamente un dato.

Tipo	Rango	Valor por defecto
byte	-128 .. 127	0
short	-32768 .. 32767	0
int	-2147483648 .. 2147483647	0
long	-9223372036854775808 .. 9223372036854775807	0L
float	-3.4E38 .. -1.18E-38 .. 0 .. 1.18E-38 .. 3.4E38	0.0f
double	-1.8E308 .. -2.23E-308 .. 0 .. 2.23E-308 .. 1.8E308	0.0d
char	'\u0000' .. '\uffff'	'\u0000'
boolean	false .. true	false

En cambio, si el tipo usado para declarar una variable no es primitivo, esta contendrá una *referencia a un objeto*, es decir, la dirección de memoria donde el objeto está almacenado. El *valor por defecto* es el valor que toma una *variable de instancia* (es decir, un atributo) cuando se la usa **sin inicialización** previa. En el caso de las *variables locales* de los constructores y los métodos, su uso sin inicialización previa no es posible (no compila).

6.1.2. Atributos de tipo String

Un atributo que represente el texto característico de un objeto (por ejemplo, el apellido de una persona), puede implementarse en Java como una instancia de la clase `String`. Esta clase tiene 13 constructores diferentes que permiten construir cadenas de caracteres. Por ejemplo, si escribiéramos lo siguiente: `new String (new char[]'h', 'o', 'l', 'a')` estaríamos creando una instancia de String a partir de un arreglo *anónimo* de 4 caracteres. Por una cuestión de practicidad, en Java es posible crear un objeto de la clase String simplemente colocando caracteres entre comillas: `"Hola, mundo!"`. El valor por defecto de los atributos que son instancias de la clase `String` (o de cualquier otra clase) es null. Cuando una variable vale `null`, esta no se refiere a ningún objeto, por ello no es posible enviarle mensajes. La cadena vacía sí es un objeto. Por lo tanto, una variable String inicializada con la cadena vacía puede responder a los mensajes que le enviemos (`length, equals, charAt, substring, concat, contains, indexOf, isEmpty, trim, startsWith, endsWith, toLowerCase, toUpperCase, replace, replaceAll`, entre otros).

6.1.3. Atributos de tipos complejos: arreglos y colecciones

Un atributo que represente un grupo de elementos del mismo tipo primitivo (por ejemplo, los números premiados en un sorteo) o un grupo de instancias de la misma clase (por ejemplo, los alumnos inscriptos en un curso) puede implementarse en Java como un *arreglo* o una *colección*. En UML, para indicar cualquiera de las dos implementaciones se utiliza la **multiplicidad**. Un *arreglo* es un objeto contenedor cuyo tamaño se define al momento de su instanciación y que no acepta demasiados mensajes (`copyOfRange, equals, fill, binarySearch` y `sort`, entre otros). El acceso a sus posiciones se realiza mediante un índice entre [ ], y se les cargan valores con =. Las colecciones no tienen un tamaño fijo (se adaptan a medida que se les va cargando su contenido). Entre las más usadas se encuentran `ArrayList, LinkedList y TreeSet. Aceptan muchos mensajes distintos, como isEmpty, size, add, set, get, remove, indexOf, contains`, etc. *Solo pueden contener datos primitivos si estos se envuelven en objetos.*
Arreglos: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html
Colecciones: https://docs.oracle.com/javase/tutorial/collections/TOC.html

6.2. Comportamiento de los objetos

Todo objeto tiene un **comportamiento** que está dado por las funciones (**métodos**) que ejecuta cuando recibe solicitudes (**mensajes**), generalmente de otros objetos. También durante la instanciación de los objetos se ejecutan instrucciones, por ejemplo, para inicializar sus atributos. Este comportamiento está codificado en **constructores**, los cuales no tienen valor de retorno (ni siquiera `void`) y deben tener el mismo nombre que su clase. Cada método debe limitarse a realizar **una única tarea bien definida**, y su nombre debe expresar esa tarea con efectividad. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.

Desde el resto de los objetos se debería poder solicitarle a un objeto la ejecución de sus métodos, por eso a estos casi siempre se los califica como `public`. Una excepción son aquellos métodos que solo se invocarán desde dentro del propio objeto, en cuyo caso se los debe hacer invisibles para los demás objetos, calificándolos como `private`.

- Hay tres formas de *invocar un método*:

  1. Pasándole un mensaje a un objeto, es decir, usando una variable que se refiera al objeto, seguida de punto (.) y el nombre del método. Por ejemplo: `estudiante.listar();`
  2. Utilizando el nombre de la clase, seguido de punto (.) y el nombre de un método de la clase. Por ejemplo: `Math.sqrt(81.0);`
  3. Usando solo el nombre del método, si este es parte de la misma clase. Por ejemplo: `cerrar();`

- Existen tres formas de *finalizar un método* y regresar el control al código que lo invocó. Si el método no devuelve un resultado, el control regresa cuando:

  1. el flujo del programa llega a la llave de cierre del método,
  2. se ejecuta la sentencia `return`;
     Si el método devuelve un resultado, el control regresa cuando:
  3. se ejecuta la sentencia return *expresión*; la cual evalúa la expresión y después devuelve el resultado al código que hizo la invocación.

Los métodos pueden devolver como máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga varios valores. El tipo del valor devuelto se indica antes del nombre del método, al declararse este último. La palabra reservada `void` se utiliza para indicar que un método no devuelve ningún valor. Los métodos (y también los constructores) pueden tener *variables locales*. Solamente es posible acceder a las variables locales dentro del ámbito en que están declaradas, y al finalizar la ejecución del método o constructor al que pertenecen, el valor de las mismas no se mantiene. Los métodos (pero no los constructores) pueden ser *recursivos*. Un método (o un constructor) puede ser invocado con cero o más **argumentos**, si hay una declaración que tenga una firma compatible. La firma (*signature*) de un método o constructor está compuesta por su nombre y los tipos de sus **parámetros**, por lo tanto es posible que haya múltiples versiones de un método o constructor, siempre y cuando sus firmas difieran. Esto se denomina *sobrecarga*.
Una característica importante de las invocaciones de los métodos es la **promoción de argumentos**. Por ejemplo, se puede llamar al método estático `sqrt` de la clase `Math` con un argumento entero, a pesar de que el método espera recibir un `double`. Tratar de realizar estas conversiones puede ocasionar errores de compilación, si no se satisfacen las reglas de promoción que especifican qué conversiones son permitidas y pueden realizarse sin perder datos. En casos en los que la información podría perderse debido a la conversión, el compilador requerirá que utilicemos un *casteo* para forzar explícitamente la conversión. En Java, el pasaje de argumentos a los métodos es **por valor**. Lo que se le pasa al método invocado es una copia del valor del argumento. Si el argumento es una variable, las modificaciones a la copia no afectan el valor de la variable original. Por ejemplo: `borrar(x)` no cambia el valor de la variable x. Sin embargo, si el parámetro no es de un tipo primitivo, el método que lo recibe puede (a través de la copia) acceder a los miembros públicos del objeto e interactuar con él, incluso modificándole el estado.