Teoría de Objetos

Por:
Carlos Lombardi
Fernando Dodino

INTRODUCCIÓN **2**

QUÉ ES UN PARADIGMA **2**

Objetivo: soft que cumpla ciertas cualidades *2*

LOS TRES CONCEPTOS BÁSICOS **3**

Objeto *3*
Mensaje *4*
Ambiente *5*
Software *5*
Volviendo a las preguntas *6*

CLASE Y MÉTODO **6**

Método *6*
Clase *6*
Clase abstracta / método abstracto *7*
Objetos sin clases *8*

POLIMORFISMO Y TIPOS **8**

El chiste del polimorfismo *8*
Polimorfismo tipado y no tipado *9*
Introducción a Interfaces, tipos básicos, asignación, tipo de una variable *11*

INTERFACES **12**

Qué es una interface *12*
Interfaces y herencia múltiple *14*
Qué se gana con las interfaces *15*
Compilador y máquina virtual *16*

REFERENCIAS **17**

EL FINAL **19**

ANEXOS **20**

EL DIBUJO 20
PRINCIPIOS DE VON NEUMANN 21

Introducción

En esta clase vamos a volver a pasar por los conceptos e ideas básicos de la orientación a objetos, refrescando un poco la cursada de Paradigmas. ¿Por qué arriesgarnos a aburrirlos? Porque creemos que los conceptos en los que hacemos hincapié, y cómo los describimos, son distintos a lo que vieron en Paradigmas. De movida ya ganaron una cosa, ver lo mismo desde dos enfoques/puntos de vista distintos enriquece.

Además, la forma de contar la orientación a objetos que elegimos destaca/abre algunos temas, problemas, ideas, puntos de vista, de los que vamos a hablar seguido en la materia, y nos interesa que esos temas aparezcan en una introducción a la OO. En particular, van a aparecer los Design Patterns, y el lenguaje Java.

Prometo que no se van a aburrir todo el tiempo, sí van a aparecer cosas que ya saben, pero mezcladas con otras que no. De hecho, un poquito de la orientación de esta clase está relacionado con preguntas en las que vemos que patinan cuando se las hacemos en el final. Aprovechando que varias cosas ya las saben, por algunos temas vamos a pasar rápido.

Qué es un Paradigma

¿Qué es la OO? De otra forma ¿qué es un paradigma? Es una visión de la computación[1], una forma arbitraria de definir qué es el software, de (justamente) verlo y analizarlo, de trabajar con él, de construirlo (dado que el soft es algo que creamos nosotros).

La verdá la verdá, con las computadoras con las que trabajamos, la computación son unos y ceros que se mueven entre una memoria y un CPU. Esto es un súper-resumen de los principios de Von Neumann (ver anexo sobre principios de Von Neumann).

A veces, muchas veces, está bueno armarnos una visión al estudiar un fenómeno o materia, que obviamente es arbitraria y artificial (en el sentido que “en realidad pasa otra cosa”). Ver, concebir el soft de determinada forma, te permite tener una comprensión de lo que estás haciendo al desarrollar, ir armando un criterio de cosas que conviene hacer y otras que conviene no hacer, crear o identificar herramientas de desarrollo que reflejen esa visión.

Objetivo: soft que cumpla ciertas cualidades

Otra cosa: en cuanto ingenieros especialistas en desarrollo de software, ¿por qué nos atrae la orientación a objetos? No por sus cualidades estéticas, porque el código “queda lindo”, o porque “tiene onda”, o por las propiedades teóricas (teoría de la computación) del paradigma.
Nuestro objetivo como profesionales es construir software que cumpla con determinadas cualidades (como el objetivo de los ingenieros civiles es construir p.ej. edificios o puentes que cumplan con ciertas cualidades, etc.):

- escabilidad en carga y en funcionalidad
- robustez
- mantenibilidad
- etc.

Durante la materia vamos a ver cómo las ideas, conceptos y algunas tecnologías asociadas a la OO, o que resulta más fácil o natural plasmar en un ambiente de desarrollo OO, ayudan a que el software construido cumpla con estas cualidades. En algún momento vamos a hablar específicamente de las cualidades deseadas en el software, ahora nos interesa solamente ligarlas con la OO y con el objetivo al estudiar algo relacionado con la informática.

[1] Una “ideología” en el concepto de Marx de la palabra. ¡Consulte con su docente marxista amigo!

Los Tres Conceptos Básicos

Para nosotros, la orientación a objetos se entiende a partir de tres conceptos: objeto, mensaje y ambiente. Hablemos un rato de cada uno.

Objeto

¿Qué es un objeto, qué merece ser llamado "objeto"?

El concepto es "comportamiento", tiremos una definición inicial

- Objeto: ente[2] que tiene comportamiento -

Es un ente computacional, que está dentro del software, puede representar o no un ente de la realidad que está afuera de la computadora.
P.ej.,

- un objeto Usuario, o uno Persona, pueden representar una persona real, en el manejo de usuarios de una aplicación,
- un objeto Persona puede representar una persona virtual en un juego como el Age of Empires (o el juego que más les guste),
- un objeto String eeeehhhh el correlato con la realidad es más difícil, un String es más bien un concepto relacionado con lo que pasa adentro de la computadora.
- un objeto Thread representa algo que está bien adentro de la computadora, acá no hay duda.

Tomemos como ejemplo una Persona, ¿cuál es el comportamiento de una persona? Comer / vestirse / transladarse / leer / … ¡y muchas otras cosas! (¡chán!).
Ahora, si yo implemento una Persona que "hace" todo esto pero a la que no le puedo preguntar ni indicar nada

- andá a Retiro
- ¿dónde estás?
- ¿qué comiste hoy?
- ¿estás caminando? ¿a qué velocidad?
- ¿te gusta el matambre?

lo que obtengo es un ente autista, que no me sirve para nada, y que no voy a considerar como objeto.

Entonces revisamos la definición

- Objeto: ente computacional que <u>exhibe</u> comportamiento -

donde el "comportamiento" son las cosas que sabe el objeto, que lo "exhibe" quiere decir que se las puedo pedir, que desde afuera del objeto se puede acceder a ese comportamiento.
A una Persona del Age of Empires le puedo pedir

- su edad
- a qué pueblo pertenece
- qué potencial defensivo tiene
- que junte madera
- que camine hasta un determinado lugar.
- que se detenga.
- que se suicide.
- su posición.

¿Quién le pide cosas a un objeto?
Otro objeto, en principio no hay más que objetos[3].

Hablemos un poquito del String, ¿qué le puedo pedir? Una fácil, que se concatene con otro String. En Smalltalk el mensaje es la coma

[2] "ente": cualquier cosa que tenga alguna forma de existencia
[3] No me diga que no se le vino a la mente a su docente de Paradigmas diciendo "En Smalltalk **todo** es un objeto"…

string1 , string2
¿qué hace esto, modifica string1? **¡no!** devuelve otro String que es el resultado de la concatenación. Una vez creado, un String nunca cambia. De objetos que nunca cambian vamos a hablar (a lo largo de la cursada), es un concepto importante.

Antes de pasar a otro tema, un número ¿es un objeto? A veeeeeeer

En Smalltalk ¿qué pasa? Bien, los números son tratados como objetos, y las operaciones (p.ej. la suma) son mensajes. P.ej.
2 + 4
es el envío del mensaje + al objeto 2 con parámetro 4. A un entero puedo preguntarle si es par
(2 + 4) even
la relación con la máquina (subyacente) pasa porque el + de los enteros está implementado con una primitiva de la máquina virtual de Smalltalk.

¿y en Java? Los números en Java no son objetos, son otra cosa. En Java hay objetos y tipos básicos. El + es un operando que viene en la sintaxis del lenguaje, y lo resuelve el compilador. El que haya cosas que no sean objetos tiene desventajas. Una rápida: en las colecciones no puedo meter números, porque las colecciones son de objetos, y los números no son objetos.

Mensaje

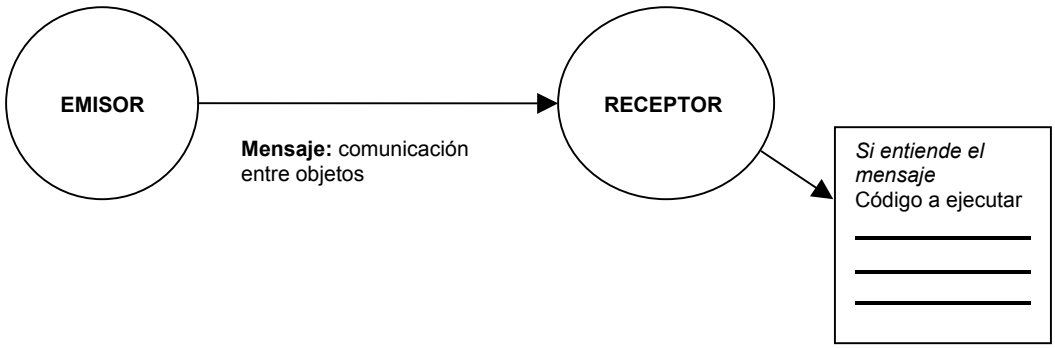
En una aplicación OO, entonces, tenemos objetos que se piden cosas entre sí. Esto nos lleva derecho al segundo concepto.

Un mensaje es una comunicación entre dos objetos emisor E y receptor R, E le pide algo a R y puede o no obtener algo como respuesta.

P.ej. en el Age of Empires, un renderizador le pide a una persona su posición. ¿Qué le devuelve la persona? ¡Un objeto! que representa la posición. Puede ser un objeto específico (de la clase Posicion), al cual le puedo pedir la coordenada x, en Smalltalk.
persona posicion x
o p.ej. un Array de dos posiciones, y yo sé que la primera es la X y la segunda es la Y.
persona posicion at:1
Pero siempre es un objeto.
¿Siempre? En Java no, puede ser un tipo básico.
Obviamente, los mensajes pueden llevar parámetros, p.ej.
persona irA: (Posicion x: 104 y: 232)

Planteo dos problemas, de los que vamos a hablar en un ratito:

1.
Un objeto ¿entiende cualquier mensaje que se le envía?
¡no!
¿Cuáles son los mensajes que entiende un objeto, cómo se determina eso?
Y asociada viene otra pregunta:
¿qué pasa si un objeto recibe un mensaje que no entiende, cómo se maneja esto?
Está claro que el mensaje se identifica por su nombre ("selector" en Smalltalk), más en general por una firma. ¿Qué diferencia hay entre un nombre y una firma? Pensar en un lenguaje que soporta sobrecarga, Java sí soporta sobrecarga. Puedo tener un mensaje
irA(int x, int y)
y otro distinto
irA(Posicion p)
Estos tienen igual nombre, pero distinta firma.
2.
Cuando un objeto recibe un mensaje que sí entiende, ¿qué pasa? Pasa que se ejecuta cierto código.
¿Qué código?
Se ejecuta el método correspondiente en la clase del objeto, o en alguna superclase. Más sobre esto en un ratito.



Dos problemas: ¿qué pasa si el receptor no entiende el mensaje? y ¿qué código se ejecuta?

Ambiente

Por lo general, nacen y mueren, o sea viven. En general, nacen porque alguien (¿quién? obvio, otro objeto) los crea, y mueren porque alguien (el "garbage collector") los destruye.

¿Dónde viven los objetos? (ya sé dentro de la computadora, ¿y si tenés una aplicación distribuida?).
Para dar cuenta del habitat de los objetos definimos

- ambiente: lugar donde viven los objetos. -

Crear un objeto = agregarlo al ambiente, destruirlo = sacarlo del ambiente.

Garbage collection: la capacidad de un ambiente de autoadministrarse, limpiando los objetos que no se necesita que estén, y liberando al código de la necesidad de administrar el ambiente.

La creación de un objeto ¿es un mensaje? Lo que importa es que tiene un aspecto distintivo, que es el agregado del objeto del ambiente. Eso viene con el ambiente, no es parte del código de una aplicación.
(para los curiosos: en Smalltalk existe un método de instancia new de la clase Behavior, que está resuelto con una primitiva que lo que hace es indicarle al ambiente que cree un objeto).
A la creación de un objeto se le puede pegar código, que es lo que uno hace en Smalltalk p.ej. en los métodos new de las clases. Ya veremos cómo es en Java.

Software

Con estos tres conceptos podemos decir qué es el software

Software: resultado de la interacción de un conjunto de objetos que viven en un ambiente y que se comunican entre sí enviándose mensajes.

El creerme esta definición de software tiene ciertas consecuencias, en particular veamos qué resulta ser "diseñar software", si creo que el soft es lo que acabo de decir.

¿Qué es diseñar software, más allá de la OO? es definir cómo se resuelve un problema mediante una aplicación, qué forma va a tener la aplicación que resuelve un problema.
Si el soft es lo que acabamos de decir, hay que definir

1. qué objetos vamos a incluir en la aplicación. P.ej. uno para cada persona del AOE[4].
2. qué mensajes va a entender cada objeto. P.ej. a las personas del AOE les puedo pedir la posición, eso me devuelve un objeto al cual le puedo pedir las coordenadas x e y.
3. cuáles son los patrones de interacción entre los distintos objetos. I.e.: cuáles son las secuencias de envíos de mensajes que van a aparecer en la aplicación. P.ej. cuando al renderizador le pido que actualice la pantalla, le va a pedir a cada persona su posición.

[4] AOE: Age of Empires

4. hay otro aspecto importante, que no apareció pero sí va a aparecer: cómo se van a conocer los objetos entre ellos. P.ej. el renderizador necesita acceder a las personas, ¿cómo va a hacer? Más sobre esto en un ratito.

Volviendo a las preguntas…

5. ¿Qué mensajes entiende un objeto? y ¿qué pasa cuando le intento enviar a un objeto un mensaje que no va a entender?
6. Cuando un objeto recibe un mensaje, se ejecuta código. ¿Qué código, dónde lo escribo?
7. ¿Cómo se crea un objeto?

En muchos lenguajes de programación OO, estas preguntas se responden introduciendo los conceptos de método y clase. Ahí vamos. Pero antes un comentario chiquito: fíjense que primero hablamos de objetos y mensajes y recién después apareció la idea de clase y método. Para pensar.

Clase Y Método

Método

Un método es la sección de código que se ejecuta al enviarse un mensaje. Se identifica con una firma (recuerden lo que hablamos hace un rato), que es la misma firma del mensaje enviado.
Entonces, cuando un objeto recibe un mensaje, se ejecuta un método cuya firma es la misma que la del mensaje.
¿Puede haber varios métodos con la misma firma?
Sí, eso es la base del polimorfismo (aunque para nosotros no todo, para que el polimorfismo tenga gracia hace falta algo más, que ya viene).
En Smalltalk es legal (¡y buscado!) que distintas clases tengan métodos con el mismo nombre. Eso tiene un límite, **un mismo objeto** no puede tener activos dos métodos con la misma firma, si no le envían un mensaje y no sabe qué hacer.
El lenguaje tiene que proporcionar el mecanismo de selección, de entre todos los métodos con la misma firma, de aquel que se va a ejecutar en ese caso. El proceso de selección/vinculación se llama binding.

"Binding dinámico" quiere decir que la selección se hace en tiempo de ejecución, en el momento que el objeto recibe el mensaje. Fíjense que depende del receptor, el emisor no tiene idea de qué método se va a ejecutar.
"Binding estático" quiere decir que la selección la hace el compilador, y (obviamente) se hace mirando el código, en particular el código que tiene el envío del mensaje.
Smalltalk ¿qué tiene? Dinámico. ¿Y Java, qué se les ocurre? Dinámico, dinámico, tan dinámico como el Smalltalk. ¿Eh, pero que sea tipado no influye? No, para nada, y ya lo vamos a ver. ¿Un ejemplo de binding estático? Los métodos no virtual en C++.

Clase

Una clase … es
1.
El molde a partir del cual se crean los objetos, pregunta 3. Si quiero crear una persona pongo
Persona new o new Persona()
el ambiente le pregunta a la clase qué características tiene que tener el nuevo objeto, p.ej. el formato de su estado interno (variables de instancia, a.k.a. atributos), y de alguna forma qué métodos va a tener disponibles el nuevo objeto.
Se dice que un objeto es instancia de una clase si fue creado usando esa clase como molde. Entonces
¿qué son las instancias? Los objetos
¿de cuántas clases es instancia un objeto? De exactamente una.
2.
El ente que determina el comportamiento de sus instancias. Contiene los métodos que se van a ejecutar cuando una instancia reciba un mensaje, pregunta 2.

1. Los mensajes que va a entender un objeto son los que tienen métodos en su clase ("su clase" = "la clase de la cual es instancia el objeto"), pregunta 1.
2. así se resuelve el polimorfismo.

¿Todos los mensajes que puede entender un objeto tienen un método en su clase? No, las clases se estructuran en una jerarquía de herencia … que ya conocen. Un objeto entiende los mensajes que tienen métodos en su clase o en alguna de sus superclases. Una subclase puede redefinir métodos, e invocar al método válido en la superclase. P.ej.

```
PosicionCorrida>>x
    ↑ (super x) + 4
```

Resumiendo: Clase

- molde para crear objetos
- determina comportamiento, contiene métodos.

Clase abstracta / método abstracto

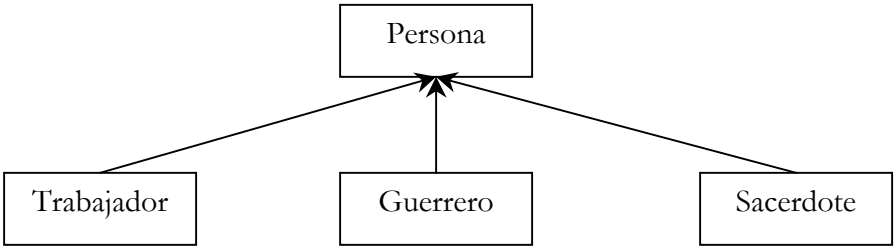
Una clase abstracta es una clase que se supone que no debería tener instancias. En Java hay un mecanismo del lenguaje que permite declarar que una clase es abstracta, y el compilador garantiza que no va a existir ninguna instancia.

> Entre paréntesis, este es el primer ejemplo de algo que Java quiere hacer: que las cosas que no deben pasar (en este caso, que exista una instancia de una clase abstracta) el mismo lenguaje permita garantizar que no pasen, y que salte en tiempo de compilación. Al mismo objetivo apunta el que Java sea tipado.

Obviamente, el chiste de una clase abstracta es que tenga descendientes concretos.
¿Por ejemplo? Object, Collection, Stream.
Método abstracto: se declara pero no se define (i.e. no se le pone código) en una clase, para que en los descendientes se defina, en cada subclase como corresponda.
La idea es que cualquier clase con algún método abstracto sea abstracta.

¿Para qué sirven estos conceptos? Para una técnica que se llama template method.

Supongamos que tenemos estas clases



y definimos que el valor de una Persona es la suma de su potencial ofensivo, su potencial defensivo y su energía.

En Smalltalk es fácil

```
Persona>>valor
    ↑ self potencialOfensivo + self potencialDefensivo + self energia
```

Ahora, el cálculo de los potenciales ofensivo y defensivo es distinto para Trabajador, Persona y Sacerdote. Entonces en Persona no puedo definir (p.ej) potencialOfensivo. Pero sí debería avisar de alguna forma que todos los descendientes concretos de Persona deben tener definido este método, para que a la persona le pueda preguntar el valor. En Smalltalk se hace así

```
Persona>>potencialOfensivo
    ↑ self implementedBySubclass
```

Está claro que Persona es abstracta, y que potencialOfensivo es abstracto en Persona.

¿Un ejemplo que conozcan de Smalltalk? Pueden mirar en su imagen preferida el select:, reject:, etc. que están todos definidos en Collection a partir del do:, y es el do: el que se define en cada subclase.

¿Cómo será en Java?
De alguna forma **tengo** que poner potencialOfensivo en Persona, si en Smalltalk no lo pongo no pasa nada, es sólo documentación, y esto está ligado con que Java es tipado. Vamos en un rato.

No quiero seguir sin remarcar una idea. ¿Para qué sirve que potencialOfensivo sea abstracto, dónde se ve la potencia del concepto? ¿En que Trabajador, Guerrero y Sacerdote tienen el mismo método, entonces hay polimorfismo? **No**, eso lo puedo tener sin necesidad de nada abstracto.
El poder está en que pude definir valor en Persona. **La utilidad está en el código que usa** las características abstractas, no en las carácterísticas en sí. Por favor, quédense con esta idea: la utilidad de algunas características abstractas está **en los usuarios**, el código o los otros objetos que usan a lo que estoy definiendo.

¡Ah! Dicho sea de paso, lo que acabamos de describir a la carrera es un Design Pattern, o sea, el template method es un pattern. Listo, tienen un ejemplo de pattern.[5]

Objetos sin clases

Programar en un ambiente OO consiste en escribir clases, que contienen los métodos donde está el código, y además la especificación del estado interno de los objetos (recordamos que todo objeto es instancia de alguna clase).

El código básico que uno espera que venga con el lenguaje (p.ej. la concatenación de Strings) viene dado por clases que vienen con el lenguaje. En el caso de Java, son las que vienen en el JDK. En los Smalltalk, son las que vienen en la imagen vacía de cada versión (y que no necesariamente coinciden, de hecho no coinciden).

Ahora, de alguna manera los conceptos de método y clase (y los que vienen atrás), dan soporte o implementan a los 3 conceptos básicos (los métodos son lo que se ejecuta cuando se envía un mensaje, la clase define qué mensajes entiende un objeto y cómo va a responder a cada uno). Si bien está fuera del alcance de la presente charla, nos gustaría decir que no es la única implementación/soporte posible (nuevamente, ¡chan!). Podemos llegar a prescindir del concepto de clase y quedarnos sólo con objetos, como en el proyecto Self (que es un Smalltalk sin clases).

Polimorfismo Y Tipos

El chiste del polimorfismo

Quiero volver un poquito al polimorfismo, que definimos a la pasada como la capacidad de que existan métodos distintos con el mismo nombre. Vemos que en la OO (al menos en los ambientes con clases) esto se implementa definiendo métodos en distintas clases.

Con esto solo, el polimorfismo es cómodo. P.ej., si tengo (en el AOE) edificios a los que tiene sentido preguntarle su potencialDefensivo, y armas a las que tiene sentido preguntarle su potencialOfensivo, no estoy forzado a poner nombres de métodos como

```
potencialOfensivoArma
potencialOfensivoPersona
```

sino que tanto las armas como las personas van a entender el mensaje potencialOfensivo, que tiene más onda, y además, al poner el mismo nombre, al que necesite usar armas y personas le queda claro que es el mismo concepto.

[5] ¡Felicitaciones! Una línea más en el currículum.

¿Listo con el polimorfismo, pasamos a otro tema? Lo que dijimos recién ¿es la ventaja importante para diseñar/programar OO que nos da este concepto? ¡No!

Lo más valioso que nos brinda la idea de polimorfismo es que al enviar un mensaje puede que no sepa, y en principio no me debería importar, qué método se va a ejecutar.
Esto es, a partir del código (o sea, del método) que contiene el envío del mensaje, no puedo determinar qué método se va a ejecutar, y también es posible que en distintas corridas del mismo código, el mismo mensaje resulte en la ejecución de métodos distintos.
Esa es también la gracia del binding dinámico, que en el momento de enviar un mensaje hay varios métodos que podrían ejecutarse, y que conviene hacer la selección en el momento del envío. Si para cada mensaje hubiera un único método posible, el binding dinámico no tendría sentido.

Con un ejemplo sin salirnos del AOE, supongamos que la clase Juego tiene un método que indica quién debe ganar el combate entre dos personas, una que ataca y la otra que defiende. En una versión muy sencilla, en Smalltalk queda así:

```
ganadorCombateEntre: atacante y: defensor
    ↑ (atacante potencialOfensivo > defensor potencialDefensivo)
      ifTrue: [atacante]   ifFalse: [defensor].
```

Recordemos que las subclases de Persona tienen cada una su implementación de potencialOfensivo y potencialDefensivo. ¿Cuál de los tres métodos potencialOfensivo, y cuál de los tres potencialDefensivo, se ejecuta al ser disparado por ganadorCombateEntre:y:?
Mirando solamente el código de ganadorCombateEntre:y: ¿lo puedo determinar? No.
En distintas invocaciones a este método, ¿es seguro que los métodos potencialOfensivo y potencialDefensivo que se invocan son los mismos? No, de hecho lo más probable es que varíen.
¿De qué depende qué métodos son invocados cada vez? Del atacante y del defensor, esto es, del receptor de cada uno de los dos mensajes.
¿Ven la potencia que tiene esto?

- Mañana puedo agregar una nueva subclase de Persona (qué se yo, Rey), con su propio cálculo de potenciales, y el Juego va a saber interactuar con un Rey, sin saber que es un Rey.
- Así como pasa con las Personas, mañana puede pasar con los Juegos, tener distintas versiones de Juegos, que ponderen distinto a la hora de decidir el ganador de un combate. ¿Dónde se ve la ganancia de eso? En los objetos que interactúen con los Juegos, como el polimorfismo de las Personas se ve en el Juego, que es quien interactúa con las personas. Otra vez: la utilidad se ve en el lugar donde se usan las características (en este caso, el mensaje polimórfico), mucho más que en donde se definen.

Repasamos el concepto de binding: selección del método para un mensaje. Acá se ve claramente que cuando el binding es dinámico, depende del receptor, es en tiempo de ejecución, y debe serlo, porque el código que contiene el envío del mensaje no puede determinar el método a invocarse.

Polimorfismo tipado y no tipado

Ahora escribamos este mismo método en Java, un poco para ir viendo algo de la sintaxis, un poco porque nos va a servir para lo que viene a continuación.

```
public Persona ganadorCombate(Persona atacante, Persona defensor) {
    if (atacante.potencialOfensivo() > defensor.potencialDefensivo()) {
        return atacante;
    } else {
        return defensor;
    }
}
```

Observen que hay que declarar el tipo de los parámetros, y el tipo de lo que se devuelve. Ahora vamos a hablar de las implicancias de esto, qué diferencias a la hora de desarrollar soft tiene con el Smalltalk, donde los tipos de los objetos no se declaran.

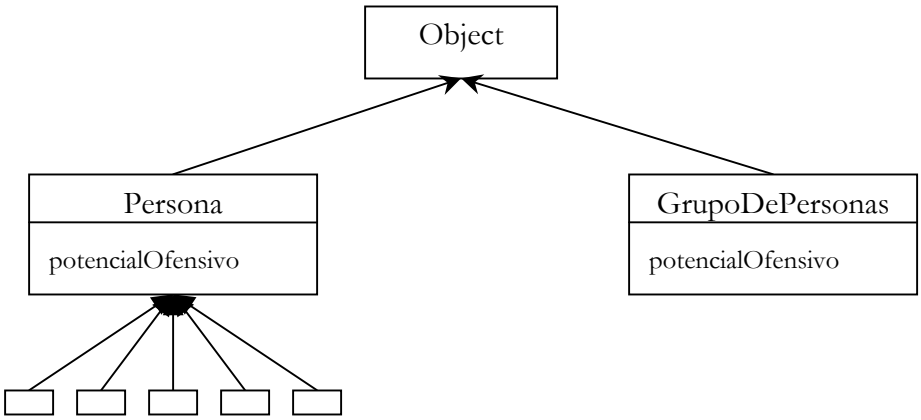
Empecemos por acá: ¿qué pasa si invoco a ganadorCombate pasando como primer parámetro un String?

Smalltalk:
```
j ← Juego new.
s ← "hola".
p ← Sacerdote new.
j ganadorCombateEntre: s y: p
```
se rompe en tiempo de ejecución, dentro de ganadorCombateEntre:y:, diciendo algo como
```
s doesNotUnderstand: potencialOfensivo.
```

Java:
```
Juego j = new Juego();
String s = "hola";
Sacerdote p = new Sacerdote();
j.ganadorCombate(s,p);
```
este código no compila, porque ganadorCombate espera una Persona, s es un String, y por donde se espera una Persona no puede pasar un String.
¿Qué es lo que sí puede pasar? Bien, una Persona o una instancia de un descendiente de Persona. Dentro de un rato vamos a ver que hay otra opción.

Las diferencias del ejemplo reflejan que Java es tipado mientras que Smalltalk no lo es.
Y se ve el argumento de venta del tipado: que muchos errores se manifiestan en tiempo de compilación, no hay que esperar a que se ejecute el código que invoca a ganadorCombate para descubrir el error. Esto genera soft más robusto, si se quiere trivialmente, porque hay casos que no pueden ocurrir (i.e. el lenguaje garantiza que no ocurran) y por lo tanto no hay que contemplar. Y si querés, hacen falta menos tests, hay algunos que no tienen sentido.
Entre paréntesis: otra vez la ventaja no se ve donde se declara/define una característica, sino donde se usa. Acá "la característica" consiste en tipar los parámetros de ganadorCombate.

Vamos ahora a otro caso. Supongamos que en el juego un grupo de personas puede atacar a una persona. Creamos una clase GrupoDePersonas que no va a extender (en principio) Persona, pero a la que sí tiene sentido preguntarle el potencialOfensivo. Tenemos



A uno le gustaría que un GrupoDePersonas pudiera atacar a una Persona, en Java:
```
Juego j = new Juego();
GrupoDePersonas gdp = new GrupoDePersonas();
Sacerdote p = new Sacerdote();
j.ganadorCombate(gdp,p);
```

¿qué pasa con este código?
- En Java: no compila, ¿por qué? Por la misma razón que con el String, por donde se espera una Persona no puede pasar un GrupoDePersonas.

- En Smalltalk: ¡todo OK!, como un GrupoDePersonas entiende el mensaje potencialOfensivo, y lo que devuelve entiende el mensaje >, la ejecución de ganadorCombateEntre:y: no trae problemas.

Entonces, en Smalltalk, ¿qué pasa, los objetos no tienen tipo? Sin entrar en la discusión de qué es un tipo y qué no lo es, sólo digo:

- en Smalltalk, el tipo de un objeto es exactamente el conjunto de mensajes que entiende -

o sea, para que un objeto pase por el primer parámetro de ganadorCombateEntre:y:, lo único que se le pide es que entienda el mensaje potencialOfensivo, cualquier objeto que cumpla eso pasa. Ahora, el "chequeo de tipos" se hace en tiempo de ejecución. Con lo cual obtenemos un sistema de tipos extremadamente flexible (que no te va a trabar nunca), pero en el que estás expuesto a errores de tipeo todo el tiempo.

En Java, el tipo de cada cosa (p.ej. parámetro de método) está declarado explícitamente. Por ahora, el tipo de cada objeto es su clase, y la relación de supertipo/subtipo está dada por la jerarquía de herencia. Por eso donde declaro que pasa una Persona puedo hacer pasar un Sacerdote. Por un lugar donde declaro una clase, puede pasar una instancia de esa clase, o de cualquier descendiente. En el ejemplo: declaro Persona, puede pasar un Sacerdote, no puede pasar un GrupoDePersonas.

Introducción a Interfaces, tipos básicos, asignación, tipo de una variable

¿Por qué dije recién que "por ahora" el tipo de un objeto es su clase? Porque en Java hay más cosas que pueden ser tipos: los tipos básicos, y la interfaz, que vamos a ver en un rato. Por ahora me interesa destacar que lo que declaramos (en el ejemplo, Persona como parámetro de ganadorCombate) no es una clase sino un tipo (p.ej. podríamos tener un parámetro int).

Ya que estamos, si en un lugar declaro que pasa un Object y envío un tipo básico ¿compila o no? No, los elementos de los tipos básicos no son objetos.
In extenso (si hace falta): si en una clase tengo un método

```
unMensaje(Object o)
```

y siendo obj una instancia de esa clase escribo

```
obj.unMensaje(3)
```

esto no compila.

Recién vimos que los parámetros de los métodos es un lugar en donde se chequean tipos, ahí mismo se declara el tipo de lo que se espera. ¿En qué otro lugar se les ocurre que hay chequeo de tipos?
En la asignación. Tiene que pasar que el tipo de la expresión del lado derecho tiene que pasar por el tipo declarado del lado izquierdo. P.ej.

```
Persona p = new Sacerdote();
```

compila, porque puedo hacer pasar un Sacerdote donde espero una Persona, pero

```
Persona p = new GrupoDePersonas();
```

no compila.
Pregunta: ¿qué pasa con

```
Sacerdote s = new Persona();  ?  No compila.
```

¿Y

```
GrupoDePersonas gdp = j.ganadorCombate(p1,p2);  ? Tampoco.
```

La variable tiene que ser Persona. ¿Otra opción? Object. Y hasta ahora no hay otra, poner (p.ej.) Sacerdote tampoco vale.
La asignación chequea tipos aunque la variable del lado izquierdo esté declarada más arriba, p.ej.

```
Persona p = new Sacerdote();
// … muchas cosas …
p = new GrupoDePersonas();
```

no compila por la segunda asignación de p.

Antes de pasar de tema, otra preguntita. Supongamos que los Sacerdotes entienden el mensaje curar(Persona), que el resto de las personas no entiende. ¿Qué pasa con este código?

```
Persona p = new Sacerdote();
Persona p2 = new Guerrero();
```

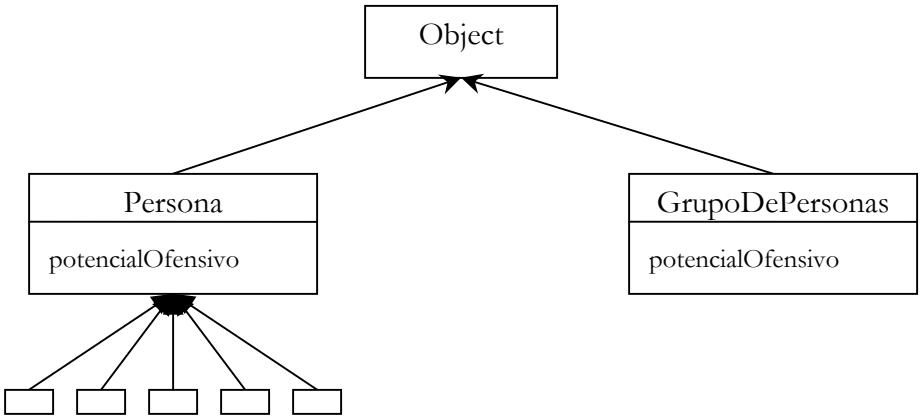
```
p.curar(p2);
```

esto no compila. Y este ejemplo me viene bien para dar un concepto que vamos a usar varias veces durante la materia.
En el momento de ejecutarse este código, la máquina virtual (es quien maneja los objetos) sabe que p está apuntando no a una Persona cualquiera, sino a un Sacerdote.
Pero el compilador lo único que puede asegurar de p es que es una Persona, y por lo tanto no puede garantizar que llegado el momento entienda el mensaje curar(Persona). El compilador ve solamente el lado izquierdo de la asignación.
Y como el que decide si este código compila o no es el compilador, esto no lo deja pasar.
En esquemático: en una asignación, lo único que ve el compilador es cómo está declarada la variable que se está asignando, el tipo es el de la izquierda, aunque pueda deducir el tipo de la expresión de la derecha, no usa esa info fuera del análisis de la misma asignación.

Interfaces

Qué es una interface

Volvamos al GrupoDePersonas. Recapitulando, tenemos este diagrama de clases

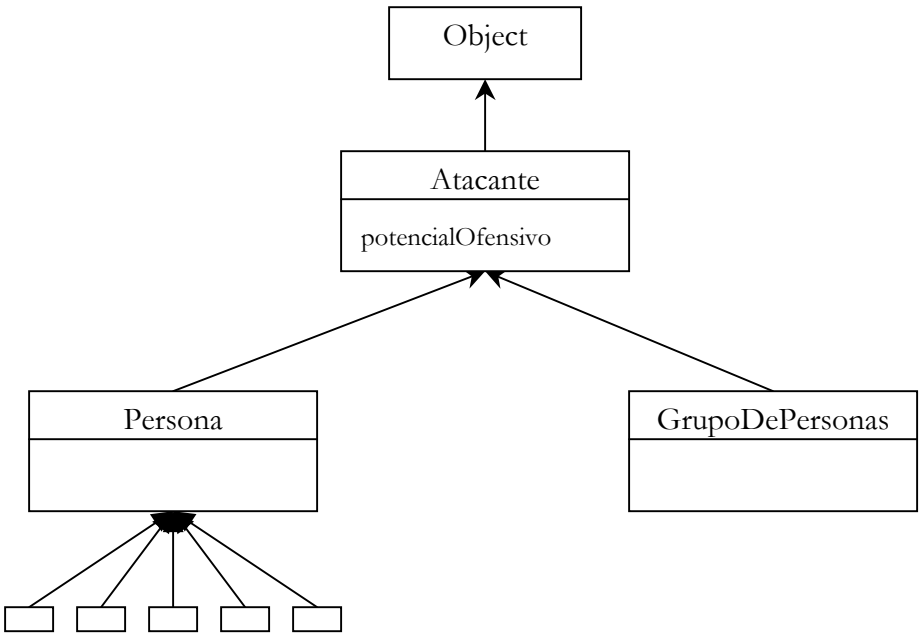


Y nos gustaría que esto compilara y funcionara correctamente:

```
Juego j = new Juego();
GrupoDePersonas gdp = new GrupoDePersonas();
Sacerdote p = new Sacerdote();
j.ganadorCombate(gdp,p);
```

Una opción es enganchar en la jerarquía de herencia las clases Persona y GrupoDePersonas, haciendo que alguna sea descendiente de la otra. Además de ser poco natural, podemos tener el problema que un GrupoDePersonas entiende mensajes que aplicados a una Persona no tiene sentido (p.ej. cantidad de integrantes) y viceversa (p.ej. la edad).

También se puede inventar una superclase que tenga métodos para los mensajes que deben entender ambos. Supongamos que lo único que tienen en común es potencialOfensivo. La superclase común podría llamarse Atacante, que es el rol que queremos que cumpla en ganadorCombate. Quedaría



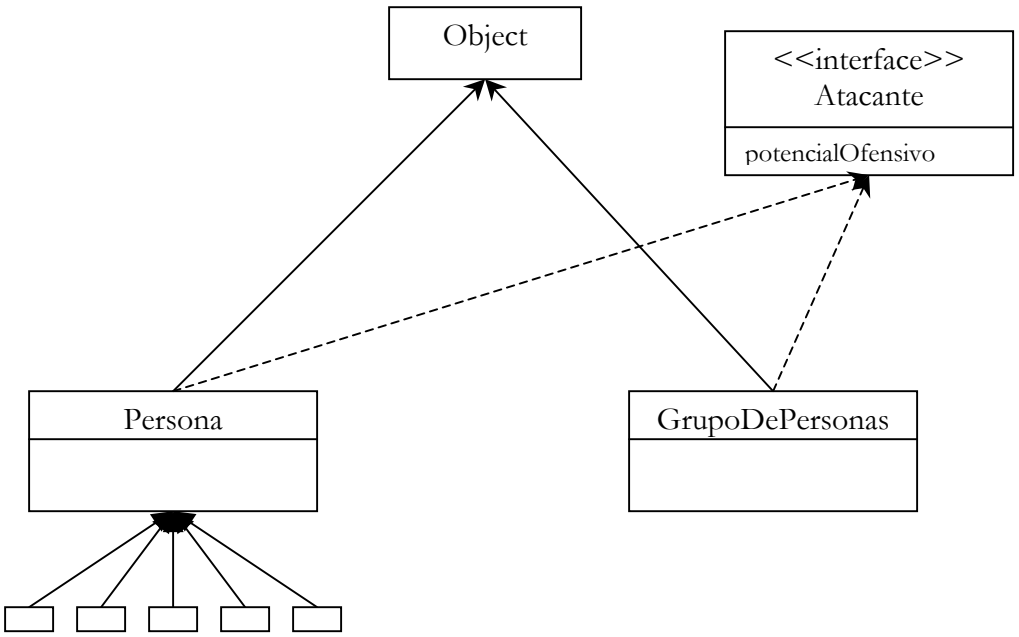
Y el método Juego.ganadorCombate nos queda

```
public Object ganadorCombate(Atacante atacante, Persona defensor) {
    if (atacante.potencialOfensivo() > defensor.potencialDefensivo()) {
        return atacante;
    } else {
        return defensor;
    }
}
```

Fíjense que el código no cambió, solamente cambiaron los tipos en el encabezado. Fíjense también que tuve que cambiar el tipo de lo que devuelve, porque ya no puedo garantizar que sea una Persona.

¿Cuál sería el código de atacante.potencialOfensivo()? Uno no sabría que ponerle, porque la forma de resolver potencialOfensivo es distinta entre las Persona (de hecho entre las distintas Personas ya dijimos que también cambia) y los GrupoDePersonas. Entonces Atacante es una clase abstracta, puramente abstracta; no tiene nada de código, es pura interfaz.

En Java hay otra forma de hacer esto, o sea, de declarar un tipo sin nada de código, sólo interfaz. Es el concepto de interface, que está en el lenguaje.
Rápidamente, una interface es un conjunto de declaraciones de métodos, sin código. Parecida a una clase abstracta en la que todos los métodos son abstractos.
Son tipos, o sea que se pueden declarar variables de tipo interface, y parámetros de tipo interface, y el tipo de retorno de un método puede ser una interface. Pero no están en la jerarquía de herencia de clases.
Sí tienen una jerarquía de herencia entre ellas, se puede decir que una interface hereda de otra, y eso quiere decir que la subinterface incluye todos los métodos declarados en la superinterface más los que declare explícitamente; esta jerarquía es aparte de la jerarquía de clases. En el dibujo nos queda



donde la línea punteada indica la relación entre una interface y las clases que la implementan, o sea que pueden pasar por ponde se declara la interface.
El código de ganadorCombate nos queda sin cambios, lo que declaro del primer parámetro es que tiene que cumplir la interfaz Atacante. No, no estoy poniendo ninguna restricción de clase, si mañana aparece una clase en cualquier lugar de la jerarquía que implemente Atacante, va a poder pasar por acá sin problemas.

Pregunta: con escribir la interface Atacante (ya vamos a ver la sintaxis) y cambiar ganadorCombate para que el primer parámetro sea un Atacante, ¿alcanza para que cualquier objeto que entienda (o que el compilador garantice que entienda) potencialOfensivo (el único método en Atacante) pase por el primer parámetro?
Hay que decirle a Java explícitamente qué clases implementan una interfaz. Eso se pone en el encabezado de la clase, p.ej. el de la clase Persona sería

```
public class Persona extends Object implements Atacante
```

la línea punteada en el diagrama representa el "implements Atacante". Si se pone en Persona y en GrupoDePersonas, el código en el que un GrupoDePersonas ataca a un Sacerdote finalmente compila.
Una clase puede implementar más de una interface, ejemplo en un ratito.

También compila lo siguiente:

```
Atacante a1 = new Sacerdote();
Atacante a2 = new GrupoDePersonas();
```

lo que no puede compilar nunca es

```
Atacante a3 = new Atacante();
```

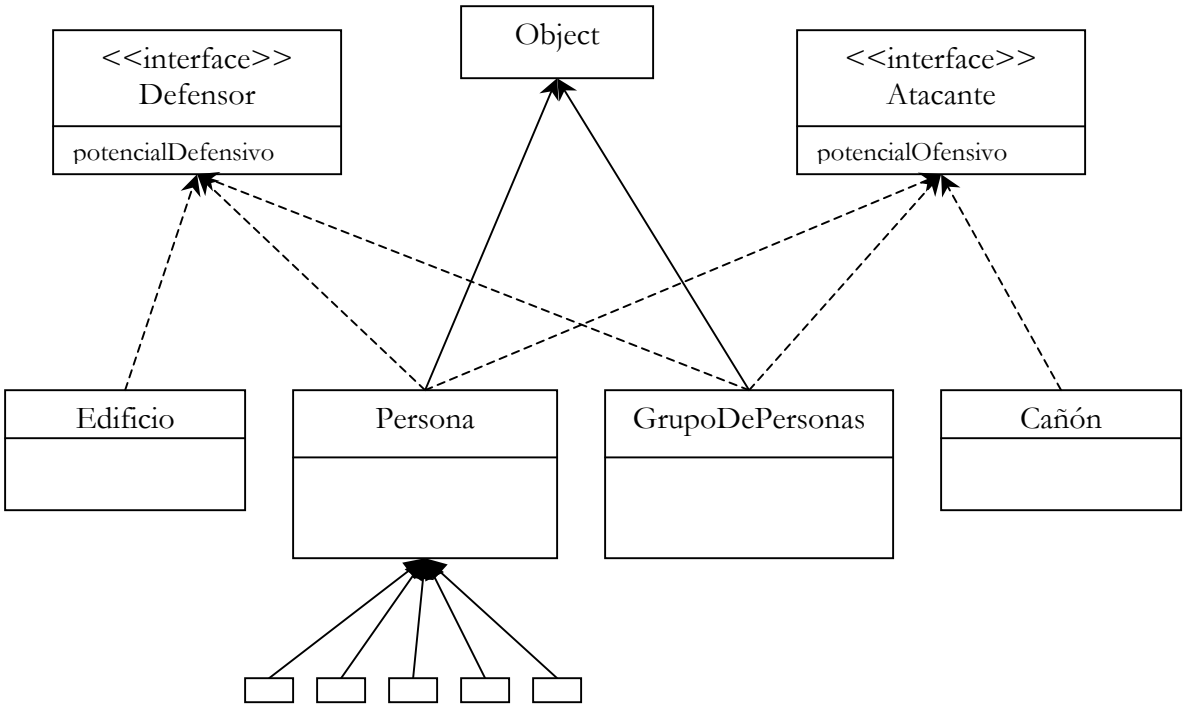
porque Atacante no es una clase, y los objetos son instancias de clases. Una interface no tiene instancias, hay objetos que cumplen con la interface, que son las instancias de las clases que la implementan.

Interfaces y herencia múltiple

Compliquemos un poco el ejemplo. Tengamos una clase Cañón, que puede atacar pero no defenderse, y otra clase Edificio, que puede defenderse pero no atacar. Las Personas pueden atacar y defenderse, supongamos que los GruposDePersonas también. Y queremos poder armar un combate entre cualquier objeto que pueda atacar y cualquiera que pueda defender.
¿Qué hacer?
Miremos el método ganadorCombate. ¿Qué hay que tocar? Que así como el primer parámetro es un Atacante, el segundo sea un Defensor, y que cualquier Defensor pase. Queda

public Object ganadorCombate(Atacante atacante, Defensor defensor) // …

Para que esto ande tenemos que definir una interfaz Defensor, y declarar qué clases la implementan. El diagrama queda



¡y todo funciona!

Acá tenemos clases que implementan más de una interfaz, p.ej. el encabezado de Persona ahora es

public class Persona extends Object implements Atacante, Defensor

En este caso la solución de la superclase común es impracticable, si invento una donde entren Edificio y Persona me queda afuera Cañón, y viceversa. Sí se podría haber hecho si Java soportara herencia múltiple, que es la capacidad de una clase de tener más de una superclase directa. C++ soporta herencia múltiple, otro lenguaje llamado Eiffel también, Java no, Smalltalk tampoco.

La herencia múltiple trae otras complicaciones, con las que no nos vamos a meter aquí. Sólo queremos decir esto: las interfaces de Java brindan una de las características de la herencia múltiple.

O sea, una clase no puede heredar de muchas clases, pero puede implementar muchas interfaces, entonces sus instancias (o las de sus descendientes) van a tener muchos tipos, y van a poder ser usados en cualquier lugar donde se declare alguno de esos tipos. Este es un uso que se le da a la herencia múltiple, y que en Java lo tenemos usando interfaces.

Lo que no podemos lograr (al menos naturalmente) es que una clase herede código de más de una clase. OK, viviremos con eso.

Qué se gana con las interfaces

Pensemos un poquito en lo que hicimos.

Básicamente, le cambiamos los parámetros a ganadorCombate, eran dos Persona, ahora son un Atacante y un Defensor (insisto en dejar para más adelante la cuestión del tipo de retorno).

¿Qué cambió? Parado en el primer parámetro, que ahora puedo usar ganadorCombate no sólo con Personas, sino con cualquier objeto, presente o futuro, que me garantice que entiende el mensaje potencialOfensivo. La garantía es que la clase implementa Atacante; Java obliga a cualquier clase que implemente Atacante a incluir el método potencialOfensivo, si no esa clase (p.ej. GrupoDePersonas) no compila.

¿Y entonces? Despegamos el polimorfismo de la jerarquía de herencia de clases. Recordemos que la característica más valiosa del polimorfismo es que en un contexto no sé qué método se va a ejecutar al enviarse un mensaje.
Sin interfaces, eso es cierto pero tengo una limitación: el receptor tiene que descender de la clase que declaro en el código (en ganadorCombate, era Persona) que tiene que tener ese método declarado, abstracto o no. P.ej. si en vez de Persona pongo Object, el método no compila. No sé qué método se va a ejecutar, pero sé que es alguno que cuelga de Persona, puede ser el de Sacerdote o el de Guerrero, pero no puede ser nunca el de GrupoDePersonas.
Con interfaces, me quito la limitación de encima. Ahora el método a ser invocado no tiene ninguna limitación relacionada con la jerarquía de herencia de clases. Alcanza con que el método esté en una clase que implementa la interface Atacante, y esas clases pueden estar en cualquier lado.
Las interfaces son tipos, desde este punto de vista, lo que estoy logrando es que un objeto tenga muchos tipos, que en un lenguaje tipado es lo que me permite que lo use en más contextos.
Un tipo es una característica que se condensa en un conjunto de mensajes que entiende un objeto, que es un concepto anterior a los lenguajes. En Smalltalk un objeto puede tener muchos tipos, no se declaran, es más flexible. En Java hay que declarar cada tipo con el que se va a trabajar, es más duro pero tenés la ventaja de que los tipos te quedan documentados en el código.
En resumen, las interfaces permiten explotar más el concepto de polimorfismo. Al dar patterns van a aparecer interfaces, y vamos a verlas más en acción.

Pensando más en diseño, al definir una interface estamos definiendo un concepto. En el AOE, estamos diciendo que existe un concepto que es el de atacante, algunos de los objetos que andan dando vuelta son Atacantes (o sea, cumplen con la interfaz Atacante), y otros no.
Así podrían definirse otras interfaces, p.ej. Defensor, Recolector, lo que sea.
Definir interfaces aumenta la comprensión del dominio, le da nombres a las cosas con las que estamos trabajando, hace que salten discusiones como “las instancias de tal clase, ¿deberían ser atacantes o no?”.
Está bueno que uno termine pensando, más que en las clases, en las interfaces que cumplen los objetos con los que trabajamos, en los tipos que tiene. Eso nos va a llevar a despegarnos de cómo están implementados los objetos, pensamos solamente en sus características públicas.
De esto vamos a hablar varias veces durante el curso.

Compilador y máquina virtual

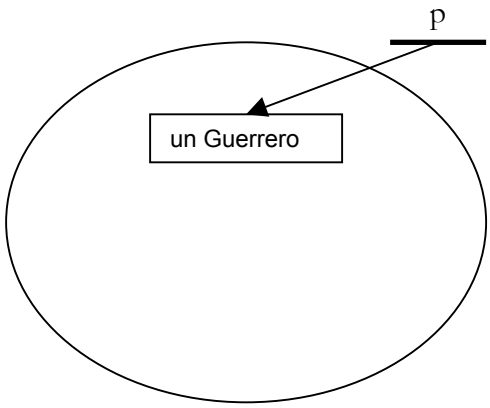
Antes de pasar a otro tema, una pregunta. Vieron que al hablar de las asignaciones hablamos de cuestiones que son propias de la máquina virtual (de qué clase es un objeto) y otras que son propias del compilador (de qué tipo es una variable, parámetro o expresión). El concepto de interface, ¿para quién es, para el compilador, para la máquina virtual, o para ambos?
Es para el compilador. Se definen interfaces para, justamente, explotar el polimorfismo más allá de los límites de la jerarquía de herencia de clases, para que el compilador acepte en determinados contextos objetos cuyas clases están desparramadas por la jerarquía de herencia, en el ejemplo, para que un GrupoDePersonas, un Cañón o un Edificio pasen por los parámetros de ganadorCombate.

La máquina virtual no usa el concepto de interface casi para nada. Los que hayan trabajado en C++ o en algún Pascal con objetos o parecido, se preguntarán ¿pero cómo logra Java esta magia, cómo hace el binding?
La respuesta es: porque la máquina virtual de Java es parecida a la de Smalltalk, en realidad es básicamente no tipada. El tipado se lo pone arriba el compilador, porque los que hicieron Java opinan que está bueno que un lenguaje sea tipado por lo que dijimos hace un rato, muchos errores saltan en tiempo de compilación.

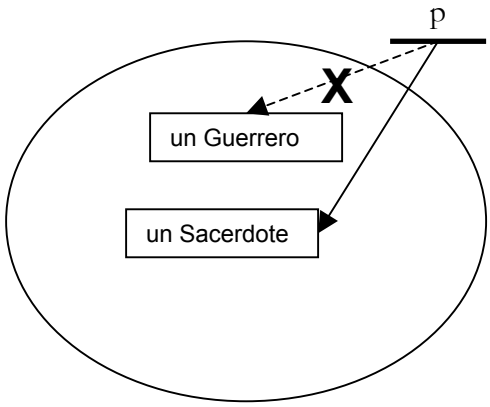
Referencias

Tomamos esta línea de código
    Persona p = new Guerrero();

¿Qué pasa a la derecha? Se le pide al ambiente que cree un nuevo objeto, que lo va a crear según el molde de la clase Guerrero.
¿Y después, en qué consiste la asignación? Las variables en Java (excepto las que son de tipos básicos) son nombres de referencias a objetos, como punteros. Y conceptualmente, están fuera del ambiente, no hay un objeto Variable que represente a p.
Tenemos



¿qué pasa si la siguiente línea es
    p = new Sacerdote();



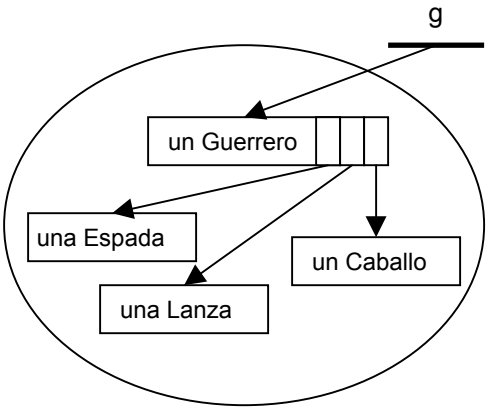
¿si en vez de
    Persona p
hubiéramos declarado
    Defensor p
qué habría cambiado en el dibujo? Nada, lo que cambia es el tipo de p. Los dos objetos creados son un Guerrero y un Sacerdote.
Entonces:
las variables son de tipos, que pueden ser interfaces, clases abstractas o clases concretas
los objetos son instancias de clases concretas
¡Ah! ¿qué pasa con el Guerrero que quedó sueltito? Bien, se lo llevará el garbage collector.

Ahora, algo más interesante. Supongamos que un Guerrero tiene dos armas y un transporte

```
public class Guerrero extends Persona {
    private Arma arma1;
    private Arma arma2;
    private Transporte transporte;

    // …
}
```

Y tengo una variable g que apunta a un Guerrero todo configuradito. ¿Cómo queda el ambiente?



Un objeto puede tener referencias a otros objetos. De hecho, es lo más común. Y le da mucha potencia a la OO, ¿por qué?
Porque el comportamiento de ese objeto, y de quienes lo usan (en nuestro ejemplo, el Guerrero y p.ej. el Juego que usa al Guerrero al calcular el ganador de un combate), se ven afectados por los objetos referenciados (en este caso, la Espada, la Lanza y el Caballo).

Un ejemplo: supongamos que el potencial ofensivo del guerrero se calcula así

```
public long potencialOfensivo() {
    return this.arma1.potencialOfensivo() + this.arma2.potencialOfensivo();
}
```

y hacés que un Guerrero ataque a un Edificio. Fíjense que

- el Juego le delega al Atacante, sin saber si es un Guerrero u otra clase, ahí hay una
- el Guerrero delega en sus armas, cada una de las cuales (Espada, Lanza, Pistola) puede tener su forma de determinar su potencialOfensivo.

Si en vez de ser un Guerrero, es otro tipo de Atacante, el cálculo cambia. Si el Guerrero en vez de p.ej. tener una Espada tiene una Gomera, también cambia.

Si lo que en un objeto A puede cambiar lo delega en otro objeto B, del cual tiene una referencia, con una interface X (o sea B cumple X) bien definida, cualquier objeto que cumpla con la interface X puede ser usado por A, y cambiando B por otro estoy logrando el cambio que quería en A.
Otra vez ¿dónde está la característica abstracta (cumplir con la interface X)? en B. ¿Dónde impacta? en A, donde se usa B.
¿De dónde viene la flexibilidad? De la combinación del polimorfismo (muchos objetos que cumplen X[6]) con el concepto de referencia (A tiene una referencia a un objeto, del cual lo único que sabe es que cumple X). De acá vamos derecho a …

[6] De ahí que algunos autores hablan de que dos objetos son polimórficos si los puedo intercambiar entre sí sin percibir cambios en la manipulación de esos objetos

El final

Hay muchas formas de verlo, pero nos quedamos con esta, que tenemos ganas de comentar en este momento:

buena parte de la potencia de la OO se obtiene explotando los conceptos de polimorfismo/binding dinámico y referencia.

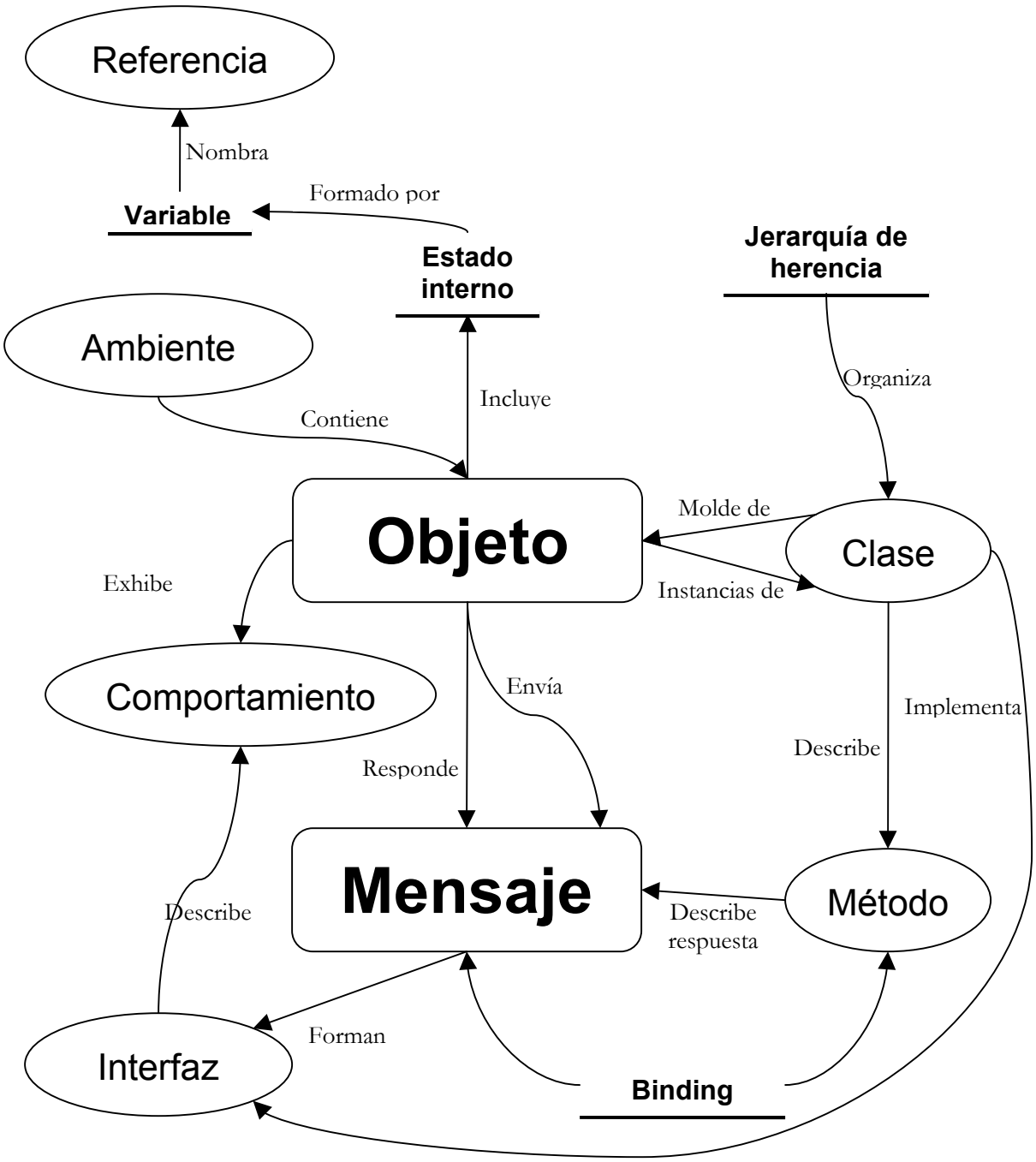
En Java, para aprovechar bien el polimorfismo, muchas veces conviene usar interfaces.

Arriesgo que varios de los patterns que vamos a ver tienen que ver con distintas formas de aprovechar estos dos conceptos.

Finalmente, está bueno no olvidar que el paradigma de objetos es sencillo, se basa en tres conceptos bastante intuitivos: objeto/mensaje/ambiente. Los métodos y clases son formas de esos conceptos. Hay objetos que se envían mensajes entre sí, y eso es todo, pero todo, lo que hay.

Anexos

El Dibujo



Principios De Von Neumann

(como aparecen en http://www.goldenmuseum.com/1403Neumann_engl.html, agregué lo que está entre paréntesis en 3.)
The essence of "Neumann's Principles" consists of the following:

1. The electronic computers should work not in the decimal but in the binary notation.
2. The program should be placed in one of the computer units, in the storage.
3. The program, as well as numbers (i.e. the data), with which the computer operates, is represented in the binary code, that is, the commands and the numbers are one-type by the form of representation.
4. The difficulties of the physical realization of the storage of the large speed and the large memory demand on the hierarchic organization of the memory.
5. The arithmetic device of the computer is constructed on the basis of the schemes executing the operation of addition.
6. The parallel principle of the computer process organization (the operations over the binary codes are realized simultaneously on all digits) is used in the computer.

(como aparecen en http://www.ucl.ac.uk/~uczcw11/modules/p028l02.ppt)
The Von Neumann Concept - The computer as a:

1. Sequentially-executed,
2. stored Program
3. machine

Un poco más in extenso:

- The architecture of a computer consists of three main parts: CPU, memory, and a connecting device (data bus) which can transmit a single data word at a time between the two.
- The memory is a linear vector of data words which are referenced by sequential addressing.
- The program which directs the activities of the CPU is stored in the same memory as the data which is to be manipulated by the program - the computer is a stored program machine.
- There is no distinction between data and instructions, nor indeed between one data type and another. There is no inherent meaning to the words which are stored in memory. The interpretation of a word is fixed only by the context at the time the word is fetched from memory.
- Since the device connecting the memory and CPU can only transmit one word at a time, instructions must be executed sequentially.

otro link que puede estar piola: http://www.csupomona.edu/~hnriley/www/VonN.html