Paradigma
Orientado a Objetos

Módulo 01: Objeto, mensaje, métodos.
Intro a method lookup.
Intro a referencias.

por Fernando Dodino
Versión 2.1
Agosto 2017

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Objeto
- 1.1 ¿Qué es un objeto?
- 1.2 Objetos conceptuales

2 Composición de un objeto

3 Mensaje

4 Ambiente

5 Referencias

6 Mensaje y método
- 6.1 Method lookup
- 6.2 Acciones y preguntas

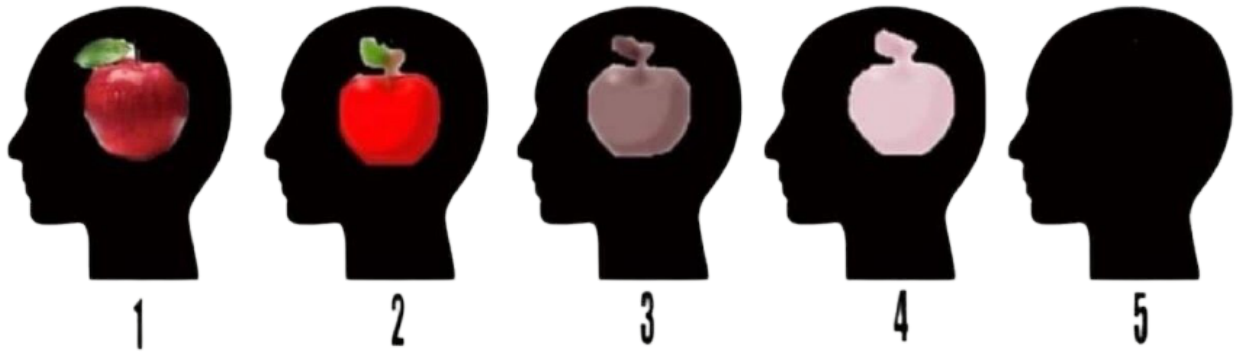
7 Resumen

1 Objeto

La principal abstracción de nuestro nuevo paradigma son los objetos, que representan distintos tipos de entidades.

1.1 ¿Qué es un objeto?

Es algo que puedo representar a través de una idea, un concepto. Tiene entidad. Por ejemplo, una fruta es algo del mundo real, que podemos conceptualizar fácilmente.



Representaciones mentales de un objeto real

Aquí vemos que nuestro modelo mental es una simplificación del objeto real. Dos personas pueden ver la misma cosa, sin embargo están abstrayendo representaciones diferentes porque tienen objetivos distintos. Lo que importa para el paradigma no es el objeto real, sino nuestro modelo mental.

1.2 Objetos conceptuales

La deuda que mantengo con otra persona, ¿es un objeto? Sí, también, porque es un concepto, no necesariamente tiene que ser un objeto *real*.

Un String, ¿es un objeto? “hola mundo” también es un objeto, al igual que un thread corriendo en la PC. Todo lo que existe en mi cabeza puede ser representado como un objeto.

Definamos entonces qué es un sistema en el paradigma orientado a objetos:

> "Un sistema es un conjunto de objetos
> que se envían mensajes para alcanzar un determinado objetivo"

Una definición similar a la que propone la Teoría General de Sistemas: "Conjunto de partes que se relacionan para un objetivo común".[1]

2 Composición de un objeto

Un objeto es una entidad, un concepto, un ente. ¿Qué es lo que nos importa de un objeto? Qué mensajes le puedo enviar, lo que forma la **interfaz**.

Ejemplo: el gato Félix sabe maullar, comer y dormir. Veamos su interfaz

3 Mensaje

¿Cómo interactúan los objetos entre sí? A través de los mensajes. ¿Cómo envío un mensaje a otro objeto?

1. primero lo tengo que conocer (más adelante veremos cómo)
2. cuando lo conozco, se qué mensajes le puedo mandar
3. ¿se cómo lo hace? no, no quiero saberlo, porque saberlo me ata a él (en términos de diseño decimos **"me acopla"**) y si yo asumo ciertas cosas en base a cómo lo hace, eso aumenta la posibilidad de tener que modificar cosas si el objeto al que le mando el mensaje cambia.

Otro ejemplo, nuestro caso testigo: pepita es un pájaro.
Mensajes → cosas que le puedo pedir a un objeto que haga

[1] Entonces un sistema en este paradigma no son solo objetos, **son objetos que se envían mensajes**. Según el principio holístico: "el todo es más que la suma de las partes", ya que el sistema no es solo sus partes sino también la interacción entre ellos

- energía (¿cuál es tu energía?)
- volá
- comé
- reiniciate (volvé a tu estado inicial)

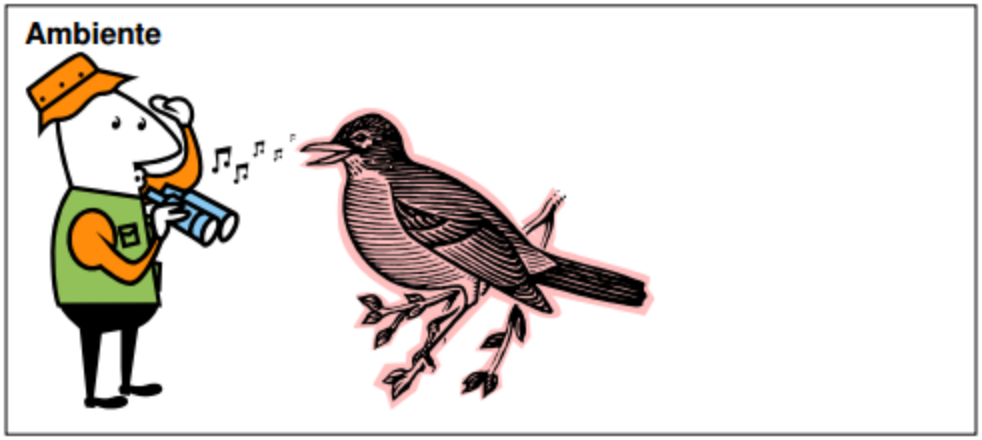
¿Qué hace volá y comé? El tema es dónde me ubico.

1. Si soy el que pide a pepita que vuele y que coma... No se exactamente lo que hace, pero se que quiero que vuele y que coma. No me interesa cómo está implementado porque yo estoy ocupado definiendo otras cosas. Ese beneficio se llama **abstracción** (concentrarnos sólo en lo que queremos resolver y dejar los detalles que no son esenciales de lado, eso nos permite mantener la complejidad lo suficientemente acotada).
2. Si soy pepita, yo sí sé cómo volar. Pero es algo mío, no tengo por qué andar contándole a todo el mundo que yo tengo energía y esas cosas. A mí me piden que vuele, vuelo. Me dicen que coma, como. Desde el lado de pepita, estoy encapsulando o agrupando funcionalidades propias. Pero no para protegerme, la implementación se puede ver en cualquier momento: yo sólo le estoy simplificando las cosas a mi cliente (donde cliente = el que me envía mensajes a mí).

4 Ambiente

Entonces tenemos: observador y objeto.

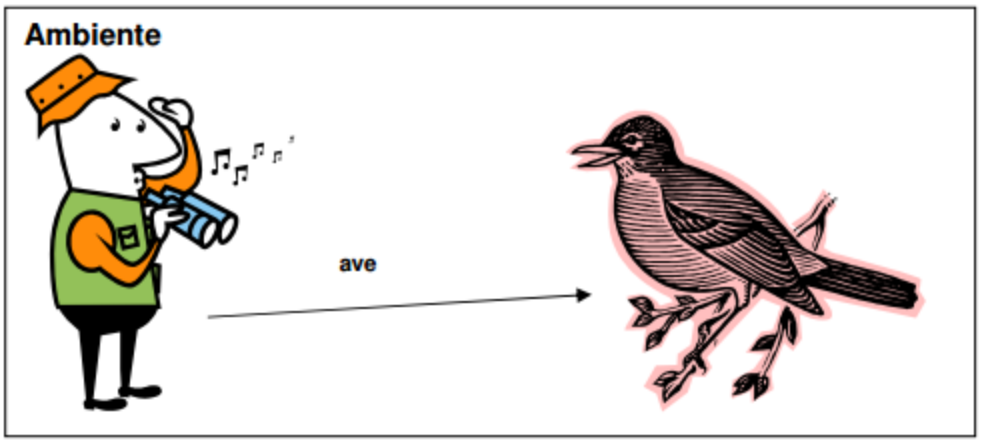
Pero ¿dónde viven esos objetos? Necesitaría un lugar (una especie de arenero donde pueda jugar con esos objetos). Entonces aparece la idea de un **ambiente**: el lugar donde viven los objetos.



La palabra ambiente es conocida en otras tecnologías como imagen, virtual machine o engine. Ya está, los metí en el arenero. Pero ya sabemos que para que uno pueda pedirle cosas a otro, necesito que se conozcan...

5 Referencias

¿Cómo hago que se conozcan? A través de referencias, y aquí surge el término **variable**. Una variable apunta o referencia a un objeto. Una variable necesita un nombre, que es la forma que tiene el observador de dirigirse al objeto observado.

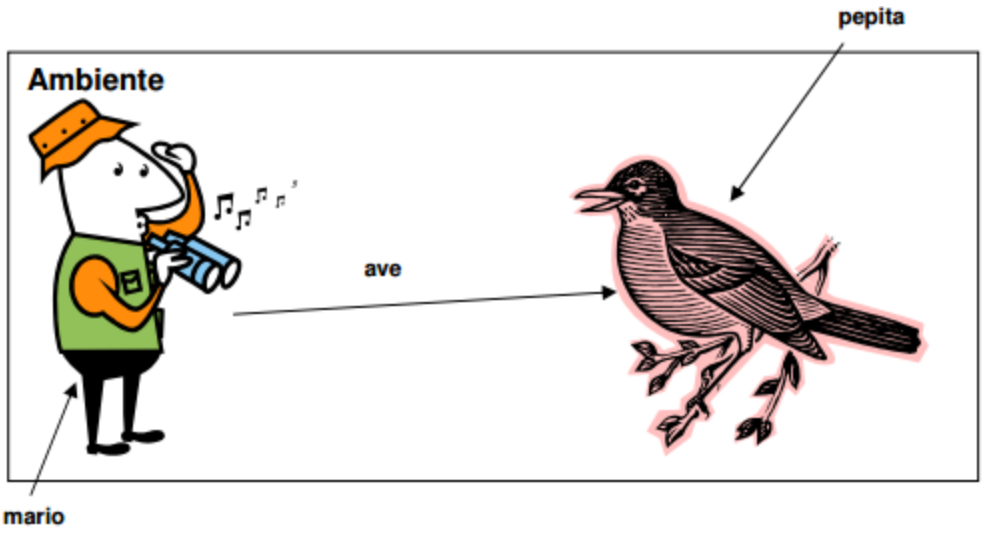


El observador le puede enviar un mensaje a pepita:

```
¡ave volá!
```

pero ¿no era que el ave se llamaba pepita? Lo que sucede es que puede haber varias referencias al mismo objeto. Para mí es pepita, para el observador es el ave, pero se

trata del mismo objeto:



En el módulo siguiente profundizaremos esta idea.

6 Mensaje y método

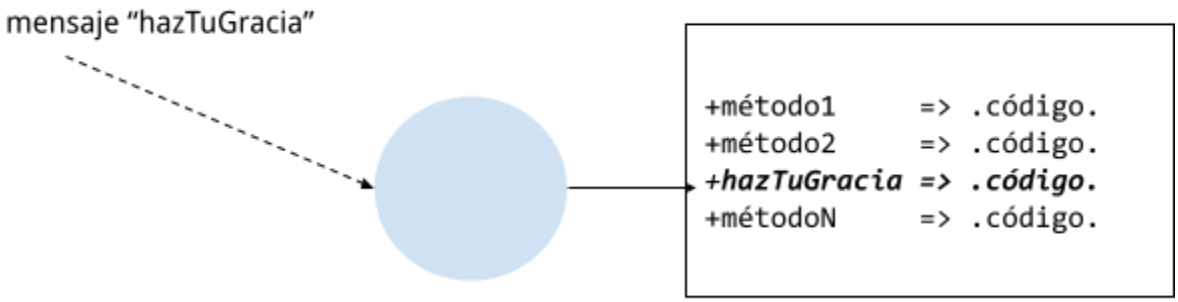
¿Qué pasa cuando mario le envía el mensaje volá a pepita? Pepita sale disparada por los aires. Bueno... y ¿en el software? Cuando le envío un mensaje a un objeto, se ejecuta código. ¿El código dónde se escribe? En el objeto receptor del mensaje. Este código recibe el nombre de método.

Entonces, pregunta de final: ¿qué diferencia hay entre mensaje y método?

Mensaje es lo que el objeto emisor le envía como orden al receptor. El emisor no sabe cómo se resuelve el mensaje, sólo pide que se resuelva (el qué y no el cómo, ¿a qué suena eso?). El receptor recibe el mensaje y se ejecuta un **método** (porción de código).

6.1 Method lookup

La estrategia que utilizan los lenguajes para resolver dónde está el código de un método al enviar un mensaje recibe el nombre de **method lookup**. En nuestro primer caso (el más simple) el código a ejecutar de un mensaje se busca en un método del objeto receptor de dicho mensaje.



method lookup básico
se busca en las definiciones del objeto un método "hazTuGracia"

6.2 Acciones y preguntas

Algunos mensajes responden a preguntas:

- pepita, ¿cuál es tu energía? ==> pepita.energia() ==> 10
- pepita, ¿sos feliz? ==> pepita.sosFeliz(ahora) ==> true
- pepita, ¿cuál es tu nombre? ==> pepita.nombre() ==> "pepita"

Mientras que otros mensajes simplemente disparan acciones que afectan al objeto en cuestión:

- pepita, ¡volá! ==> pepita.vola(10) ==> pepita se desplaza 10 kilómetros
- pepita, ¡comé! ==> pepita.come(2) ==> pepita come 2 gramos

En el primero de los casos nos importa el resultado, en el segundo nos importa más el **efecto colateral** que produce en el objeto.

7 Resumen

En este primer módulo conocimos los conceptos básicos del paradigma de objetos: la definición de objeto como representación de un concepto. Cada objeto define su interfaz, que son los mensajes que puedo enviar y que se traducen en métodos codificados.

Paradigma
Orientado a Objetos

Módulo 02: Referencias. Estado. Compartir objetos. Identidad.

por Fernando Dodino

Versión 2.1

Agosto 2017

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Repaso
- 1.1 Mensaje y método
- 1.2 Interfaz e implementación

2 Estado de un objeto

3 Breve introducción a Wollok
- 3.1 Primeros pasos en Wollok
- 3.2 Definir un objeto en Wollok

4 Accessors

5 Primeras pruebas en Wollok

6 Referencias en Wollok

7 Relaciones bidireccionales

8 Compartir valores

9 Referencias globales

10 Identidad

11 Resumen

1 Repaso

1.1 Mensaje y método

Mensaje y método no es sólo un juego de palabras. Me define de qué lado me pongo:

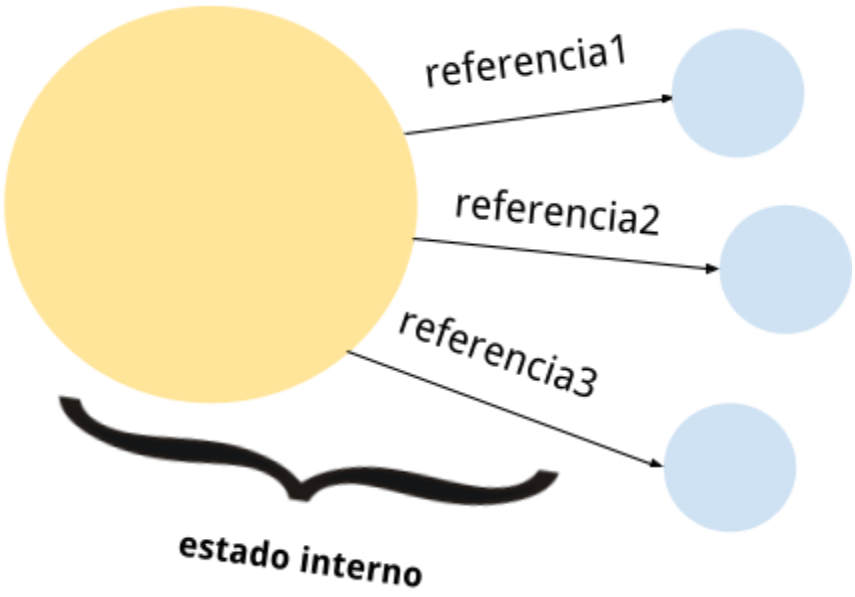
- **Mensaje**: soy usuario del objeto. Soy un observador que pido al objeto que haga algo (no quiero saber cómo lo resuelve, sólo quiero que lo haga).
- **Método**: yo soy un objeto, ejecuto código que está en mi definición, más allá de quién lo haya pedido.

1.2 Interfaz e implementación

- **Interfaz** es lo que un objeto publica para que otro objeto lo use (es el qué).
- **Implementación** es lo que un objeto encapsula para definir cómo se termina resolviendo un mensaje.

2 Estado de un objeto

Hemos visto que un objeto puede enviar mensajes a otros objetos, para lo cual los tiene que conocer. El conjunto de referencias que tiene un objeto representa su **estado**.



Al modelar una solución con objetos, buscaremos que el estado de un objeto sea manipulado únicamente por métodos definidos en el propio objeto. Cada referencia define un **atributo**, que tiene un nombre y un valor, el objeto apuntado.

3 Breve introducción a Wollok

Wollok es un lenguaje de programación **con fines didácticos**

- no obliga al desarrollador a definir tipos para los objetos
- es interpretado
- provee un entorno integrado de desarrollo o IDE donde trabajar

3.1 Primeros pasos en Wollok

Lo primero que hay que hacer es generar un nuevo proyecto Wollok:
Nuevo > Proyecto Wollok... escribimos el nombre del proyecto y aceptamos.
Luego, sobre el project, en la carpeta *src* hacemos botón derecho Nuevo > Archivo de Wollok... y darle un nombre al archivo con extensión *.wlk*

3.2 Definir un objeto en Wollok

Veremos un ejemplo que muestra la sintaxis básica: definimos a pepita, una golondrina que sabe

- volar: esto disminuye su energía en 10 unidades
- comer una cantidad (de alpiste en gramos): aumenta su energía en 2 unidades * la cantidad que coma

```
object pepita {
  var energia = 100

  method volar() {
    energia = energia - 10
  }

  method comer(cantidad) {
    energia = energia + 2 * cantidad
  }
}
```

Algunas observaciones:

- `object pepita`: define un nuevo objeto de nombre pepita, entre llaves deben definir atributos y comportamiento
- aquí vemos que no es obligatorio definir los tipos de los parámetros ni de los métodos que codificamos. ¿Qué pasa si un objeto no puede resolver un mensaje? El *method lookup* falla y aparecerá un mensaje de error.
- `var energia = 100`: define una referencia o atributo energia, con valor inicial 100 (un valor literal de *tipo* número).
- `method volar()`: define un método volar, sin parámetros. Entre llaves podemos ver el comportamiento.
- en el método comer vemos que recibimos un parámetro cantidad, que lo necesitamos para calcular la cantidad de energía que debemos aumentar.

4 Accessors

Como vimos en el párrafo Estado de un objeto, no es aconsejable que quien use a pepita manipule directamente sus variables, sino que debe enviarle mensajes que accedan o modifiquen las referencias a los objetos que conoce. Para ello existe cierto tipo de mensajes llamados **accessors**, cuyo fin es publicar la referencia (getter) o modificar dicha referencia (setter).

En el mismo ejemplo de pepita vamos a mostrar cómo definir un método getter y uno setter:

```
object pepita {
    ...

    // getter
    method energia() { return energia }

    // setter
    method energia(_energia) { energia = _energia }
```

(ésta es la convención por defecto de Wollok)

El getter se define como un método de una sola línea, que devuelve algo.
El setter no, es un método que modifica el estado interno del objeto, no tiene necesidad de devolver nada.

5 Primeras pruebas en Wollok

Ahora que creamos un objeto pepita, podemos enviar mensajes en la consola interactiva: parado sobre el archivo .wlk que creamos recientemente, botón derecho Run As > Launch with REPL[1]

```
Wollok interactive console (type "quit" to quit):
>>> pepita.energia()
100
>>> pepita.volar()
>>> pepita.energia()
90
>>> pepita.energia(170)
>>> pepita.energia()
170
```

La sintaxis en general es

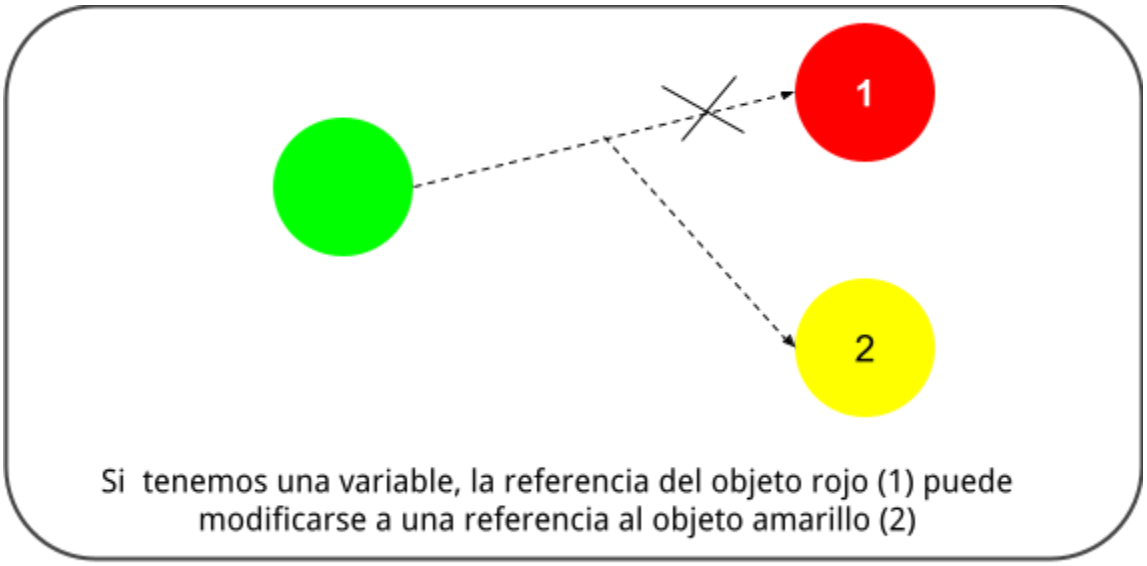
```
objeto.mensaje()
objeto.mensaje(parametro1)
objeto.mensaje(parametro1, parametro2)
```

6 Referencias en Wollok

Wollok tiene dos maneras de definir referencias

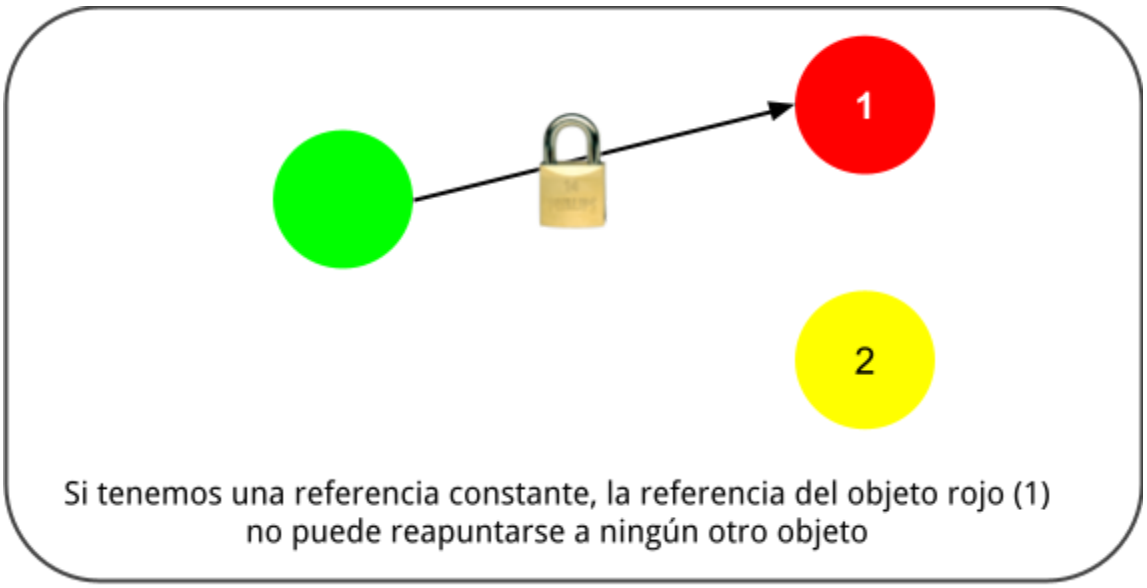
- **variables (var):** es una referencia que puede cambiar el objeto al que apunta
- **valores (const):** es una referencia fija a un objeto, que no puede cambiar (una vez inicializada no es válida una nueva operación de asignación)

[1] El ejemplo es didáctico, en el futuro preferiremos tener métodos como volar() y comer() que van a modificar la referencia energía, en lugar de setearla desde otro lugar que es justamente exponer el estado interno hacia los objetos que lo usan.



Si tenemos una variable, la referencia del objeto rojo (1) puede modificarse a una referencia al objeto amarillo (2)

```
var age = 10
age = 11
age = age + 1
```

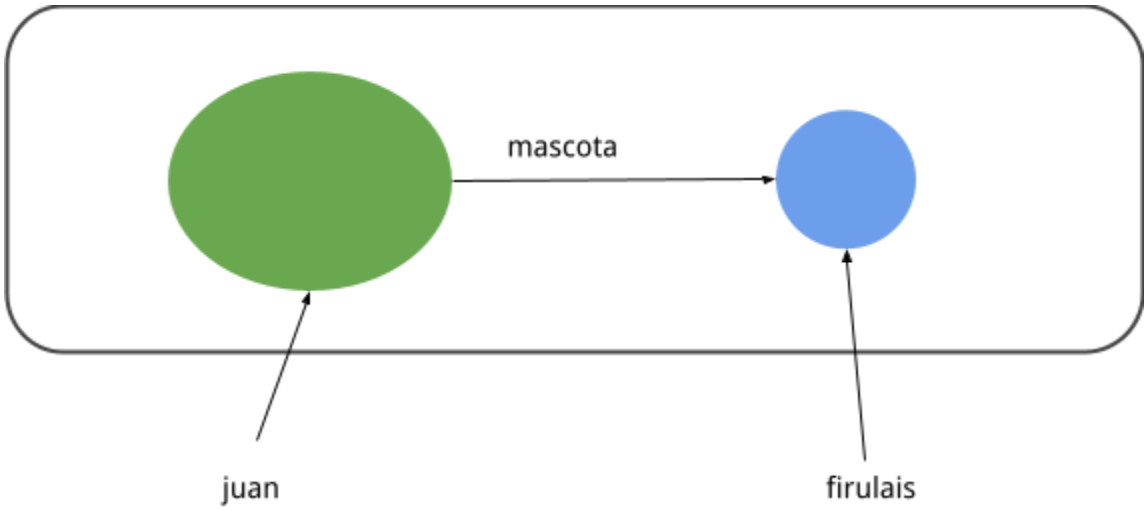


Si tenemos una referencia constante, la referencia del objeto rojo (1) no puede reapuntarse a ningún otro objeto

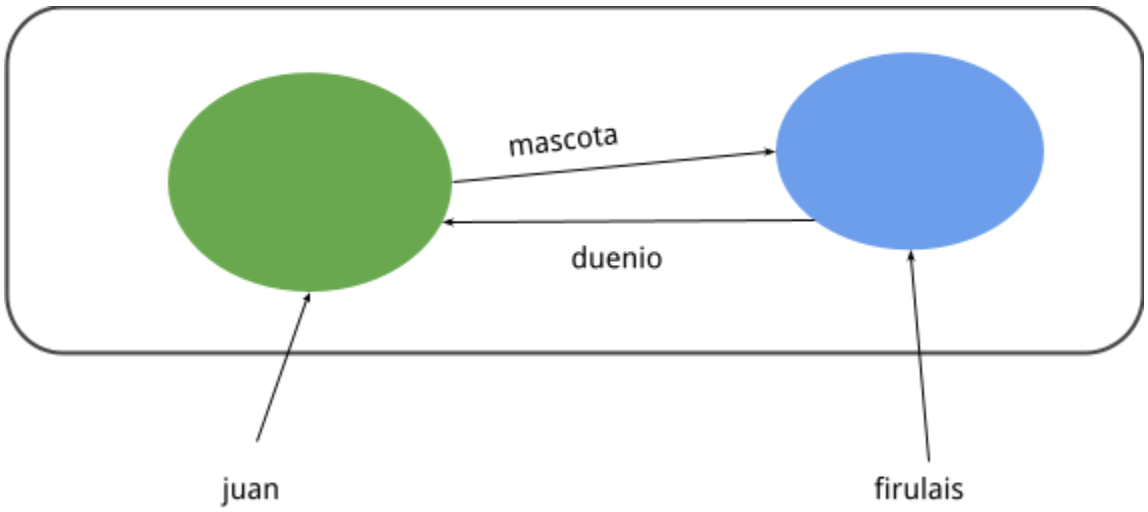
```
const adultAge = 21
adultAge = 18  // ESTO DA ERROR
```

7 Relaciones bidireccionales

Supongamos que tenemos el siguiente escenario: Juan tiene una mascota Firulais. Esto se grafica de la siguiente manera:



Esto implica que juan conoce a firulais, pero firulais no conoce a juan: la relación es unidireccional. Para establecer una relación bidireccional, necesitamos incorporar un atributo dueño en firulais:



Hacemos esto en Wollok:

```
object juan {
  var mascota

  method mascota(_mascota) { mascota = _mascota }
}
```

```
object firulais {
  var duenio

  method duenio(_duenio) { duenio = _duenio }
}
```

En la consola interactiva escribimos

```
Wollok interactive console (type "quit" to quit):
>>> juan.mascota(firulais)
>>> firulais.duenio(juan)
```

Tener una relación bidireccional

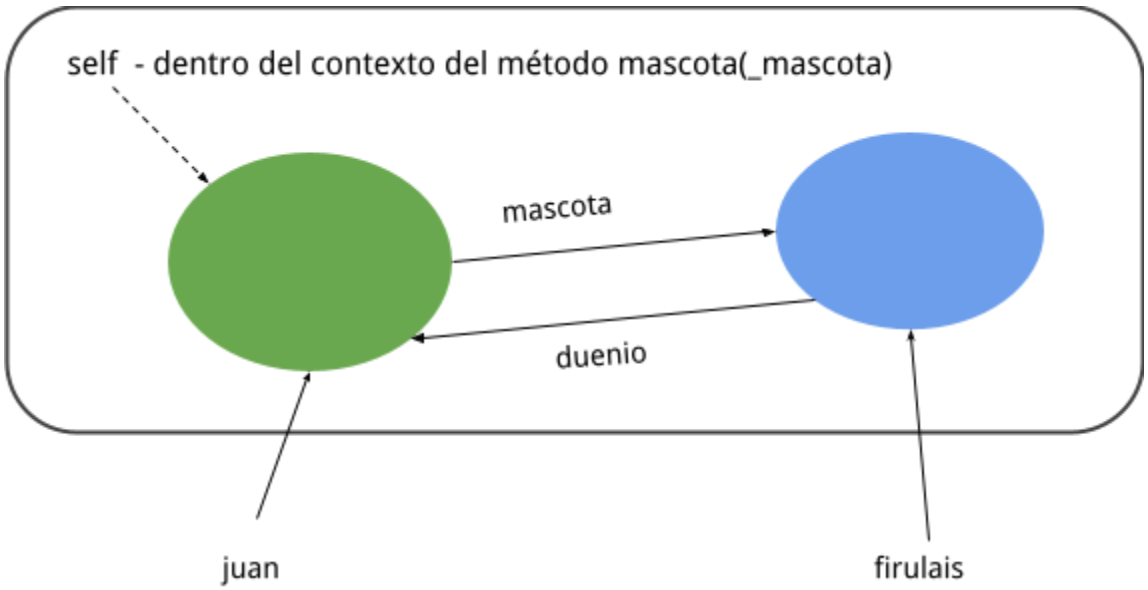
- es cómodo para **usarlo**, porque cualquiera de los dos objetos puede enviar un mensaje a otro
- pero requiere mantener ambas referencias sincronizadas, para evitar inconsistencias

Una alternativa puede ser que uno de los objetos sea el responsable de actualizar la otra referencia:

```
object juan {
   var mascota

   method mascota(_mascota) {
      mascota = _mascota
      mascota.duenio(self)
   }
}

object firulais {
    var duenio

   method duenio(_duenio) { duenio = _duenio }
}
```

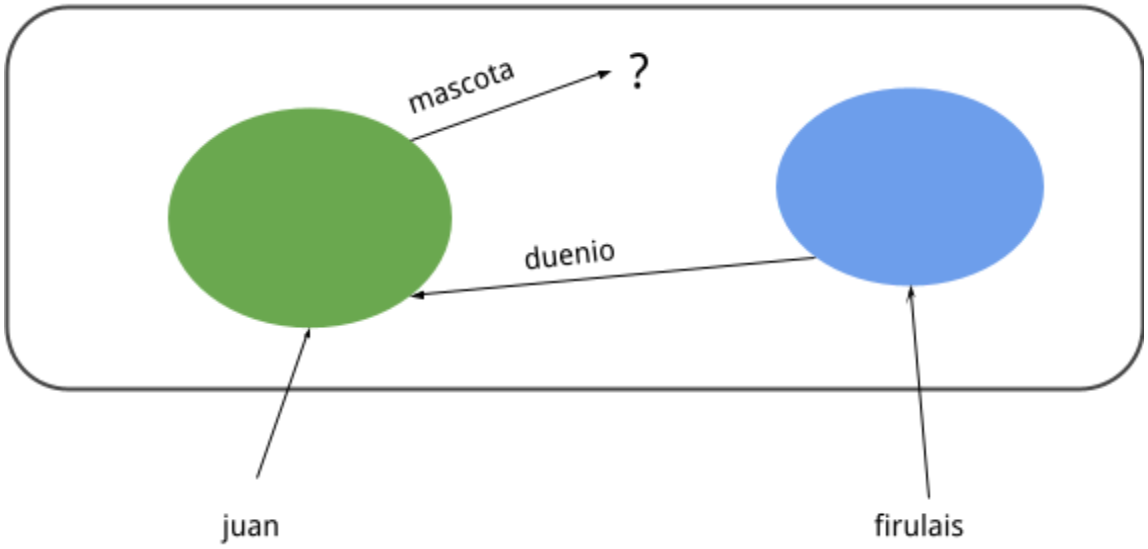
¿Qué es self? Dentro del contexto de un método es una referencia al propio objeto donde estamos ubicados.

```
Wollok interactive console (type "quit" to quit):
>>> juan.mascota(firulais)
```



Esta solución tiene como desventaja que si enviamos el mensaje duenio a una mascota, podemos dejar una referencia sin definir en el dueño.

```
Wollok interactive console (type "quit" to quit):
>>> firulais.duenio(juan)
```



8 Compartir valores

¿Qué pasa si juan y tefi quieren compartir actividades con firulais? Le pasamos la referencia al mismo objeto:

```
object tefi {
   ...

   method jugarCon(_mascota) { ... }
}

object juan {
   ...

   method alimentarA(_mascota) { ... }
}
```

Si tenemos los objetos tefi y juan...

```
Wollok interactive console (type "quit" to quit):
>>> tefi.jugarCon(firulais)
>>> juan.alimentarA(firulais)
```

9 Referencias globales

Tanto juan, como tefi y firulais son **objetos autodefinidos**, o *well-known objects* (wko). Entonces cualquier objeto podría enviarle un mensaje sin necesidad de guardar una referencia propia:

```
object firulais {
  method jugar() {
     //
  }
}

object juan {
  method jugarConMascota(){
     firulais.jugar()
  }
}
```

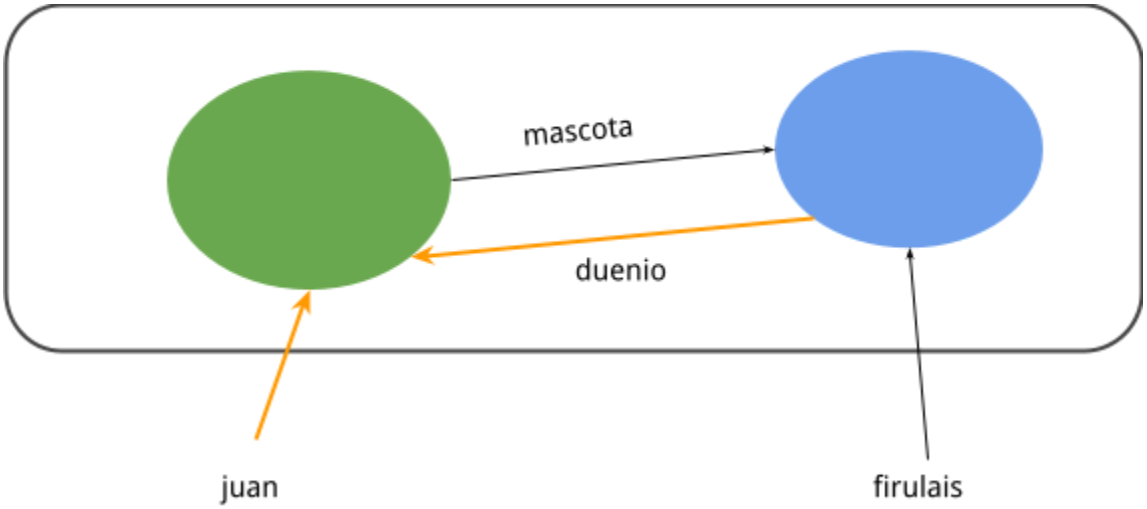
Ésta **no es una buena práctica**, porque ahora Juan ¡solo puede jugar con firulais! Acortar el camino utilizando los objetos autodefinidos como referencias es tentador, pero el costo que tiene cualquier acceso global es que nos lleva a

soluciones rígidas, donde no es posible modificar el objeto al cual queremos enviarle el mensaje. Por eso por más que nos parezca burocrático, es siempre preferible que definamos las referencias de un objeto como locales a ese objeto (mediante *var* o *const*).

10 Identidad

En un sistema con objetos, cada objeto sabe que es él y ningún otro objeto más, por eso es el responsable de responder ante un mensaje. En el ambiente cada objeto tiene su propia **identidad**. En un sistema orientado a objetos es frecuente tener diferentes referencias y querer determinar si estamos hablando de un objeto u otro, entonces la definición de identidad nos dice que

Dos referencias son idénticas si apuntan al mismo objeto



En el ejemplo anterior, las referencias juan y duenio son idénticas, porque apuntan al mismo objeto

11 Resumen

Hemos visto que los objetos tienen estado interno, conformado por los atributos que son nombres de referencias hacia otros objetos. Las referencias pueden cambiar el objeto al que apuntan (en cuyo caso son referencias variables) o no (constantes). Cada referencia marca qué objeto conoce a otro, por eso para que la relación sea bidireccional necesitamos dos referencias entre los mismos objetos. Y además un objeto tiene identidad, que es lo que lo distingue de los demás objetos.

Paradigma Orientado a Objetos

Módulo 03: Introducción a Polimorfismo.

por Fernando Dodino

Versión 2.1

Agosto 2017

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Introducción al Polimorfismo
- 1.1 Analogía
- 1.2 Repaso mensaje y método
- 1.3 Breve ejemplo
- 1.4 Polimorfismo y Tipos

2 Resumen

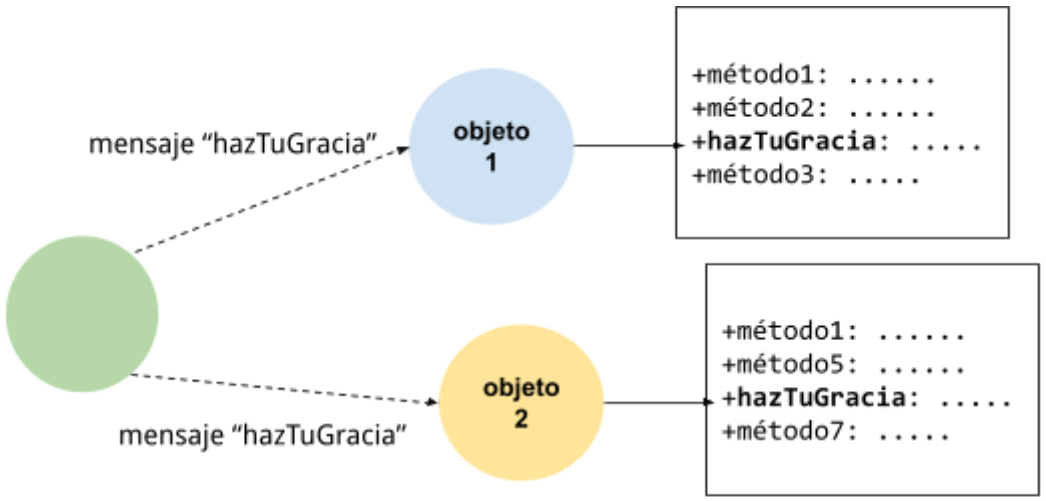
1 Introducción al Polimorfismo

¿Qué pasa si dos objetos son capaces de responder a un mismo mensaje?
El que manda el mensaje puede cambiar la referencia de un objeto a otro sin notar la diferencia.

Se produce entonces un concepto fundamental en el paradigma, cuando tenemos 3 objetos

- un objeto que envía un mensaje
- y dos objetos que implementan el mensaje de dos maneras diferentes (cada uno define un método distinto)

El observador puede cambiar la referencia sin preocuparse por el envío del mensaje, que permanece sin cambios. Decimos entonces que los dos objetos que comparten cierta interfaz en común son **polimórficos** para ese observador.



polimorfismo: dos objetos son polimórficos a la vista de un tercero porque puede enviar el mismo mensaje sin importar cómo estén implementados los métodos

Ojo, no es necesario que los dos objetos polimórficos definan exactamente los mismos métodos. Alcanza con que implementen al menos un mensaje en común (como en el ejemplo el método hazTuGracia).

1.1 Analogía

Para ilustrar este concepto utilizaremos algunas analogías, veamos el botón *Play* en cada uno de estos artefactos...

AT&T
9:42 AM
Done
00:00
Loading Movie...
-01:28
YouTube

SAMSUNG
VCR & DVD PLAYER
DVD
VIDEO
OPEN/CLOSE
VHS
DVD-V9650
EJECT
POWER
DOLBY DIGITAL
DIVX
12:48:36
CHANNEL
DVD/VCR
HD
REC

PAUSE
REW
FF
PLAY
STOP

AUTO STOP
COUNTER
SAVE
commodore
RECORD
PLAY
REWIND
F. FWD
STOP
EJECT

Todos los artefactos entienden el mensaje ▶ . Es decir que si lo lleváramos a una representación en objetos, podríamos enviarles a todos el mismo mensaje:

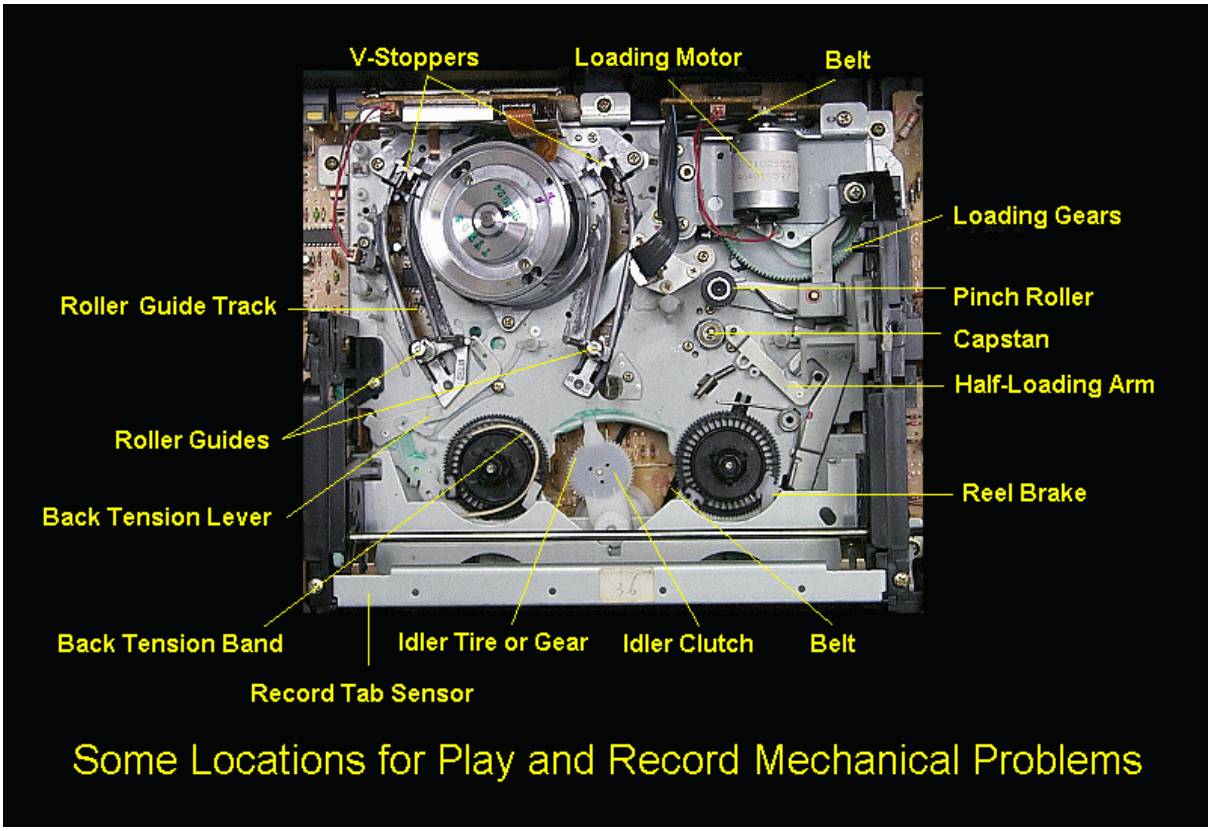
```
Wollok interactive console (type "quit" to quit):
>>> ipod.play()
>>> dvdLiving.play()
>>> vhsRecorder.play()
```

No obstante, está claro que cada artefacto hace cosas diferentes, porque sus implementaciones son diferentes (los métodos no son iguales). Por otra parte, no podemos pretender que funcionen igual, y no lo harán: un video reproductor no puede grabar, de la misma manera que no podemos ir a la pista 8 de un cassette. Pero sí es útil que varios objetos compartan un mensaje, para que alguien use a esos objetos dentro de algún contexto.

> ¿Para quién es bueno eso? No para los objetos que son polimórficos, sino **para quienes usamos esos artefactos**.

1.2 Repaso mensaje y método

¿Por qué no quiero ver cómo se implementa el método en cada objeto?



Ah, cierto. La complejidad.

1.3 Breve ejemplo

Tenemos dos empleados de un negocio que vende empanadas, Galván y Baigorria. Galván cobra un sueldo fijo (hoy es 15.000, pero eso puede modificarse). Baigorria cobra 15 pesos por cada empanada que vende. Generamos el ejemplo en Wollok[1]:

```
object galvan {
	var sueldo = 15000

	method sueldo() = sueldo

	method sueldo(_sueldo) { sueldo = _sueldo }
}

object baigorria {
	var cantidadEmpanadasVendidas = 100
	var montoPorEmpanada = 15

	method venderEmpanada() {
		cantidadEmpanadasVendidas = cantidadEmpanadasVendidas + 1
	}

	method sueldo() = cantidadEmpanadasVendidas * montoPorEmpanada
}
```

Lo probamos en un REPL:

```
>>> baigorria.sueldo()
1500
>>> baigorria.venderEmpanada()
>>> baigorria.sueldo()
1515
>>> galvan.sueldo()
15000
```

Vemos que tanto a baigorria como a galvan podemos preguntarle el sueldo. No nos importa cómo lo resuelve cada uno, pero sí que en algún punto comparten la misma interfaz: mensaje sueldo() sin parámetros. A baigorria no le podemos asignar el sueldo.

```
>>> baigorria.sueldo(29)
ERROR: baigorria does not understand sueldo(29) (line: 1)
```

[1] El ejemplo puede descargarse de https://github.com/wollok/polimorfismo-empanadasGimenez

Vamos a agregar entonces un tercer objeto, un observador para estos... tenemos un empleador, Giménez, que es quién le paga el sueldo a estos empleados. Queremos decirle a Giménez que le pague a Baigorria o a Galván. Cuando paga, disminuye su dinero. Vamos a dejar de lado la recepción del dinero por parte de los empleados, ya que extendería el ejemplo didáctico.

Agregamos entonces el objeto que representa al empleador, con su correspondiente método para pagar a un empleado. Asumimos que su "capital" o dinero inicial es de $ 300.000.

```
object gimenez {
   var dinero = 300000

   method dinero() = dinero
   method pagarA(empleado) { dinero = dinero - empleado.sueldo() }
}
```

¿Quién recibe el mensaje sueldo enviado por gimenez? Y... puede ser baigorria o galvan, viendo sólamente el código no podemos saber.

Por ejemplo, si en la consola ingresamos:

```
>>> gimenez.pagarA(baigorria)
>>> gimenez.dinero()
298485
```

Al evaluar la consulta, el parámetro de pagarA(empleado) es baigorria, y por lo tanto empleado.sueldo() es un mensaje que termina provocando la evaluación del método sueldo() de baigorria. Pero bien podríamos haber hecho la consulta para galvan. El mensaje que manda el observador es siempre *empleado.sueldo( )*, pero el método que finalmente se termina evaluando, es decir, sueldo() del objeto baigorria o sueldo() del objeto gimenez, se determina en tiempo de ejecución.

1.4 Polimorfismo y Tipos

Volvamos al método del objeto gimenez:

```
method pagarA(empleado) { dinero = dinero - empleado.sueldo() }
```

¿de qué tipo es el parámetro empleado? en principio, de cualquier objeto que entienda el mensaje sueldo() sin parámetros y que devuelva un número. El sistema de tipos de Wollok sabe que son por el momento dos los objetos posibles: baigorria y galvan.

Incluso, si generamos un nuevo objeto gutierrez que tenga implementado un método sueldo(), tan pronto grabemos el archivo el sistema de tipos lo detectará y actualizará los posibles tipos para el parámetro empleado en el método pagar:

```
1  object galvan {
2      var sueldo = 15000
3      method sueldo() { return sueldo }
4      method sueldo(nuevoValor) { sueldo = nuevoValor
5  }
6
7  object baigorria {
8      var cantidadEmpanadasVendidas = 0
9      const montoPorEmpanada = 15
10
11     method venderEmpanadas(cuantas) {
12         cantidadEmpanadasVendidas += cuantas
13     }
14
15     method sueldo() =
16         cantidadEmpanadasVendidas * montoPorEmpanada
17 }
18
19 object gimenez {
20     var dinero = 300000
21
22     method dinero() { return dinero }
23
24     method pagarA(empleado) {
25         dinero -= empleado.sueldo()
26     }
```

2 Resumen

En este capítulo hemos conocido uno de los conceptos fundamentales del paradigma: el polimorfismo como una herramienta para agrupar objetos que tienen diferente comportamiento (distinto código) pero la misma interfaz. Entonces puedo enviarles mensajes sin saber cómo lo implementan, e incluso sin saber a qué objeto particular le estoy hablando: puedo cambiar la referencia sin tener que hacer cambios.

Paradigma
Orientado a Objetos

Módulo 04: Objetos básicos.
Números. Strings.
Booleanos. Fechas.
Bloques. Colecciones.

por Fernando Dodino
Versión 3.1
Agosto 2019

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Números
- 1.1 Números con decimales

2 Booleanos

3 Strings

4 Fechas

5 Lambdas: objetos bloque
- 5.1 Introducción, ejemplo básico
- 5.2 Bloques con un parámetro
- 5.3 Bloques con dos parámetros
- 5.4 Contexto de los bloques
- 5.5 Comparación con el paradigma funcional

6 Introducción a colecciones
- 6.1 ¿Qué es una colección?
- 6.2 Representación
- 6.3 Interfaz de una colección
- 6.4 Intro a tipos de colecciones en Wollok

7 Ejemplo integrador: Misiones de un héroe
- 7.1 Shrek, nuestro único héroe en este lío
- 7.2 Primera misión: liberar a Fiona
- 7.3 Cuántas misiones tiene un héroe
- 7.4 Segunda misión: buscar la piedra filosofal
- 7.5 Filtrar misiones difíciles
- 7.6 Filtrar elementos de una colección
- 7.7 Filter y el efecto colateral
- 7.8 Map: Encargados de una misión
- 7.9 Sumar puntos de recompensa

8 Resumen

La distribución de Wollok viene con objetos básicos que nos resultarán muy útiles para poder construir nuestras propias definiciones.

1 Números

Los números en Wollok se representan como objetos inmutables, esto quiere decir que

- un número no cambia su estado interno
- la suma de 1 + 2 resulta en un nuevo número que representa al 3.

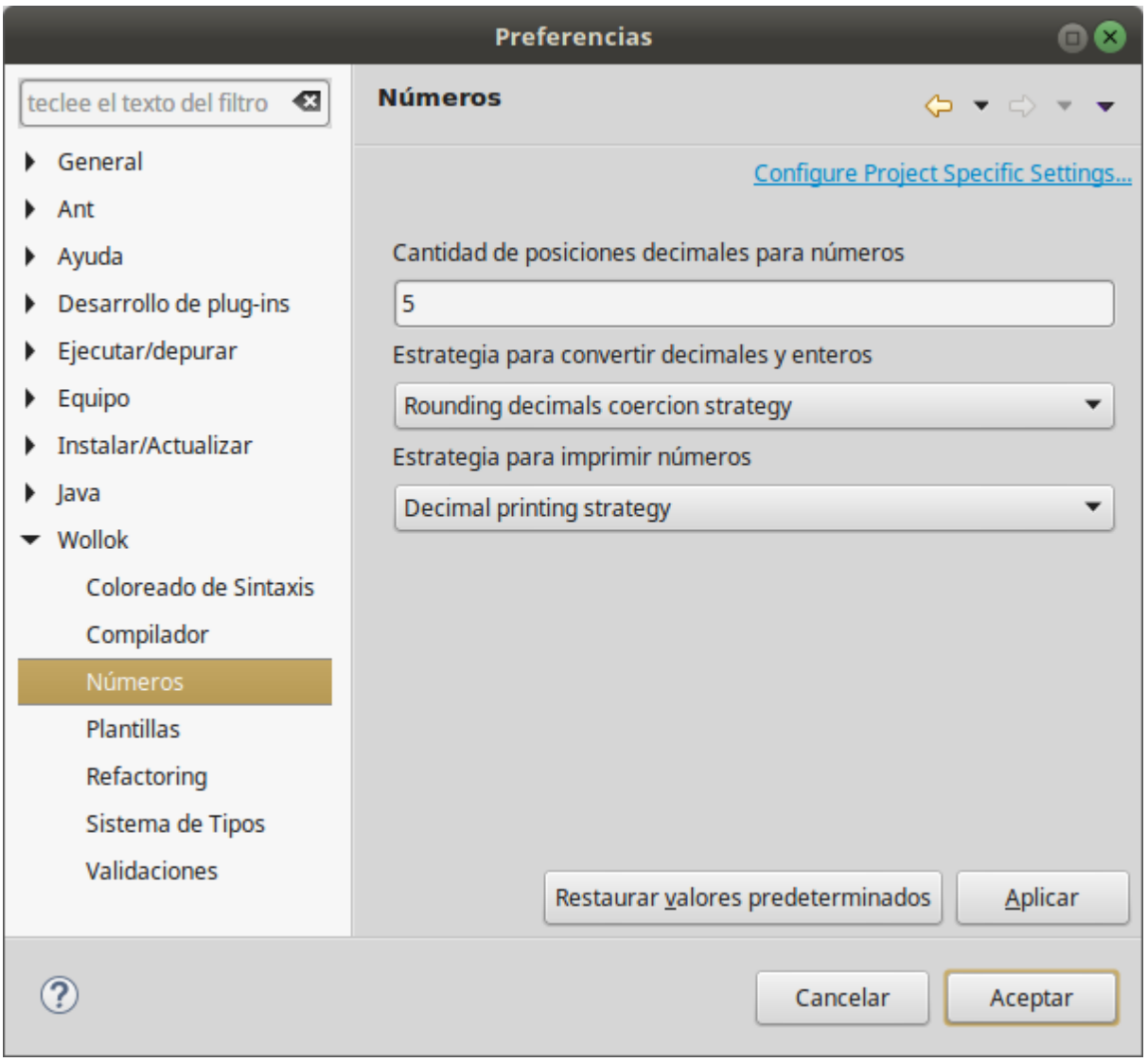
```
const a = 1
var b = a + 10  // suma
b = b - 1       // resta
b = b * 2       // multiplicación
b = b / 2       // división
b = b % 2       // resto
b = b ** 3      // elevado a (3 en este caso)
5.between(2, 7) // preguntamos si 5 está entre 2 y 7 ==> sí
3.min(6)        // el menor número entre 3 y 6 ==> 3
3.max(6)        // el mayor número entre 3 y 6 ==> 6
(3.1416).truncate(0)  // la parte entera de 3.1416 ==> 3 --
(3.1416).truncate(2)  // 3.14
(3.1416).roundUp(0)   // el primer entero mayor a 3.1416 ==> 4  --
(3.1416).roundUp(2)   // 3.15
```

1.1 Números con decimales

Es posible configurar la forma de trabajar con números que tengan decimales. En particular, se puede cambiar

- la cantidad de decimales máxima que puede admitir, por defecto son cinco
- y qué hacer en caso de recibir un número que excede la máxima cantidad de decimales permitido, por defecto lo redondea hacia arriba
- la forma de imprimir un número, por defecto eliminando los ceros no representativos

La configuración de números se activa mediante el menú Ventana > Preferencias, en la solapa Wollok > Números:



Para una configuración por defecto, con cinco decimales como máximo, redondeando para arriba y con la impresión de los decimales representativos, vemos algunas operaciones:

```
>>> 5.2912413
5.29124
>>> 5.2912488
5.29125
>>> 1.000005 - 1.000004
0.00001
>>> 1.000002 - 1.000001
0
```

Si configuramos que en lugar de redondear impida el trabajo con más de 5 decimales, la respuesta por consola será diferente:

```
>>> 5.2912413
```

```
wollok.lang.Exception: El valor 5.2912413 debe tener menos de 5
posiciones decimales
```

2 Booleanos

Hay dos objetos booleanos representados con los literales “true” y “false”. Al igual que los números también son objetos inmutables, la expresión

`(true || false)`

no modifica el objeto original.

```
const hecho = true and true
const esTrue = true
const esFalse = false

const seraFalse = esTrue and esFalse

const seraTrue = esTrue or esFalse

const seraTrue = not false
```

Para aquellos que estén acostumbrados a los operadores con símbolos (en lenguajes como C o java) pueden usar esta sintaxis si se sienten más cómodos:

- and: `a && b`
- or: `a || b`
- not: `!a`

3 Strings

Las cadenas de caracteres se delimitan con una o dos comillas.

```
const unString = "hola"
const otroString = 'mundo'
```

También son objetos inmutables (al concatenar “hola” y “mundo” tenemos un nuevo String “holamundo”). O bien...

```
const holaMundo = unString + " " + otroString + " !"
```

4 Fechas

Una fecha es un objeto inmutable que representa un día, mes y año (sin horas ni minutos). Se crean de dos maneras posibles:

```
>>> const hoy = new Date()
        // toma la fecha del día
>>> const unDiaCualquiera = new Date(day = 24, month = 11, year = 2017)
        // se ingresa en formato día, mes y año
```

Algunas operaciones que podemos hacer con las fechas son:

```
>>> const hoy = new Date()
>>> hoy
24/11/2017
>>> hoy.plusYears(1)     // sumo un año
24/11/2018               // devuelve una nueva fecha
>>> hoy.plusMonths(2)    // sumo 2 meses
24/1/2018
>>> hoy.plusDays(20)
14/12/2017
>>> hoy.isLeapYear()     // pregunto si el año es bisiesto
false
>>> hoy.dayOfWeek()      // qué día de la semana es
friday
>>> hoy.month()
11
>>> hoy.year()
2017
>>> const ayer = hoy.minusDays(1)
                         // resto un día para obtener el día de ayer
>>> ayer < hoy           // comparo fechas
true
>>> ayer - hoy           // comparo fechas
-1                       // diferencia en días entre ayer y hoy
>>> const haceUnMes = hoy.minusMonths(1)
>>> ayer.between(haceUnMes, hoy)
true                     // ayer está entre hace un mes y hoy
```

5 Lambdas: objetos bloque

5.1 Introducción, ejemplo básico

Al igual que en otras tecnologías, Wollok nos permite tener un objeto que representa un bloque de código[1], de manera de

[1] A veces se referencia este tipo de expresiones como *closure* (sobre todo quienes vienen del mundo Java). En términos generales esto no es correcto, dado que son ideas diferentes: una lambda es una función/método que no tiene nombre, mientras una closure o cierre o clausura es un método que puede acceder a las variables disponibles en el contexto en que se la declaró

- poder generar referencias a dichos bloques
- pasarlos como parámetro
- elegir en qué momento ejecutar una porción de código: en un momento instanciamos un bloque de código y tiempo después
    - lo ejecutamos si se cumple una condición,
    - ante un evento
    - o bien lo ejecutamos *n* veces hasta que se cumpla una condición: por ejemplo para reprocesar una instrucción hasta que no haya errores.

Veamos el primer ejemplo básico, un bloque de código que permite conocer el valor absoluto del número 4:

```
>>> const abs4 = { => 4.abs() }
>>> abs4
{ => 4.abs() }

>>> abs4.apply()
4
```

En la primera línea creamos la expresión lambda. En la última línea evaluamos el código que contiene abs4, y devolvemos el valor 4.

5.2 Bloques con un parámetro

Vamos a mejorar el ejemplo anterior, pasando como parámetro un número cuyo valor absoluto queremos conocer:

```
>>> const abs = { numero => numero.abs() }
>>> abs
{ numero => numero.abs() }
>>> abs.apply(-8)
8
```

La operatoria es exactamente igual, sólo que como vemos hay un método apply que permite recibir un parámetro.

5.3 Bloques con dos parámetros

También podemos definir lambdas que reciben 2 parámetros, por ejemplo para seleccionar el mayor número entre dos, o bien para sumarlos:

```
>>> { num1, num2 => num1.max(num2) }.apply(4, 2)
4
```

(esto incluye variables que no son locales dentro del bloque que definimos, tiene un alcance mayor).

```
>>> { num1, num2 => num1 + num2 }.apply(4, 2)
6
```

O bien podemos currificarlo, como hemos visto en el paradigma funcional:

```
>>> { num1 => { num2 => num1 + num2 }}.apply(4)
{ num2 => num1 + num2 }   // devuelve un bloque

>>> { num1 => { num2 => num1 + num2 }}.apply(4).apply(2)
6
```

5.4 Contexto de los bloques

Un dato importante de los closures es que no solo acceden a sus parámetros, sino también a cualquier otra referencia en el contexto donde fueron definidas. Esto las vuelve realmente poderosas. Veamos un ejemplo muy sencillo:

```
var to = "world"
const helloWorld = { "hello " + to }
helloWorld.apply() == "hello world"              // true

to = "someone else"
helloWorld.apply() == "hello someone else"       // true
```

Se podrá ver que el closure accede a la variable “to” que es definida fuera del contexto del closure mismo, dentro del programa. Si cambiamos esta referencia, este efecto se propaga al closure (como se muestra en la segunda llamada, el valor devuelto es diferente).

5.5 Comparación con el paradigma funcional

`( \num1 num2 -> num1 + num2 )` Haskell	`{ num1, num2 => num1 + num2 }` Wollok

Más allá de la diferencia en la sintaxis, podemos hacer una comparación entre los bloques de código en Objetos y las expresiones lambdas de Haskell:

- ambas sirven para **abstraer comportamiento** que no interesa reutilizar en otro contexto, por eso no tienen nombre
- las expresiones lambda de Haskell solo devuelven un valor, mientras que en objetos podemos retornar un valor, o tener **efecto colateral** (en ese caso puede no importar el valor resultante)
- ambas son particularmente útiles para evitar hacer tareas repetitivas y subir el grado de **declaratividad**. Tomando los ejemplos de soluciones con map, filter y fold, tienen un menor grado de conocimiento del algoritmo y del orden

que la misma solución con recursividad (en el caso de funcional) o un forEach (en el caso de objetos).

6 Introducción a colecciones

6.1 ¿Qué es una colección?

La colección nos permite representar un conjunto de objetos relacionados:
los jugadores de un equipo de fútbol, un cardumen, una lista de cosas a comprar en el supermercado, las cosas que un héroe guarda en su mochila, un ejército, son ejemplos de este tipo de abstracciones.

Otra definición posible es que una colección nos sirve para modelar una relación 1 a N:

- Una factura tiene muchas líneas con productos
- Un escritor publicó varios libros
- Una fiesta tiene muchos invitados
- Un héroe tiene que cumplir varias misiones

A primera vista una colección es un conjunto de objetos. Si la vemos con más detalle nos damos cuenta que es más preciso pensarla como un conjunto de referencias: los elementos no están dentro de la colección, sino que la colección los conoce.

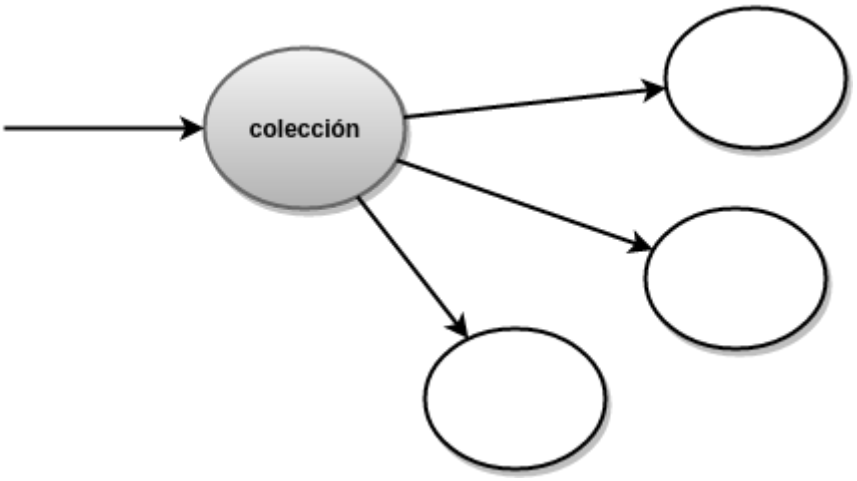


Figura 3.1: La colección es un conjunto de referencias a otros objetos

6.2 Representación

Podemos graficar la relación dinámica entre un equipo de fútbol y los jugadores que lo integran mediante un diagrama de objetos. Este es un diagrama con características *dinámicas*, porque muestra el estado de los objetos en un momento determinado.

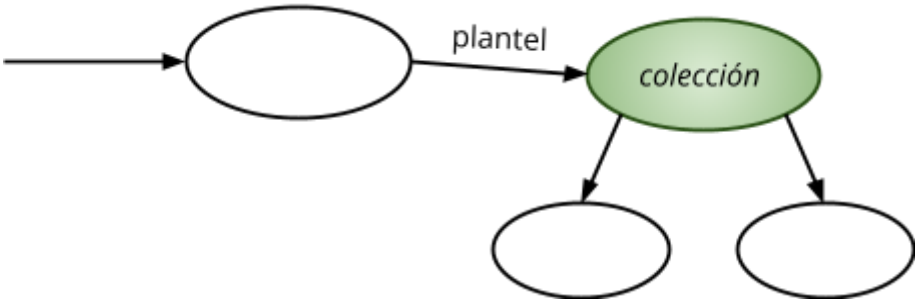


Figura 3.2: El plantel de jugadores de un equipo como una colección de objetos

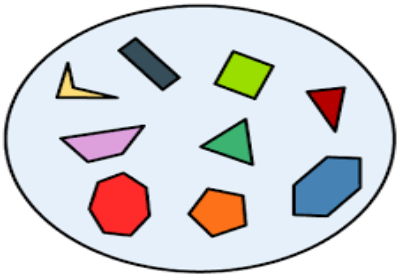
6.3 Interfaz de una colección

Supongamos que tenemos un álbum de fotos, otra representación posible de una colección de objetos. ¿Qué podemos hacer con esas fotografías?

- Mirarlas, "recorrerlas": iterar una colección
- Averiguar cuántas fotos hay: saber el tamaño de una colección
- Saber si está una determinada foto en el álbum, es decir, si un elemento pertenece a la colección
- Pegar una foto nueva: agregar un elemento a la colección
- Regalar una foto a alguien: eliminar un elemento de la colección
- Seleccionar las fotos del viaje de 1999 a Ushuaia: filtrar elementos de una colección
- Saber las fechas en la que saqué mis fotos: transformar los elementos de una colección
- Saber si hay alguna foto de Navidad: determinar si algún elemento satisface un criterio

6.4 Intro a tipos de colecciones en Wollok

A primera vista, podemos diferenciar dos diferentes tipos de colecciones en Wollok



los **conjuntos**, que modelan al conjunto matemático: no hay orden en los elementos y no puede haber elementos repetidos. Se definen mediante el literal #{ }

```
const numeros = #{1, 2, 3, 4}
```

las **listas**, en donde los elementos tienen un orden y puede haber elementos repetidos. Se definen mediante el literal [ ]

```
const fila = [francisco, mirta, hector]
```

7 Ejemplo integrador: Misiones de un héroe

En un sistema que administra diferentes tipos de héroe, a un héroe se le puede asignar misiones y nuevos objetivos. Por ejemplo, un mago puede encargarle buscar un ítem mágico en una montaña lejana. Un anciano puede encargarle liberar a su hija de los terribles trolls que habitan en la gruta de los sin nariz[2].

Cada vez que un héroe tiene uno de estos encuentros, anota los datos de la misión en su diario personal. Toda misión suma en el camino del héroe: las misiones tienen una recompensa de respeto, que luego puede canjearse por oro.

¿Qué abstracciones surgen? El héroe tiene una colección de misiones, algunas de las cuales puede ser

1. buscar un ítem mágico, que es difícil si el ítem a buscar queda a más de 100 kms. (la localización del ítem puede ir variando)
2. liberar a fiona, custodiada por una cantidad variable de trolls (si son 4 ó 5 la misión es difícil).

7.1 Shrek, nuestro único héroe en este lío

Definiremos un objeto shrek, un gran (anti)héroe. Por el momento sabemos que tiene misiones, la primera pregunta que nos tenemos que hacer es: ¿necesitamos mantener el orden de cada una de las misiones? ¿podemos asignarle dos veces la misma misión? A primera vista, el orden no parece importante y vamos a suponer que, de la misma manera que uno no puede bañarse dos veces en el mismo río[3], uno no libera a la misma doncella dos veces.

```
object shrek {
      const misiones = #{}
}
```

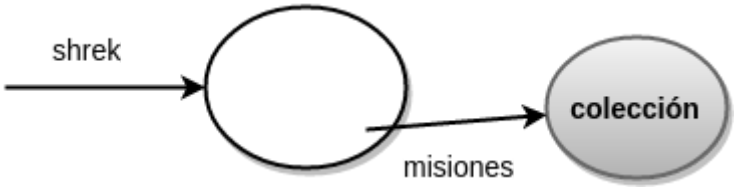


Fig.3.3: Misiones es una referencia a un *conjunto*

[2] El ejemplo completo puede descargarse de https://github.com/wollok/heroes-con-objetos

[3] En realidad el dicho de Heráclito es *ποταμοῖς τοῖς αὐτοῖς ἐμβαίνομεν τε καὶ οὐκ ἐμβαίνομεν, εἶμεν τε καὶ οὐκ εἶμεν τε.*
En los mismos ríos entramos y no entramos, [pues] somos y no somos [los mismos].
Esto implica que tanto el río como el que se baña en él cambian todo el tiempo.

La referencia es **constante**, porque agregaremos o eliminaremos elementos pero **siempre apuntando a la misma colección**.

7.2 Primera misión: liberar a Fiona

Shrek tiene su primera misión asignada por Lord Farquaad: liberar a una doncella encerrada en la torre más alta de un castillo protegido por un ejército de 5 trolls[4]. Creamos un objeto que representa la misión de liberar a Fiona

```
object liberarAFiona {

}
```

El lector observará que no definimos todavía comportamiento ni atributos: no queremos adelantarnos ni tomar decisiones antes de tiempo. Por el momento nuestro objetivo es incorporar la misión al héroe y para eso necesitamos mandarle un mensaje a shrek:

```
>>> shrek.agregarMision(liberarAFiona)
```

Entonces en shrek, debemos escribir el método agregarMision(), que delega la acción al objeto que está representando la colección:

```
object shrek {
      const misiones = #{}
      method agregarMision(_mision) { misiones.add(_mision) }
}
```

Cuando enviemos el mensaje a shrek

```
>>> shrek.agregarMision(liberarAFiona)
```

en nuestro ambiente tendremos estas referencias:

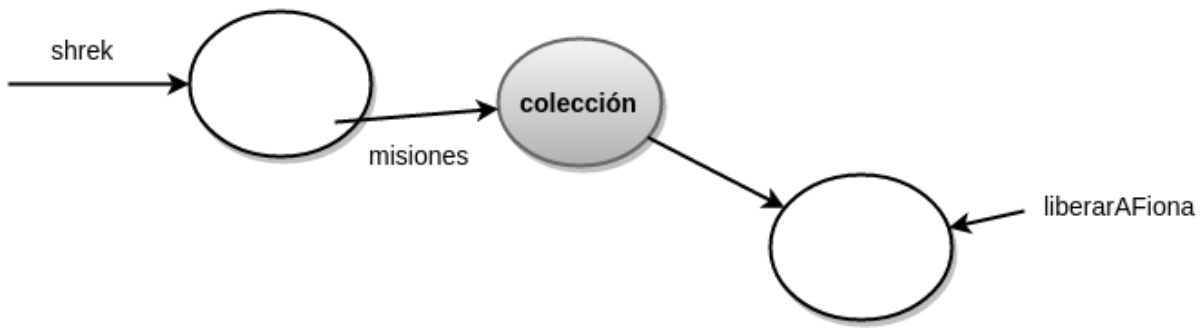


Fig 3.4: Primera misión para shrek

[4] Luego de unos años, la Dragona decidió tercerizar la seguridad.

7.3 Cuántas misiones tiene un héroe

¿Cómo sabemos cuántas misiones tiene un héroe? Le preguntamos a shrek
`shrek.cantidadDeMisiones()`

Son dos momentos diferenciados:

1) primero la interfaz,
2) luego cómo implementarlo

```
object shrek {
      ...
      method cantidadDeMisiones() = misiones.size()[5]
}
```

7.4 Segunda misión: buscar la piedra filosofal

Shrek se cruza inesperadamente con el director de una famosa escuela de magia, quien le pide que en secreto busque una famosa piedra filosofal que dejó olvidado una noche de borrachera en los montes Urales, a 40 kms. del pantano donde vive nuestro héroe.

Creamos entonces nuestro objeto buscarPiedraFilosofal:

```
object buscarPiedraFilosofal { }
```

Y ejecutamos nuevamente la consola interactiva, agregando ambas misiones:

```
>>> shrek.agregarMision(liberarAFiona)
>>> shrek.agregarMision(buscarPiedraFilosofal)
```

Podemos volver a preguntar a shrek la cantidad de misiones, no esperamos sorprender al lector con la respuesta.

7.5 Filtrar misiones difíciles

Nuestro nuevo requerimiento necesita aplicar varios conceptos, vamos con el primero: las misiones “liberar a Fiona” y “buscar la piedra filosofal” tienen que entender el mismo mensaje.

```
>>> liberarAFiona.esDificil()
>>> buscarPiedraFilosofal.esDificil()
```

[5] Otra forma de definir el método es con la definición del cuerpo del método y el return explícito
`method cantidadDeMisiones() { return misiones.size() }`
Wollok permite esta definición simplificada del método cuando abarca una sola línea y devuelve un valor. En caso de duda recomendamos seguir con la sintaxis original.

Entonces, necesitamos que sean **polimórficos**. ¿Para quién es esto? No para las misiones, sino para shrek, que es quien posteriormente los va a usar. Pero no nos apuremos, ahora sí tenemos que definir el comportamiento en las misiones

```
object liberarAFiona {
      var cantidadTrolls = 5
      method esDificil() = cantidadTrolls.between(4, 5)
}

object buscarPiedraFilosofal {
      var kilometrosDistancia = 40
      method esDificil() = kilometrosDistancia > 100
}
```

Fíjense que no necesitamos -por el momento- definir *accessors* (*getters* / *setters*) para los atributos de los objetos liberarAFiona y buscarPiedraFilosofal.

7.6 Filtrar elementos de una colección

¿Cómo pregunto qué misiones difíciles tiene Shrek?

```
>>> shrek.misionesDificiles()
#{liberarAFiona}
```

Las colecciones definen un mensaje filter, al que le podemos pasar el criterio que deben cumplir los elementos que nos interesan:

```
object shrek {
      ...
      method misionesDificiles() =
            misiones.filter({ mision => mision.esDificil() })
}
```

Filter recibe un objeto bloque que espera un parámetro que representa una misión (por el momento puede ser liberarAFiona o buscarPiedraFilosofal), y devolverá un booleano indicando si la misión es difícil.

Para resolver las misiones difíciles

- shrek conoce a sus misiones
- pero delega la respuesta a cada misión, que entiende el mensaje `esDificil()`. No importa cómo lo implementan, lo importante es que todos devuelven un valor booleano que permite discriminar misiones fáciles de difíciles.

7.7 Filter y el efecto colateral

El resultado de enviar el mensaje misionesDifíciles() no altera las referencias de la colección original:

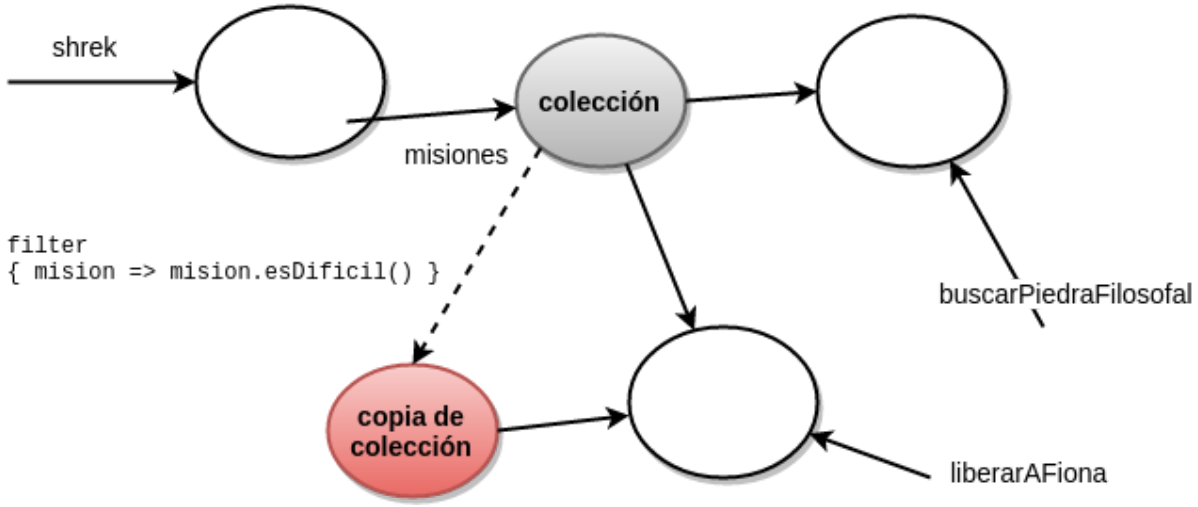


Figura 3.5: Filter devuelve una copia de la colección con los elementos que cumplen una condición

Esto permite hacer consultas sin que haya efecto sobre las referencias originales, no sería lógico que al querer conocer los clientes que nos deben plata perdamos los que sí pagaron[6].

7.8 Map: Encargados de una misión

Cada misión tiene un solicitante, en particular

- liberar a Fiona fue solicitado por Lord Farquaad
- mientras que buscar la piedra filosofal fue solicitado por Mr. Dumblecofcofdore

Una forma de modelar esto es mediante una referencia o variable, otra por el momento es tener un método que retorne una constante. No nos vamos a preocupar mucho más por el momento, más adelante tendremos la oportunidad de mejorar esta idea:

```
object liberarAFiona {
      var cantidadTrolls = 4
      method solicitante() = "Lord Farquaad"
      ...
```

[6] De la misma manera funciona el filter que hemos visto en Haskell, facilitado por la ausencia de asignaciones destructivas

```
object buscarPiedraFilosofal {
      var kilometrosDistancia = 40
      method solicitante() = "Mr DumblecofcofDore"
      ...
```

¿Cómo le pedimos a shrek los solicitantes de todas sus misiones?

```
>>> shrek.agregarMision(liberarAFiona)
>>> shrek.agregarMision(buscarPiedraFilosofal)
>>> shrek.solicitantesDeMisMisiones()
#{"Lord Farquaad", "Mr Dumblecofcofdore"}
```

Ahora podemos pensar en el método solicitantesDeMisMisiones() que tiene que aplicar una transformación a cada elemento de la colección:

INPUT: una misión	OUTPUT: su solicitante (un String)
cómo lo logramos: `mision.solicitante()`	

Nos ayudamos con el mensaje map() a la colección:

```
object shrek {
      ...
      method solicitantesDeMisMisiones() =
            misiones.map({ mision => mision.solicitante() })
}
```

Al igual que filter, map no tiene efecto colateral:

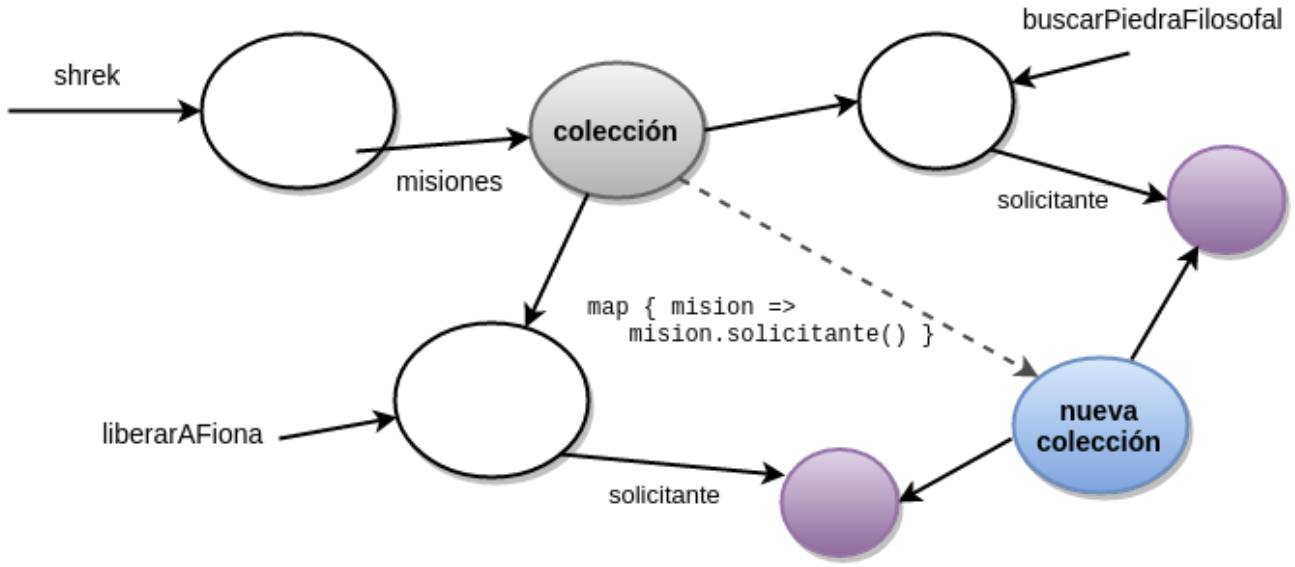


Figura 3.5: Map devuelve una colección nueva con los elementos transformados

Esto tiene sentido, porque además pasamos de un conjunto de misiones a un conjunto de solicitantes.

7.9 Sumar puntos de recompensa

Cada misión tiene una cantidad de puntos de recompensa, un valor numérico. Los puntos de recompensa por liberar a fiona son 2 * cantidad de trolls que la custodian. Respecto a buscar el ítem mágico, son 10 puntos si la distancia a recorrer es más de 50 kilómetros ó 5 en caso contrario.

```
object liberarAFiona {
      ...
      method puntosRecompensa() = cantidadTrolls * 2
}

object buscarPiedraFilosofal {
      ...
      method puntosRecompensa() =
            if (kilometrosDistancia > 50) 10 else 5
}
```

Nuevamente, ¡podemos enviar mensajes a liberarAFiona o buscarPiedraFilosofal en forma indistinta! Encontramos otro contexto en el cual ambos objetos son **polimórficos**. Y esto nos permite resolver nuestro tercer requerimiento: queremos saber los puntos de recompensa de las misiones que tiene encargado un héroe, esto se logra... enviando el siguiente mensaje

```
>>> shrek.totalPuntosDeRecompensa()
15
```

claro está.

Lo resolvemos aprovechando otra construcción conocida por nosotros: sum, que necesita el criterio para sumar números obtenidos en base a elementos de una colección.

```
object shrek {
      ...
      method totalPuntosDeRecompensa() =
            misiones.sum({ mision => mision.puntosRecompensa() })
}
```

Y aquí vemos claramente que shrek *aprovecha el polimorfismo* para sumar los puntos de recompensa de las misiones, sin importarle cómo lo termina implementando cada una.

8 Resumen

En este capítulo presentamos a los objetos básicos de Wollok: los números, los strings, los booleanos, las fechas, los bloques y finalmente las colecciones como una forma de tener una referencia a múltiples objetos, y comenzamos a conocer su rica interfaz que nos permite resolver requerimientos sin necesidad de tener que bajar demasiado a detalle el algoritmo.

Paradigma Orientado a Objetos

Módulo 05: Introducción al Testeo unitario automatizado

por Lucas Spigariol
Fernando Dodino
y colaboración de
Matías Freyre
Versión 2.0
Agosto 2017

Distribuido bajo licencia Creative Commons Share-a-like

Indice

- Introducción
- Casos de prueba
  - Ejemplo
  - Solución
- Pruebas manuales
  - Consola (REPL)
  - Programas
  - Wollok Game
  - Problemas
- Testeo unitario automatizado
  - Unitario
  - Automático
  - Independencia
  - Información sintética y precisa
- Definición de tests
  - Describe
  - Test
- Ejecución de los test
  - Éxito
  - Fallas
  - Errores
  - Resultados del test
- Resumen
- Apéndice: interfaz de assert

1. Introducción

Cuando hacemos las cosas, ¿de qué manera nos damos cuenta de si las estamos haciendo bien? En general, las probamos. Realizar pruebas o evaluaciones nos permite saber si lo que hicimos cumple con el objetivo. Lo mismo nos pasa cuando construimos software: es importante validar que funcione correctamente.

Si bien hay múltiples aspectos de un determinado software que se pueden analizar, como su eficiencia, su expresividad, su facilidad para realizar modificaciones, etc. y cada una de ellas tiene sentido desde diferentes puntos de vista, en esta oportunidad nos enfocamos en pruebas para determinar específicamente su correctitud, es decir, que el software esté haciendo realmente lo que debe hacer, y no otra cosa. En otras palabras, buscamos quedarnos tranquilos que los resultados que obtenemos al ejecutar un programa son los esperados.

Existen diversas formas de ejecutar código *Wollok* que nos permiten ver qué sucede y de esta manera asegurarnos que haga lo que debe hacer: está la consola interactiva (REPL), se cuenta con la interfaz gráfica que provee *Wollok Game* y se pueden definir programas. Todas ellas tienen en común que son relativamente sencillas y se utilizan de una manera manual, que puede ser útil en determinados contextos, pero que a medida que las soluciones se complejizan no resultan convenientes.

Frente a esta situación, de manera similar a otros lenguajes de programación de la actualidad, *Wollok* ofrece una herramienta para probar el código de una manera mucho más práctica, confiable y profesional, mediante lo que se denominan **tests unitarios**, que tienen la característica fundamental de ser **automatizados**.

2. Casos de prueba

Independientemente del método elegido, lo primero a considerar es qué queremos probar. Incluso antes de tener el código escrito, conociendo los detalles del problema a resolver ya podemos identificar un abanico de situaciones posibles, que vaya desde los casos típicos y frecuentes hasta las situaciones límite o de escasa probabilidad pero que de todas maneras también deben ser contemplados.

Lo importante es poder anticipar, para cada posible situación, cuáles son los resultados correctos que deberá brindar el código, para poder hacer efectivamente una validación. Y aquí, la palabra clave es "anticipar": un problema que suele suceder en la realidad, -cuando se trabaja con poco profesionalismo-, es probar el funcionamiento de una solución sólo viendo que el código ejecuta sin dar error, que arroja algún resultado e inocentemente confiar que es el correcto, sin tener contra qué contrastarlo.

De esta manera, cualquier estrategia de testeo se basa en construir casos de prueba donde se describen las diferentes situaciones de posible ocurrencia y se anticipa cuáles deben ser los resultados correctos a obtener en cada una de ellas. En particular, en el paradigma de la programación orientada a objetos… ¿Qué queremos probar de un objeto? Principalmente, su comportamiento: queremos enviarle mensajes al objeto y verificar que suceda lo esperado.

2.1. Ejemplo

Tomemos como ejemplo a *pepita*. Para determinar los casos de prueba veamos cómo está planteado el problema:

> *"Sabemos que pepita tiene inicialmente 100 unidades de energía. Cuando come alpiste, su energía se incrementa en 4 veces la cantidad de gramos de alimento ingerido. Al volar, su energía disminuye la cantidad de kilómetros que vuela más 10 unidades fijas. Queremos averiguar si pepita es fuerte, lo cual sucede cuando su energía supera las 50 unidades, y también nos interesa saber en cualquier momento cuál es su energía actual".*

Empecemos por lo que queremos averiguar de pepita

- En la situación inicial de pepita, es decir sin que haya volado ni comido, si le preguntamos la energía, la respuesta debe ser 100.
- En las mismas condiciones, si le preguntamos si es fuerte, la respuesta debería ser afirmativa. (100 > 50)

Con esto, ya tenemos dos casos de prueba.

Sigamos con los otros requerimientos, que no son consultas que tienen una respuesta como resultado, sino que son indicaciones que deben provocar un cambio en el objeto.

- Al decirle a pepita que vuele 5 kms, asumiendo que estaba en su estado inicial, su energía debe quedar en 85. (100 - (5 + 10) = 85)
- Si pepita, en su estado inicial, recibe la orden de comer 120 gramos de alpiste, su energía pasa a ser 580. (100 + (120 * 4) = 580)

Y tenemos de esta manera dos nuevos casos de prueba.

Podríamos querer probar qué sucede con otros valores, por ejemplo, si pepita volase otra cantidad de kilómetros o comiese otra cantidad de alpiste. Sería no sólo tedioso (e interminable) plantear casos de prueba con todos los valores posibles sino fundamentalmente irrelevante, pero es recomendable prestar atención a ciertos valores

que representen situaciones particulares. Por ejemplo, sería interesante pedirle a pepita que vuele 0 km (que podría significar que arranca a volar y se arrepiente instantáneamente) o que coma 0 gramos (que, aunque parezca trivial, también es una orden válida) que son casos singulares que ayudan a garantizar que el objeto funcione correctamente en todos los casos.

Por lo tanto, obtenemos dos casos de prueba más:

- Al decirle a pepita que vuele 0 kms, asumiendo que estaba en su estado inicial, su energía debe pasar a ser 90. (100 - (0 + 10) = 90)
- Si pepita, en su estado inicial, recibe la orden de comer 0 gramos de alpiste, su energía sigue siendo la inicial de 100. (100 + (0 * 4) = 100)

Otra situación interesante a probar es que la pregunta por si pepita es fuerte tenga respuestas negativas cuando realmente corresponda. Para este caso, no basta con la situación inicial porque ya vimos que allí la respuesta debe ser afirmativa; lo que tenemos que hacer es preguntarle si es fuerte habiendo previamente volado una suficiente cantidad de kilómetros. Obtenemos así un nuevo caso de prueba.

- Partiendo de la situación inicial, si hacemos que pepita vuele 60 kilómetros de manera que su energía llegue a 30 (100 - (10 + 60)), ante la pregunta si es fuerte la respuesta debe ser negativa (30 < 50).

Por último, es también útil probar los casos límite. Ya vimos qué sucede con los niveles de energía menores y mayores al límite. Y si es exactamente el límite, ¿qué debería pasar? De acuerdo con la definición, es fuerte cuando SUPERA las 50 unidades. Entonces, nuestro último caso sería.

- Partiendo de la situación inicial, si hacemos que pepita vuele 40 kilómetros de manera que su energía llegue a 50 (100 - (10 + 40)), ante la pregunta si es fuerte la respuesta debe ser nuevamente negativa (porque no se cumple 50 > 50).

Esquemáticamente, los casos de prueba planteados son:

Acciones	Verificación
ninguna	pepita queda con energía 100
ninguna	pepita es fuerte
que pepita vuele 5 kilómetros	pepita queda con energía 85
que pepita coma 120 gramos	pepita queda con energía 580
que pepita coma 0 gramos	pepita queda con energía 100

que pepita vuele 0 kilómetros	pepita queda con energía 90
que pepita vuele 60 kilómetros	pepita no es fuerte
que pepita vuele 40 kilómetros	pepita no es fuerte

2.2. Solución

Para poder continuar con la explicación, veamos el código de una posible solución al problema planteado:

```
object pepita {
	var energia = 100

	method energia() { return energia }

	method esFuerte() { return energia > 50 }

	method volar(kms) {
		energia = energia - (kms + 10)
	}

	method comer(gramos) {
		energia = energia + 4 * gramos
	}
}
```

Como se puede observar, tenemos definido al objeto pepita, con una variable que representa la energía y cuatro métodos, uno para cada requerimiento. Entre los métodos, dos de ellos tienen valores de retorno y se corresponden con los mensajes que representan preguntas, consultas o interrogantes a pepita, mientras que los otros dos tienen efecto, es decir provocan un cambio de estado en pepita, y se corresponden con los mensajes que son pedidos, órdenes o indicaciones para que pepita realice.

3. Pruebas manuales

Ahora que tenemos una solución, una alternativa es probarla con las herramientas que ya conocemos. Vamos haciendo cada una de las pruebas y mientras lo que observamos coincida con lo que esperamos, seguimos avanzando sin problemas, pero cuando obtenemos un resultado diferente al esperado, que es cuando la prueba que

hicimos tiene sentido, interrumpimos las pruebas para arreglar el problema. Luego volvemos a probar.

3.1. Consola (REPL)

Abrimos una consola y nos ponemos a interactuar con pepita enviándole mensajes, para ver allí mismo ver su resultado.

Para el primer caso de prueba, hacemos

```
>>> pepita.energia()
100
```

Si nos fijamos en los casos de prueba que teníamos planteados, podemos comprobar que efectivamente era el resultado esperado. Luego queremos saber si es fuerte, y lo hacemos de manera similar.

Cuando llega el momento de probar cómo vuela, hacemos

```
>>> pepita.volar(5)
>>>
```

Como se trata de un mensaje que es una orden, que no tiene retorno, no vemos nada. Entonces, inmediatamente usamos el otro mensaje, haciendo

```
>>> pepita.energia()
85
```

Tenemos que manualmente verificar que sea el valor que habíamos predicho.

Luego, queremos ver si funciona bien el comer, pero para ello sería un error enviar inmediatamente el mensaje. Como la consola conserva el estado del ambiente con todos los objetos, el efecto de los mensajes es acumulativo. Si queremos evaluar el caso de prueba planteado anteriormente, debemos volver a ejecutar la consola para que vuelva todo al estado inicial y recién allí enviar el mensaje comer, con el argumento indicado y ver qué sucede.

Así, podemos ir probando cada caso, teniendo la precaución de plantear cada vez la secuencia correcta y reiniciando la consola oportunamente.

3.2. Programas

Wollok permite crear programas (en archivos con extensión *wpgm*) en los que se especifica una secuencia de código que se ejecuta completa sin necesidad de interacción. En el programa se escribe la serie de mensajes que se envían a los objetos, y teniendo la precaución de mostrar por consola los resultados a medida que son enviados, podemos observar su funcionamiento.

Por ejemplo, podríamos escribir

```
import pepita.*

program pepitaFuerte {
	console.println("En la situación inicial")
	console.println("¿Pepita es fuerte?")
	console.println(pepita.esFuerte())
	pepita.volar(60)
	console.println("Luego de volar 60 kilómetros")
	console.println("¿Pepita es fuerte?")
	console.println(pepita.esFuerte())
}
```

Al ejecutarse, el resultado que vemos por consola sería

```
En la situación inicial
¿Pepita es fuerte?
true
Luego de volar 60 kilómetros
¿Pepita es fuerte?
false
```

Contrastando manualmente con los valores que anticipamos en los casos de prueba correspondiente, podemos verificar que esta parte de nuestra solución funciona correctamente.

Podríamos agregar más líneas de código a este mismo programa para contemplar los casos de prueba restantes o hacer nuevos programas, y teniendo la precaución de mostrar en cada caso los resultados de una manera entendible por la consola e interpretarlos adecuadamente, logramos el objetivo de analizar todos los casos de prueba.

3.3. Wollok Game

Una variante es utilizar *Wollok Game*, que nos permite que los objetos sean representados gráficamente en una interfaz de usuario. Ya sea que configuremos eventos en dicha interfaz para que se envíen mensajes a los objetos ante su ocurrencia, que los escribamos uno por uno en la consola o que los hayamos predefinido en una secuencia mediante un programa, a medida que el objeto va ejecutando los métodos correspondientes, se modifica su representación gráfica y vemos su efecto en la pantalla. Por ejemplo -modificando adecuadamente la solución planteada- podemos tener dos imágenes diferentes, una para representar a pepita fuerte y otra para pepita no fuerte, mostrar en un recuadro visible en todo momento la energía de pepita, hacer que el volar implique también un desplazamiento en la pantalla, o lo que se nos ocurra para poder comprobar visualmente lo esperado en cada uno de los casos de prueba[1].

3.4. Problemas

Con cualquiera de estas estrategias las pruebas son validadas **manualmente.** Si bien pueden resultar prácticas en ciertas ocasiones o problemas sencillos, traen aparejados una serie de problemas que aumentan exponencialmente a medida que la cantidad de código aumenta:

- Es tedioso tener que recordar cada vez que se quiere probar algo cuál es el conjunto de casos de prueba a considerar. Resulta reiterativo escribir nuevamente la misma serie de consultas.
- La verificación manual de cada uno de los valores esperados frente a cada uno de los mensajes, no solo resulta poco práctico sino que es propenso a equivocaciones y se corre el riesgo de aceptar como válido cualquier valor que no sea exactamente el esperado.
- Puede demandar un esfuerzo considerable el envío de mensajes para llevar a un objeto al estado deseado para una determinada prueba.
- En la consola, si luego de enviar numerosos mensajes se detecta un error y se realizan cambios en el código para corregirlo, suele ser dificultoso reproducir la misma secuencia para verificar si el cambio fue adecuado.
- En un programa, cuando se envía una sucesión de mensajes y el resultado final no es el esperado, es dificultoso detectar en cuál de todos los mensajes se produjo el problema.

[1] Para más detalles pueden descargar el ejemplo https://github.com/wollok/pepitaGame

- Al realizar cambios en el código y por lo tanto modificar los casos de prueba, el tiempo que hay que invertir cada vez en probar es similar al que se dedicó originalmente.
- Frente a un cambio o corrección en el código, si solo probamos lo que se modificó, no siempre se puede estar seguro que no se haya afectado a otras partes del sistema, por lo que debería volver a hacerse las pruebas anteriores, con todo lo que ello implica.

4. Testeo unitario automatizado

Teniendo un panorama más completo de las limitaciones, riesgos y dificultades que tienen estas formas manuales de probar la correctitud de una solución, cobra sentido la implementación de otra forma de realizar pruebas que no solo simplifique y sistematice la tarea, sino que garantice su confiabilidad.

Para ello, *Wollok* cuenta con una herramienta para definir **test unitarios automatizados**, que se inscribe en la tendencia actual de los lenguajes de programación de uso profesional de utilizar el concepto **Unit Test** como parte del proceso de desarrollo de software.

Un test, esencialmente, es una porción de código *Wollok* en la que se describe una determinada situación, mediante el envío de mensajes a los objetos correspondientes, y se especifica cuál es el resultado esperado.

4.1. Unitario

El tipo de testeo que abordamos se califica como "unitario", porque la estrategia, en vez de hacer una gran y extensa prueba de un programa completo, se basa en identificar unidades significativas del código y probar casos puntuales donde éstas intervengan. Vamos a definir tests para los casos de prueba, explicitando mediante código la vinculación entre las situaciones planteadas y los resultados esperados.

Si mediante un mismo test probamos muchas cosas a la vez, cuando el test falla es más difícil saber cuál de los múltiples motivos falló. En cambio, si en un test probamos una sola cosa, al encontrarnos con que falla, tenemos una mayor certeza sobre cuál es el problema a corregir. Por este motivo, es fundamental el carácter unitario del test.

Los mismos principios de modularización y delegación de responsabilidades que caracteriza al paradigma de objetos, ayudan a que sea relativamente sencillo identificar las unidades sobre las que se van a definir los test.

Por otra parte, la forma en que anteriormente planteamos los casos de prueba ya tuvo en cuenta implícitamente los criterios de unicidad, por lo que -entendidos de esta manera- a cada caso de prueba le va a corresponder un test específico.

4.2. Automático

La característica fundamental de los tests es que se puede automatizar su ejecución y consecuentemente su validación.

Como en el código de cada test, junto a la descripción de la situación a probar se especifica el resultado esperado, la validación es realizada por la misma máquina, sin mediar la interpretación de la persona, logrando mayor velocidad y confiabilidad.

A su vez, tratándose de código escrito, los tests quedan guardados al igual que el código de la solución propiamente dicha y cuando se desea, se pueden ejecutar todos de una vez, obteniendo un informe del resultado de cada uno de ellos. De esta manera, cuando en el proceso de desarrollo, al correr los tests se detecta algún problema y se corrige la solución, es sencillo volver a ejecutar todas las pruebas anteriores. También, si una solución que ya se probó que funciona correctamente se quiere refactorizar, basta con correr nuevamente todos los tests para garantizar que sigue funcionando adecuadamente.

Ante el aumento en la cantidad y variedad de tests, éstos se pueden agrupar y organizar de diferentes maneras, para permitir variantes en su ejecución.

4.3. Independencia

Una característica que se desprende de su carácter unitario y que posibilita la automatización de los tests sin generar conflictos entre ellos, es que cada uno se concibe en forma independiente de cualquier otro test.

La lógica de ejecución de tests parte del supuesto que cada uno se corre a partir de la situación inicial del sistema, es decir que el ambiente se reinicia entre test y test, garantizando su total independencia.

Además, ante la detección de un problema por parte de algún test, los demás pueden seguir ejecutándose sin inconvenientes. El informe final detalla cuáles tests ejecutaron sin inconvenientes y cuáles detectaron algún problema.

4.4. Información sintética y precisa

Teniendo en cuenta que la importancia de una prueba está dada por su capacidad para detectar problemas, la información más valiosa que aporta no es cuando funciona,

sino cuando nos permite darnos cuenta que el resultado obtenido no es el esperado. En este caso, es importante que nos oriente con precisión acerca de dónde se produjo el problema y nos pueda brindar la mayor información posible.

Ya sea que se decida ejecutar uno, muchos o todos los tests, cuando el test corre bien, la información que arroja es bien sintética, simplemente dice que funcionó: no hay nada más que agregar que nos distraiga. En cambio, cuando un test falla, nos advierte con mayor contundencia y detalla cuál era el resultado esperado y el que realmente encontró.

5. Definición de tests

Los tests se definen en el mismo proyecto en que está la solución propiamente dicha, pero en archivos aparte. En el IDE de Wollok, el menú es Archivo > Nuevo > Test Wollok (o Descripción de tests de Wollok). Ponerle un nombre representativo (con extensión *.wtest*, recomendamos que no tenga el mismo nombre que tu archivo *.wlk*) y aceptar.

El elemento clave es **assert,** un objeto autodefinido de la biblioteca *lib*, que entiende los mensajes que se utilizan en el siguiente ejemplo (y otros más que oportunamente presentaremos) y que permite una forma de plantear las expectativas declarativamente, delegando internamente al framework de testing que provee Wollok toda la lógica de la ejecución y validación automatizada.

Retomando el ejemplo de pepita, los tests quedarían de esta manera:

```
import pepita.*

describe "Tests de Pepita" {

	test "energía inicial de pepita" {
		assert.equals(100, pepita.energia())
	}

	test "pepita comienza siendo fuerte" {
		assert.that(pepita.esFuerte())
	}

	test "pepita vuela y baja su energía" {
		pepita.volar(5)
		assert.equals(85, pepita.energia())
	}

	test "pepita come y sube su energía" {
```

```
        pepita.comer(120)
        assert.equals(580, pepita.energia())
    }

    test "pepita empieza a volar, no vuela y la energía baja" {
        pepita.volar(0)
        assert.equals(90, pepita.energia())
    }

    test "pepita come 0 gramos y la energía se mantiene" {
        pepita.comer(0)
        assert.equals(100, pepita.energia())
    }

    test "pepita vuela muchos kilómetros, y ya no es fuerte" {
        pepita.volar(60)
        assert.notThat(pepita.esFuerte())
    }

    test "pepita vuela pocos kilómetros, y ya no es fuerte" {
        pepita.volar(40)
        assert.notThat(pepita.esFuerte())
    }
}
```

Al inicio, necesitamos hacer el **import** para incluir el código a testear, indicando el nombre de archivo donde se encuentran definidos los objetos. Se recomienda poner *nombreDeArchivo*.* para contemplar todas las entidades definidas en dicho archivo.

5.1. Describe

Identificamos a un conjunto de tests con la palabra reservada **describe** y un nombre expresivo con el cual lo identificamos. Al igual que los objetos y las clases, se utilizan { } para delimitar el inicio y fin de de la presente entidad, agrupando a todos los tests que forman parte de ella.

5.2. Test

Dentro del describe, cada test se define de la siguiente manera:

- Cada uno comienza con la palabra reservada **test** seguido de una cadena de caracteres que explique lo que se está probando. Es importante ponerle una

buena descripción porque esa misma leyenda es la que va a aparecer en el informe de errores, y cuando el test falle nos va a permitir detectar más fácilmente cuál fue el problema: es bueno identificar la clase de equivalencia, es decir, el escenario de prueba y no entrar en detalles redundantes (decir que la energía queda en 85 implica que si cambian las reglas de cálculo debemos cambiar en dos lugares).

- Se usan { } para marcar el inicio y fin del test, de igual manera que para los métodos.
- En las primeras líneas, salvo tests muy sencillos, generalmente es necesario enviar los mensajes que van configurando la situación a probar. Es lo que en los casos de pruebas denominamos "acciones". Algunas bibliografías separan la instanciación de objetos de prueba y las acciones propiamente dichas, en lo que se denomina el patrón Arrange-Act-Assert.
- Por último se le envía un mensaje al objeto **assert** para realizar concretamente la prueba, contrastando lo que se que se quiere probar con lo que se espera. Es lo que en los casos de prueba titulamos "validación". Hay básicamente tres alternativas:
    - El mensaje típico es **equals**, con dos parámetros: en primer lugar va el valor que se espera que retorne el mensaje que va en segundo lugar. Se lo puede ver en la mayor parte de los tests.
    - En los casos donde se espera que la respuesta sea **true**, se le envía el mensaje **that**, con un único parámetro que es el mensaje booleano que se espera que se valide afirmativamente. Se lo puede ver en el segundo test.
    - En forma análoga, existe un mensaje **notThat** que testea que no se verifique el mensaje booleano. Se lo puede ver en el último de los tests.

En los primeros dos tests, como se trata de probar mensajes que tienen valor de retorno y parten de la situación inicial, no hizo falta enviar mensajes previo al assert. Del tercero al último se busca probar que funcionen adecuadamente métodos con efecto como volar o comer, se envían primero dichos mensajes y luego el mensaje que se coloca en el assert es uno muy sencillo que se limita a exhibir el efecto causado. Los últimos 2 tests son similares en su estructura, aunque lo que se desee probar sea el mensaje que va en el assert.

Dentro de un test escribimos un assert por caso de prueba (esto puede implicar uno o más asserts en el test, lo importante es que todos tengan que ver con el mismo escenario que estamos probando).

También es válido tener tests "sueltos", sin declarar el **describe**, pero lo que se estila y recomienda es agruparlos por su coincidencia en cuanto a los objetos que

utilizan. Más adelante explicaremos otras características que refuerzan la idea de agrupar los test.

Es posible tener más de un describe por archivo, aunque si pertenecen a unidades diferentes (ej: aves, entrenadores) es conveniente definirlo en archivos diferentes.

6. Ejecución de los test

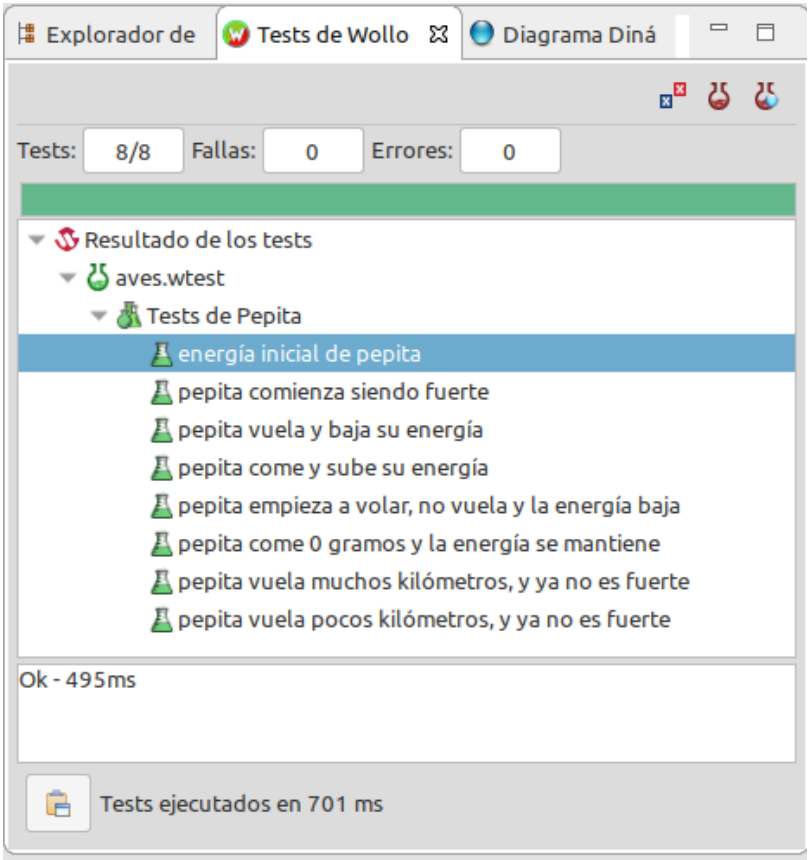
Teniendo definidos los tests, es momento de probar nuestra solución.

En el IDE, teniendo seleccionado un archivo de tests, se clickea el botón de ejecutar (En el menú, Ejecutar > Ejecutar <Ctrl F11>).

En caso que se tengan definidos varios archivos de tests, además de ejecutarlos uno por uno de la manera explicada, se los puede correr todos juntos (se selecciona el proyecto, menú, Ejecutar > Ejecutar como > Ejecutar todos los tests del proyecto Wollok).

6.1. Éxito

Asumiendo que contamos con el código de la solución y los tests tal como se plantearon, al ejecutar se activa la solapa de testing en el IDE, se corren todos los tests definidos y se muestra lo siguiente:



¿Cuál es nuestra salida? En este caso, todos los tests pasaron las verificaciones exitosamente:

- La cantidad de tests ejecutados coincide con los definidos. La cantidad de fallas y errores es 0.
- Se visualiza una barra de color verde.
- Para cada test, un tilde verde y la descripción.
- Al hacer click sobre cada ítem, se muestra simplemente un Ok debajo.

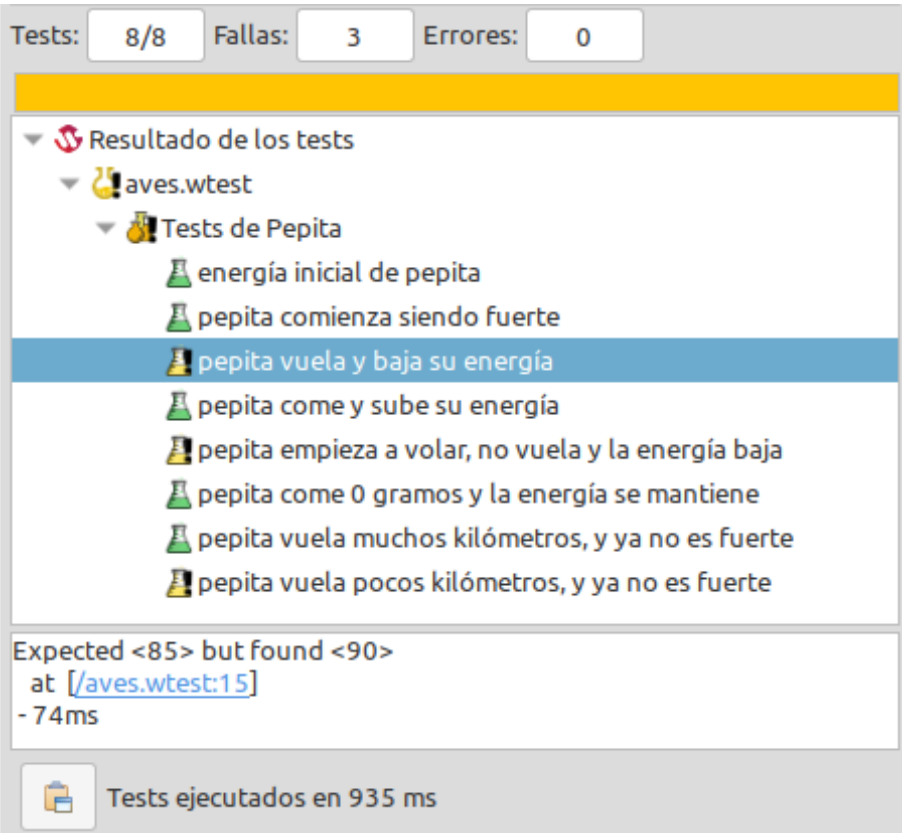
Nos quedamos tranquilos que nuestra solución funciona.

6.2. Fallas

En nuestro primer ejemplo pasaron correctamente todos los tests, ciertamente es lo deseable, pero es algo que no siempre nos va a pasar. Modifiquemos el código para que el programa no funcione adecuadamente. ¿Qué pasa si hacemos que pepita al volar le sume 5 como valor fijo en lugar de 10?

```
method volar(kms) { energia = energia - (kms + 5) }
```

Ejecutamos nuevamente los tests:



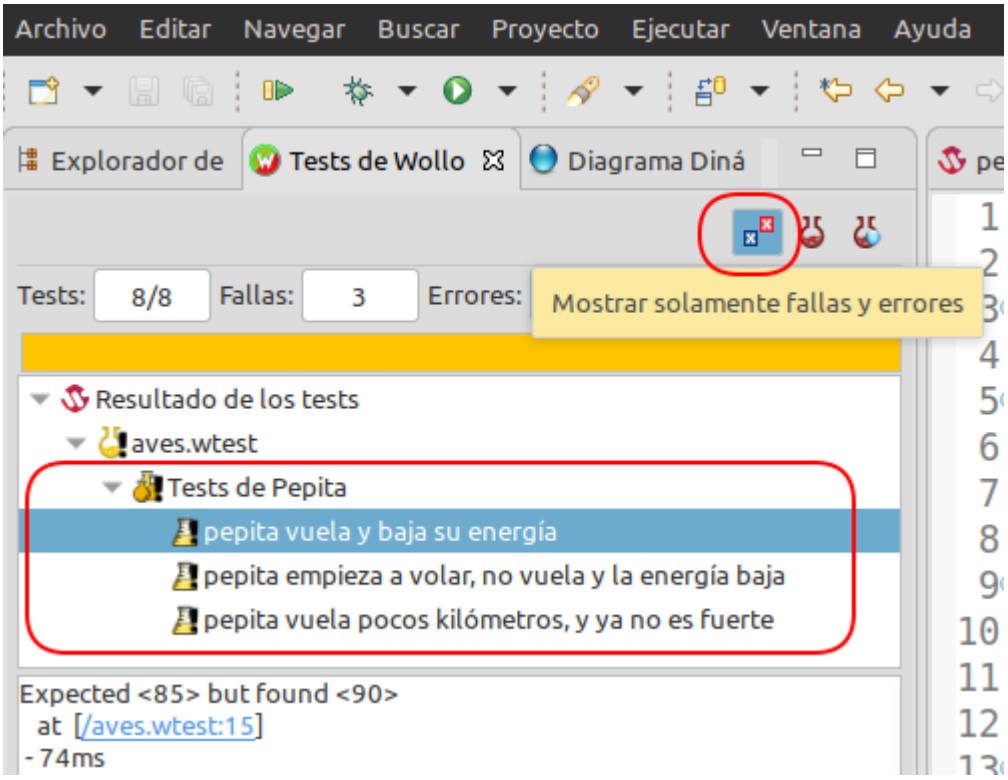
¿Cuál es nuestra salida ahora? Vemos que algunos test corrieron correctamente y otros fallaron, detectando problemas en el código.

- La cantidad de tests ejecutados coincide con los definidos, pero en este caso hay fallas. La cantidad de errores permanece en 0.
- La barra está de color amarillo.
- Todos los tests con su descripción, los validados con tilde verde y los que fallaron con un signo de admiración **!**.
- Al hacer click sobre los item de los test con fallas, vemos debajo el detalle de la información del test:

```
Expected [85] but found [90]
     at  [/aves.wtest:15]
     ...
```

Detectó precisamente la diferencia entre lo que esperaba el test y el resultado que arroja nuestro código: al volar 5 kilómetros, hizo (100 - (5 + 5)) = 90 que difiere de los 85 esperados de acuerdo a la definición original del problema.

Si activamos el ícono que muestre solamente los errores y fallas, aparecen únicamente los tests que no salieron ok (algo útil cuando tenemos que cerrar alguna entrega y queremos focalizar en los tests que andan mal)

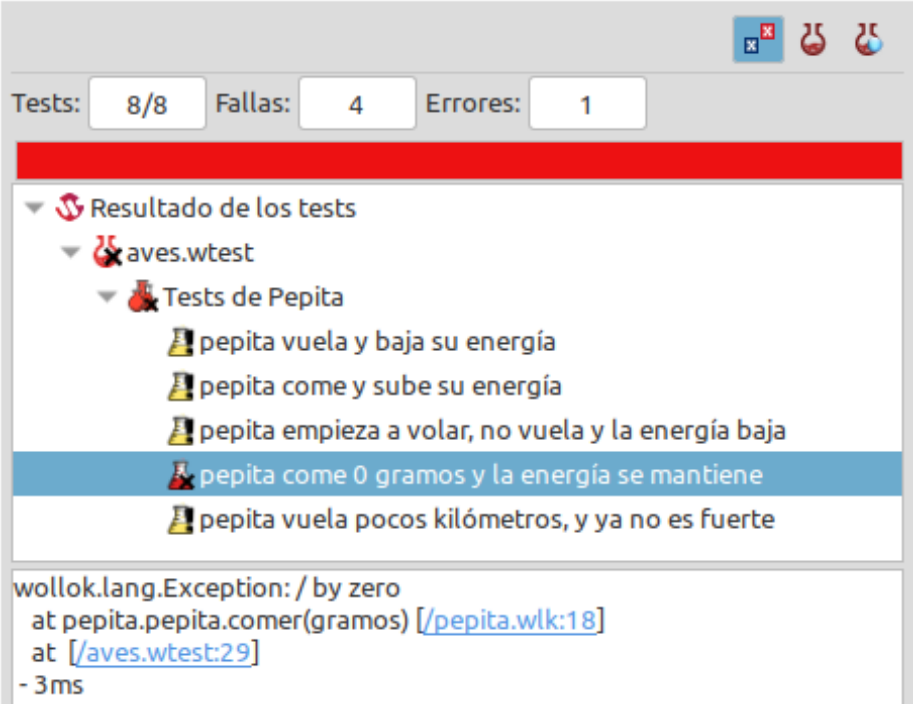


6.3. Errores

Ahora veamos qué pasa si modificamos el código de la solución, de una manera más contundente que eventualmente produzca un error de ejecución. Si gramos es 0, como ocurre en uno de los tests, la división da un error:

```
method comer(gramos) {
      energia = energia + 4 / gramos
}
```

Corremos nuevamente los tests:



¿Cuál es la nueva salida? Vemos que algunos test corrieron correctamente y otros dieron error.

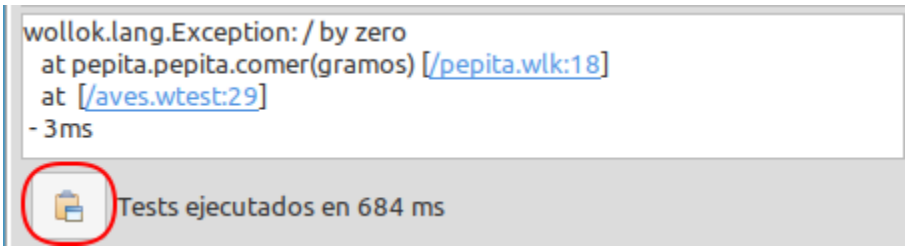
- La cantidad de tests ejecutados coincide con los definidos, en este caso hay tanto fallas como 1 error. Nótese que el error no interrumpió la ejecución de los restantes tests, y el test que comenzó a ejecutarse y dio error, se contabiliza como ejecutado, aun sin haber podido concluir.
- La barra está de color rojo (evocando la convención del semáforo) ya que el error tiene precedencia sobre las fallas.
- Todos los test con su descripción, los validados con tilde verde y los que dieron error tienen una **X** resaltada en rojo.

- Al hacer click sobre el ítem del test con error, vemos debajo el detalle del error.

```
wollok.lang.Exception: / by zero
      at pepita.pepita.comer(gramos) [/pepita.wlk:18]
      at [/testPepita.wlk:29]
```

Ese test me permite identificar cuál fue el motivo y en qué línea de código se detectó el error. Se puede observar la pila de envío de mensajes (*stack trace*). Podemos hacer click en cada línea para acceder al código fuente wollok correspondiente.

También podés hacer click en el botón que está abajo para copiar la pila de mensajes en el portapapeles y eventualmente enviarlo por mail.



6.4. Resultados del test

Cada uno de los test tiene tres salidas posibles

- ¡Todo bien! Ok. El código hace lo que esperamos.
- ¡Falló! El código no hace lo que esperábamos. ¡Menos mal que hicimos un test para detectarlo!
- ¡Error! No se pudo hacer la validación. Hay errores que impidieron completar su ejecución.

Es importante notar que cuando hubo fallas o errores, no tuvimos que tocar el código en los tests. Cambiamos el código de pepita, y pudimos verificar su comportamiento en todos los casos ejecutando el mismo conjunto de tests. Es lo que se denomina un **test de regresión**[2].

7. Resumen

El testeo unitario es una herramienta fundamental para poder garantizar que cada objeto está cumpliendo con la responsabilidad que le fue encomendada. Para lograr este objetivo, además de las tradicionales pruebas manuales, podemos definir tests

[2] Pruebas que se hacen sobre una pieza de software existente que sufrió modificaciones. Para más información ver Regression Testing.

unitarios que son automatizados, independientes y aislados, que especifican una serie de condiciones a verificar. Tener una gran batería de los tests no es sinónimo de un correcto diseño, pero constituye un buen punto de partida. Además fomenta el abandono de malas prácticas como “código que anda no se toca”, por una mucho más sana en la cual el miedo a modificar el software se contrasta con la posibilidad de correr pruebas de regresión todas las veces que sea necesario.

Apéndice: interfaz de assert

`assert.that(condition)`
Verificar que la condición se cumple `assert.that(cliente.esMoroso())`
`assert.notThat(condition)`
Verificar que la condición sea falsa (false = ok) `assert.notThat(cliente.esMoroso())`
`assert.equals(actual, expected)`
Verificar que dos valores sean iguales `assert.equals(90, pepita.energia())`
`assert.notEquals(actual, expected)`
Verificar que dos valores sean distintos `assert.notEquals(90, pepita.energia())`
`assert.throwsException(block)`
Verificar que al ejecutar block (un bloque de código) se produce un error cualquiera `assert.throwsException({monedero.sacar("A")})`
`assert.throwsExceptionLike(exceptionExpected, block)`
Verificar que al ejecutar block (un bloque de código) se produce la misma excepción que la exceptionExpected, con el mismo mensaje assert.throwsExceptionLike(new BusinessException("hola"), { throw new BusinessException("hola") } ✅ assert.throwsExceptionLike(new BusinessException("chau"), { throw new BusinessException("hola") } ⚠️(no coincide el mensaje) assert.throwsExceptionLike(new OtherException("hola"), { throw new BusinessException("hola") } ⚠️(no coincide la excepción)
`assert.throwsExceptionWithMessage(errorMessage, block)`

	Verifica que al ejecutar block (un bloque de código) se produce el mismo mensaje de error que el esperado (errorMessage), sin importar el tipo de excepción que se trate assert.throwsExceptionWithMessage("hola",{ throw new UserException("hola") } ✅ assert.throwsExceptionWithMessage("hola",{ throw new OtherException("hola") } ✅ assert.throwsExceptionWithMessage("chau",{ throw new UserException("hola") }⚠️

Paradigma
Orientado a Objetos

Módulo 06: Objetos anónimos.
Repaso polimorfismo.

por Fernando Dodino
Versión 1.0
Enero 2018

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Ejercicio: viajes en taxi

2 Repaso de objetos

3 Pruebas

4 Implementando pasajeros de prueba con objetos anónimos

5 Evitando repetir ideas: method en un describe

6 Convivencia de objetos anónimos y wko

7 Resumen

1 Ejercicio: viajes en taxi

Para una aplicación de viajes en taxi queremos modelar varios choferes, pero en particular nos interesa Daniel, quien lleva a todos los pasajeros que no sean jóvenes (cada pasajero se define como joven o viejo según sus características)

2 Repaso de objetos

¿Cómo representaremos a Daniel que lleva a los pasajeros que no sean jóvenes? Mediante

- un objeto conocido Daniel
- que tiene como comportamiento un método llevaA(). ¿qué necesita ese método? ¿puede ser la edad? No, porque eso no determina si es viejo o no. Esa no es responsabilidad de Daniel, debe delegarlo en cada pasajero.

```
object daniel {
    method llevaA(pasajero) = !pasajero.esJoven()
}
```

3 Pruebas

Para verificar que la definición de Daniel es correcta, armamos los casos de prueba:

Caso de prueba	Resultado esperado
Preguntamos si Daniel lleva a un pasajero joven	No lo lleva
Preguntamos si Daniel lleva a un pasajero que no es joven	Sí, lo lleva

Bastante simple, solo que para poder implementar los tests necesito tener

- a Daniel, que es un objeto autodefinido (*wko*)
- un pasajero joven
- un pasajero que no sea joven

Pero todavía no tengo aquí demasiadas definiciones sobre los pasajeros. Aquí tenemos dos opciones:

- hacer un relevamiento sobre el comportamiento esperado de los pasajeros
- o bien no atacar este problema pero sí cerrar una prueba unitaria sobre Daniel, que es lo que vamos a hacer a continuación.

4 Implementando pasajeros de prueba con objetos anónimos

Wollok permite crear objetos anónimos, que se asignan a una referencia y se pueden utilizar en ese contexto. Vemos la implementación de ambos tests:

```
import choferes.*

describe "Tests de Daniel" {

	test "Daniel no lleva a un pasajero joven" {
		const pasajeroJoven = object {
			method esJoven() = true
		}
		assert.notThat(daniel.llevaA(pasajeroJoven))
	}

	test "Daniel lleva a un pasajero que no es joven" {
		const pasajeroViejo = object {
			method esJoven() = false
		}
		assert.that(daniel.llevaA(pasajeroViejo))
	}

}
```

En cada test definimos un objeto que no tiene nombre (a diferencia de los objetos autodefinidos o *well-known objects*), asignándolo a las referencias locales pasajeroJoven y pasajeroViejo, respectivamente. Esa referencia está disponible durante la ejecución de cada uno de los tests.

¿Qué interfaz necesitamos tener para el pasajero? Solo es preciso que implemente el método esJoven, sin parámetros y que devuelva un valor booleano.

5 Evitando repetir ideas: method en un describe

El lector habrá notado que en el ejemplo anterior estamos repitiendo la misma idea al crear el objeto anónimo

```
test "Daniel no lleva a un pasajero joven" {
	const pasajeroJoven = object {
		method esJoven() = true
	}
```

```
	assert.notThat(daniel.llevaA(pasajeroJoven))
}

test "Daniel lleva a un pasajero que no es joven" {
	const pasajeroViejo = object {
		method esJoven() = false
	}
	assert.that(daniel.llevaA(pasajeroViejo))
}
```

Cuando los pasajeros tienen más rasgos diferenciales, como la edad, la altura, el peso, etc. o bien cuando aparecen nuevos choferes que también precisan tener un pasajero, crear los objetos anónimos es una tarea que se vuelve más burocrática y tediosa. Para ello vamos a abstraer la creación del pasajero en un **método** dentro del **describe**, donde vamos a permitir pasarle como parámetro si es o no joven:

```
import choferes.*

describe "Tests de Daniel" {

	method crearPasajero(joven) = object {
		method esJoven() = joven
	}

	test "Daniel no lleva a un pasajero joven" {
		const pasajeroJoven = self.crearPasajero(true)
		assert.notThat(daniel.llevaA(pasajeroJoven))
	}

	test "Daniel lleva a un pasajero que no es joven" {
		const pasajeroViejo = self.crearPasajero(false)
		assert.that(daniel.llevaA(pasajeroViejo))
	}

}
```

Aquí vemos que crearPasajero es un método que devuelve un objeto anónimo, algo que también podemos escribir así:

```
	method crearPasajero(joven) {
		return object {
			method esJoven() = joven
```

```
        }
    }
```

El método recibe como parámetro si es joven, y lo utiliza en la definición del método esJoven.

6 Convivencia de objetos anónimos y wko

¿Es necesario hacer esto en todos los casos? No, por supuesto que eso depende de las definiciones de nuestro negocio. Por ejemplo, tenemos a la pasajera Magalí que tiene 19 años, y cree que una persona es joven hasta los 30.

```
object magali {

    var edad = 19

    method cumplirAnios() {
        edad = edad + 1
    }

    method esJoven() = edad < 30
}
```

Daniel entonces no llevará a Magalí, como podemos comprobarlo en este test[1]:

```
    test "Daniel no lleva a alguien que piensa que es joven" {
        assert.notThat(daniel.llevaA(magali))
    }
```

Lo interesante es que Magalí y nuestros objetos anónimos son polimórficos en el contexto de uso de Daniel, ya que todos implementan el método esJoven().

7 Resumen

En este capítulo hemos conocido los objetos anónimos, que sirven para generar abstracciones sencillas. Esto resulta útil cuando no nos interesa reutilizarlos en otro contexto, como en el testeo unitario. Los objetos anónimos tienen atributos y comportamiento como los *wko*, y pueden ser utilizados en forma polimórfica entre sí.

[1] Fíjense que en este caso el test no dice “Daniel no lleva a Magalí”, sino qué es lo que está probando, en una forma más general y menos sujeta a cambios. Si las reglas de negocio cambian, solo debemos modificar el assert y no la descripción del test.

Paradigma Orientado a Objetos

Módulo 07: Propiedades

por Fernando Dodino

Versión 1.0

Enero 2018

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Repaso de accessors

2 Código boilerplate

3 Propiedades en Wollok
- 3.1 Definición
- 3.2 La propiedad constante solo define getter
- 3.3 Redefiniendo accessors implícitos

4 Resumen

Nota: este feature estará disponible a partir del release 1.6.4

1 Repaso de accessors

Hemos visto en el módulo 2 que para poder acceder a la referencia de un objeto, éste debe definir accessors, en particular los getters y setters:

```
object pepita {
	var energia = 0

	method energia() = energia	// GETTER

	method energia(_energia) {	// SETTER
		energia = _energia
	}
	method volar() {
		energia = energia + 10
	}
}
```

De esa manera, podemos desde otra entidad (otro objeto, un test o un programa) conocer el valor de la referencia energia para pepita:

```
import pepita.*

describe "tests para pepita" {

	test "energia inicial para pepita" {
		assert.equals(100, pepita.energia())
	}

	test "energia para pepita luego de volar" {
		pepita.volar()
		assert.equals(90, pepita.energia())
	}
}
```

En el ejemplo recién contado, tomamos la decisión de publicar el getter y el setter, aunque en realidad solo estamos utilizando el getter en el test. De hecho, podríamos cuestionar si definir el setter es una decisión de diseño correcta, ya que la energía se va modificando cada vez que pepita vuela (y eventualmente cuando coma).

2 Código boilerplate

Pensemos en la definición de un auto cualquiera:

```
object dodge1500 {
      var color
      var kilometraje
      var conAire = false
      const patente = "RVM 363"
      const anioPatentamiento = 1979
}
```

Para poder definir los atributos, tuvimos que pensar qué referencias se pueden modificar y cuáles no: por eso la patente y el año son inmutables, mientras que permitiremos ponerle aire al auto, o cambiarle el color. Por supuesto el kilometraje se incrementará a medida que el auto vaya rodando.

La segunda decisión que acompaña a los atributos, es qué tipo de *accessors* definiremos para cada uno:

- para las referencias const vamos a publicar solo los getters
- en el caso del aire y el color, podemos publicar tanto getters como setters
- y en el caso del kilometraje, podríamos publicar un setter, pero vamos a preferir escribir un getter y tener un método de negocio realizarViaje(kilometros) para que incremente el kilometraje.

```
object dodge1500 {
      var color
      var kilometraje
      var conAire = false
      const patente = "RVM 363"
      const anioPatentamiento = 1979

      method patente() = patente
      method anioPatentamiento() = anioPatentamiento
      method conAire() = conAire
      method conAire(_conAire) { conAire = _conAire }
      method color() = color
      method color(_color) { color = _color }
      method kilometraje() = kilometraje
      method realizarViaje(kilometros) {
            kilometraje = kilometraje + kilometros
      }
```

```
}
```

La definición está llena de ***código boilerplate***, burocracia que no solo es tediosa de escribir, sino que también se entremezcla con el código que verdaderamente nos importa, el que está asociado a decisiones del negocio.

3 Propiedades en Wollok

3.1 Definición

Una propiedad en Wollok permite explicitar un contrato específico para una referencia:

- si la referencia es variable (var), Wollok genera su getter y setter
- si la referencia es constante (const), Wollok genera el getter

La creación de los accessors es **implícito**, no se visualiza en el código ya que la idea es conservar únicamente el comportamiento definido por el negocio.

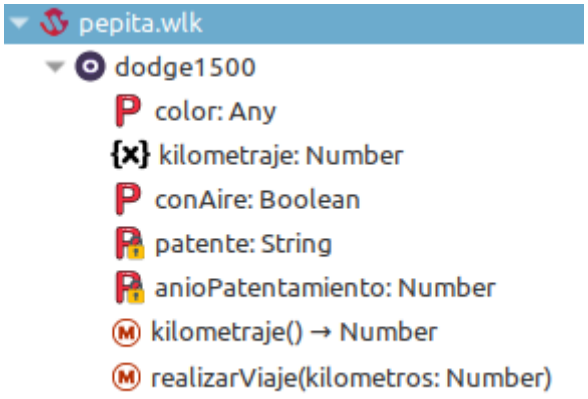
El mismo ejemplo de arriba quedaría así:

```
object dodge1500 {
	var property color
	var kilometraje
	var property conAire = false
	const property patente = "RVM 363"
	const property anioPatentamiento = 1979

	method kilometraje() = kilometraje

	method realizarViaje(kilometros) {
		kilometraje = kilometraje + kilometros
	}
}
```

Como no queremos publicar un setter para el atributo kilometraje, elegimos

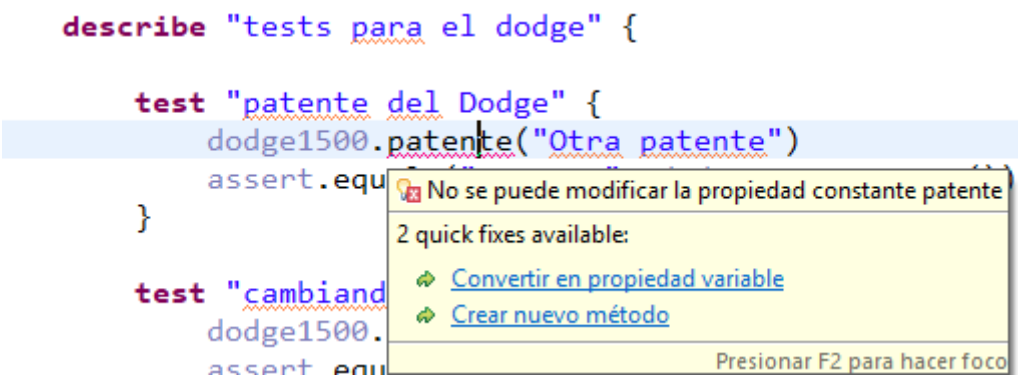
- no aplicar el modificador property para kilometraje
- y definir manualmente un getter específico para kilometraje

En la vista Esquema (Outline) se visualizan las propiedades (con una P) vs. atributos (demarcados con **{x}**). Si son constantes veremos un candado abajo a la derecha:



3.2 La propiedad constante solo define getter

Recordemos que si la patente es una propiedad cuya referencia es constante, no tendremos setters:



3.3 Redefiniendo accessors implícitos

En el caso de definir un método accessor propio, éste pisa el implícito que define la property, tanto para getters como para setters. Continuando con el ejemplo del auto

```
object dodge1500 {
    ...
    const property anioPatentamiento = 1979

    method anioPatentamiento() = anioPatentamiento + 1
```

Si construimos un test específico para el año, veremos que se toma en cuenta la definición explícita del método anioPatentamiento() escrita por nosotros:

```
describe "tests para el dodge" {
```

```
test "anio patentamiento de un auto común" {
      assert.equals(1980, dodge1500.anioPatentamiento())
}
```

4 Resumen

Wollok permite la definición de propiedades, lo que evita al programador tener que escribir o leer el código de los accessors (getters y/o setters) cuya necesidad es meramente administrativa.

Paradigma
Orientado a Objetos

Módulo 08: Clases.
Method lookup y polimorfismo con clases.

por Valeria Pennella
Fernando Dodino
revisado por Matías Freyre
revisado por Lucas Spigariol
Versión 1.4
Septiembre 2019

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Introducción al concepto de clase

2 Clase e instancia

3 Estructura de un objeto

4 Method lookup con clases

5 Clases y tipos

6 Polimorfismo con clases

7 Tipos y polimorfismo

8 Repaso: objetos anónimos, wko y clases

9 Resumen

1 Introducción al concepto de clase

Necesitamos modelar un sistema para un comercio de ventas de libros. Sabemos que vendemos libros como:

- El Principito
- Martín Fierro
- El Reino del Revés

Vamos a tener 3 objetos, donde cada uno representa un libro diferente pero todos tienen características similares. Cuando a cada libro le pregunte si es caro cada uno va a responder lo que corresponda según su precio. Es decir

```
>>> elPrincipito.esCaro()
>>> martinFierro.esCaro()
>>> elReinoDelReves.esCaro()
```

Si creamos El Principito tendríamos esta definición:

```
object elPrincipito {
      var property precio = 100
      method esCaro() = precio > 150
      method aumentar(aumento) { precio = precio + aumento }
}
```

Considerando que el precio inicial de cada libro es 100, pero luego cada uno puede variar independientemente definimos los otros dos libros de la siguiente manera:

```
object martinFierro {
      var property precio = 100
      method esCaro() = precio > 150
      method aumentar(aumento) { precio = precio + aumento }
}

object elReinoDelReves {
      var property precio = 100
      method esCaro() = precio > 150
      method aumentar(aumento) { precio = precio + aumento }
}
```

Luego, podemos hacer

```
>>> elPrincipito.aumentar(80)
>>> martinFierro.aumentar(50)
>>> elReinoDelReves.aumentar(100)
```

y ante la misma pregunta, las respuestas van a ser diferentes

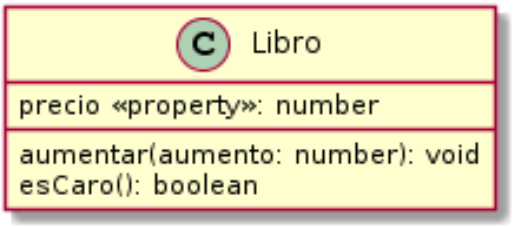
```
>>> elPrincipito.esCaro()      // true
>>> martinFierro.esCaro()      // false
>>> elReinoDelReves.esCaro()   // true
```

Lo que vemos es que si bien cada uno de los libros tienen un precio diferente, todos los libros tienen el mismo comportamiento para decir si es caro; también es la misma la forma de aumentar el precio. Para no tener código duplicado podemos definir una clase Libro donde esté definido el comportamiento de todo libro. Es decir...

```
/**
 * Definición de la clase Libro
 */
class Libro {
      var property precio = 100
      method esCaro() = precio > 150
      method aumentar(aumento) { precio = precio + aumento }
}
```

De esta manera, toda la lógica que teníamos definida en cada uno de los objetos, como es igual para todos, podemos tenerla directamente en un solo lugar que es la Clase.

Representamos la clase Libro en un diagrama de clases UML



Aquí vemos que una clase se representa con un rectángulo con 3 divisiones:

- arriba está el nombre de la clase
- en el medio las referencias, ya sean constantes o variables
- en el último compartimento escribimos los métodos

2 Clase e instancia

¿Y cómo hago ahora para crear cada uno de los libros? A partir de la clase Libro, voy a crear cada uno de los diferentes ejemplos concretos.

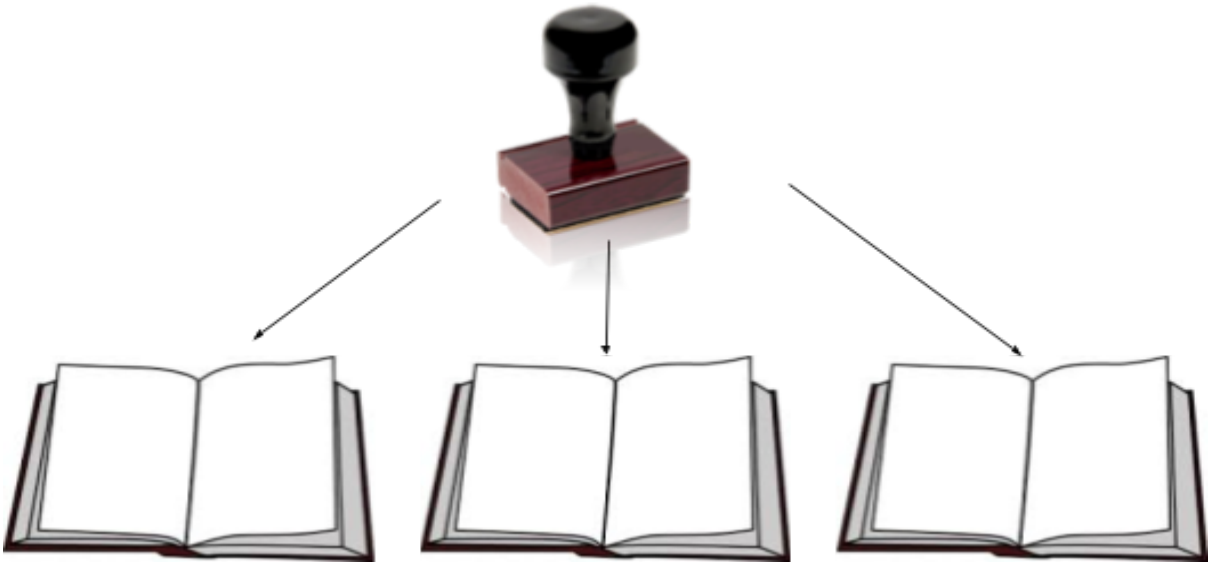


Fig.1: La clase puede verse como un *sello* con el que generamos las instancias

Otra analogía es ver la clase Libro como un molde a partir del cual vamos a crear las instancias es decir:

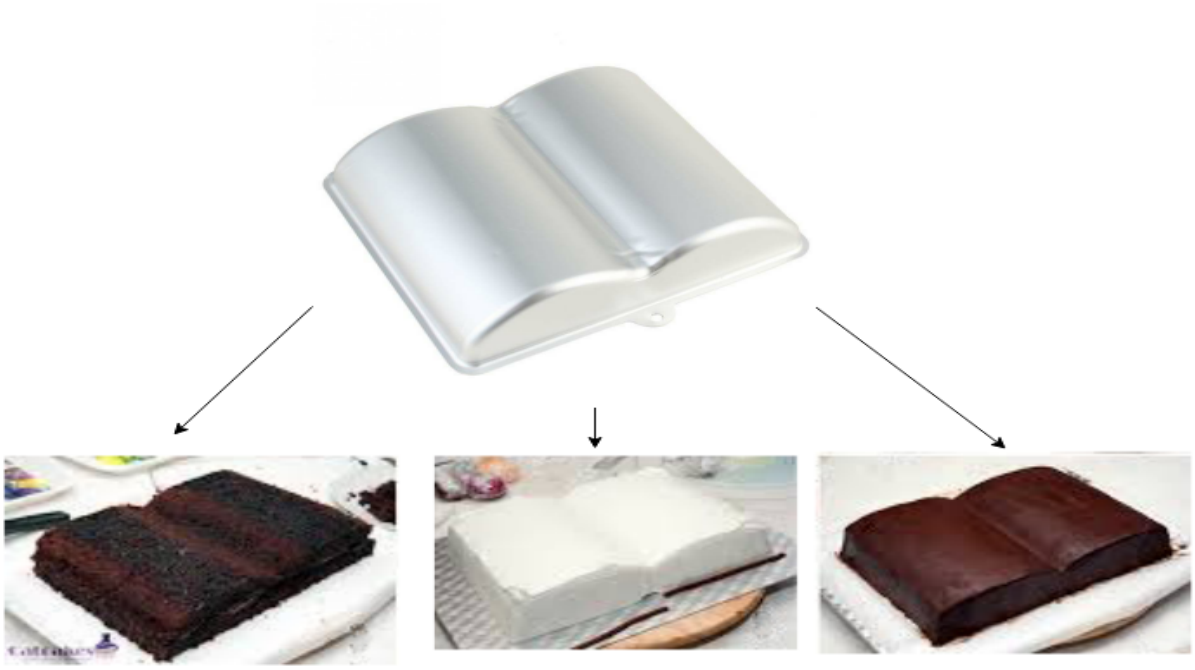


Fig. 2: con el *molde* (hueco) instanciamos una chocotorta, una torta glaseada y una bombón

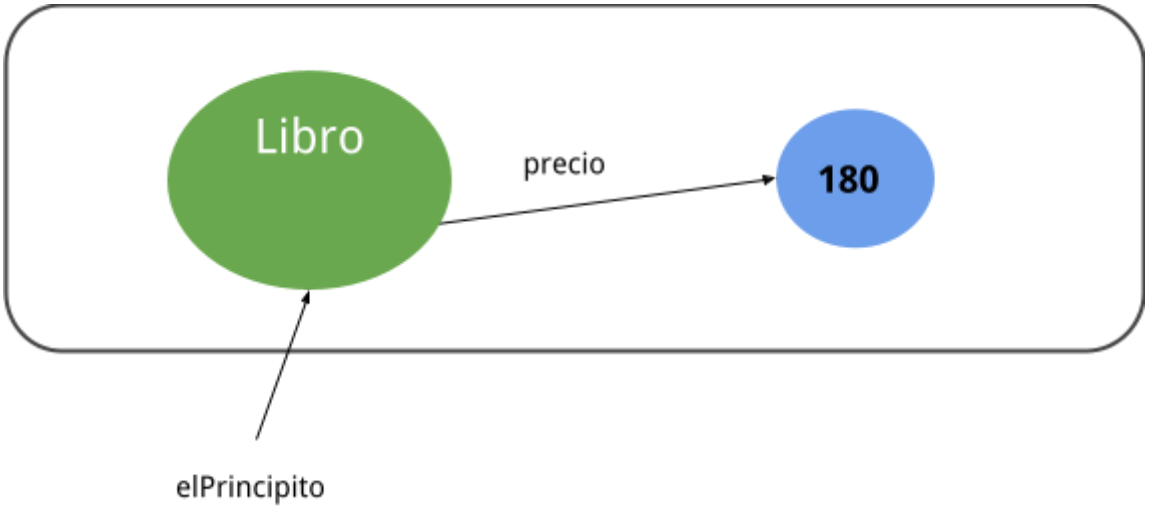
Para crear el objeto `elPrincipito` escribimos:

```
>>> const elPrincipito = new Libro()
```

Para tener un nuevo libro creamos con new, una nueva instancia[1] de la clase Libro. Por lo tanto la variable `elPrincipito` va a apuntar a una instancia de `Libro`.

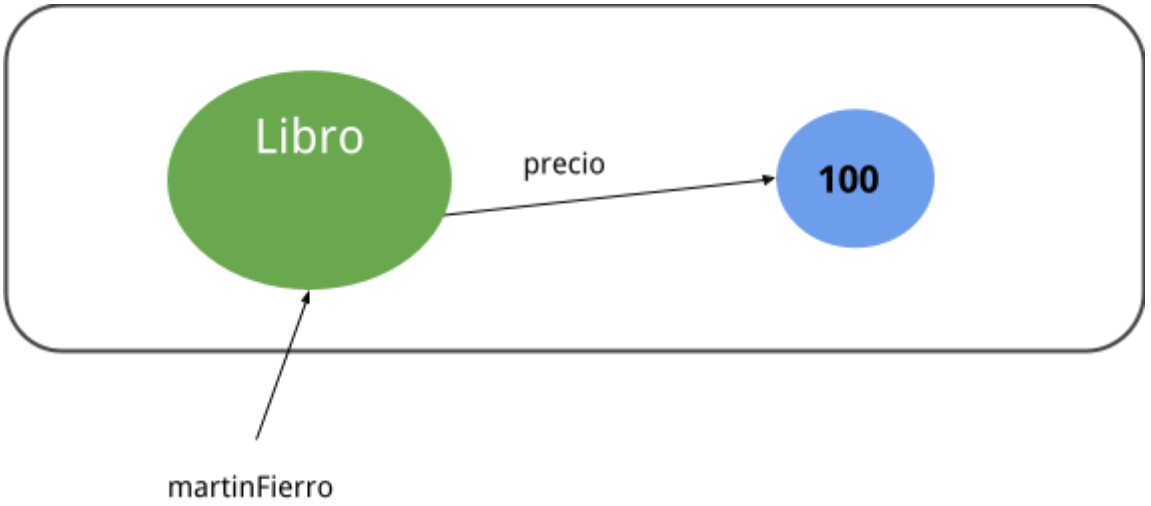
Y voy a poder decirle al principito que aumente de precio y preguntarle si es caro

```
>>> elPrincipito.aumentar(80)
>>> elPrincipito.esCaro()
```



De la misma manera puedo crear una referencia a martinFierro, que será un libro.

```
>>> const martinFierro = new Libro()
```



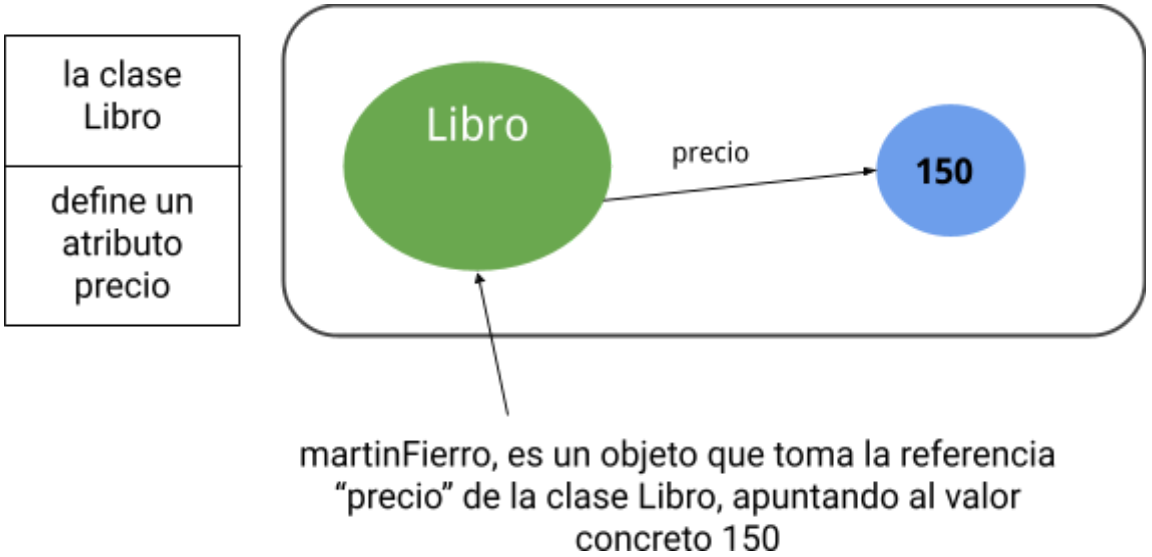
[1] Hablamos de instancia como sinónimo de objeto

3 Estructura de un objeto

Al instanciar un objeto, éste tiene todas las referencias definidas en su clase (lo que también se llama **estructura**), solo que con sus valores particulares. En el ejemplo anterior, Libro define que todo libro tiene una referencia a precio.

El valor de cada libro es propio de cada instancia, por eso las referencias o atributos se llaman también **variables de instancia**.

```
>>> const martinFierro = new Libro()
>>> martinFierro.aumentar(50)
```



4 Method lookup con clases

Nuestros objetos ya no tienen comportamiento, sino que definimos esa responsabilidad en la clase. Debemos entonces cambiar nuestro mecanismo de method lookup.

Según esta definición: qué sucede cuando enviamos el mensaje...

```
>>> elPrincipito.esCaro()
```

elPrincipito no es un objeto que tenga comportamiento explícito. Es una instancia de una clase Wollok. Entonces vamos a buscar la definición a la clase a la que pertenece elPrincipito. Ahí vemos que existe una definición del método esCaro

```
method esCaro() = precio > 150
```

Y se termina resolviendo para los atributos propios del objeto elPrincipito: como la referencia precio de este objeto apunta a 180, elPrincipito es un libro caro.

De la misma manera si preguntamos si el Martín Fierro es caro...

```
>>> martinFierro.esCaro()
```

Nuevamente vamos a buscar la definición a la clase a la que pertenece martinFierro, que es como suponemos la clase Libro.

```
method esCaro() = precio > 150
```

Y se termina resolviendo para los atributos propios del objeto martinFierro: como precio referencia a 150, martinFierro **no** es un libro caro.

5 Clases y tipos

Un tipo es un conjunto de mensajes que entiende un objeto.

Un caso particular es considerar todos los mensajes que entiende un objeto, entonces el tipo está definido por la clase. En nuestro ejemplo anterior, el Libro define un tipo que entiende estos mensajes:

- precio() y precio(_precio): implícitos por la definición de la propiedad precio
- esCaro()
- aumentar(aumento)

Pero además de la clase, nosotros podemos definir otros tipos para el mismo objeto. Por ejemplo, podemos definir que un libro, un teléfono celular o un auto comparten en común un tipo: todos nos saben decir si son caros, independientemente de cómo lo implementen (podríamos no tener referencias al precio).

Repasamos entonces: ¿de qué tipo es un objeto? Puede ser de muchos, “El Principito” puede ser

- un libro
- un objeto que sabe decirnos si es caro
- o un objeto de una clase Wollok genérica, que sabe mostrarse por consola

Cada una de estas características define un **tipo**.

Los tipos permiten encontrar similitudes entre clases que aparentemente no tienen conexión, como veremos en el siguiente ejemplo.

6 Polimorfismo con clases

Además de libros, en el local agregamos la venta de DVD's y láminas. De todos los productos se conoce su precio.

- En el caso de los DVD el precio de venta es el precio por minuto * cantidad de minutos del dvd. Además, se sabe que es caro si la cantidad de minutos es mayor a 100.
- En cambio en las láminas el precio es el ancho * alto * precio base del material con el cual está pintada (ej.: acuarela, óleo, crayón, etc). Y se sabe que es cara si el precio base del material es mayor a 50 pesos.

Si bien todos los productos definen un método precio(), el cálculo del precio es diferente para cada producto, ya sea libro, DVD o lámina. Como el comportamiento es diferente, necesitamos clases diferentes. Por lo tanto vamos a crear la clase DVD y la clase Lámina en Wollok así:

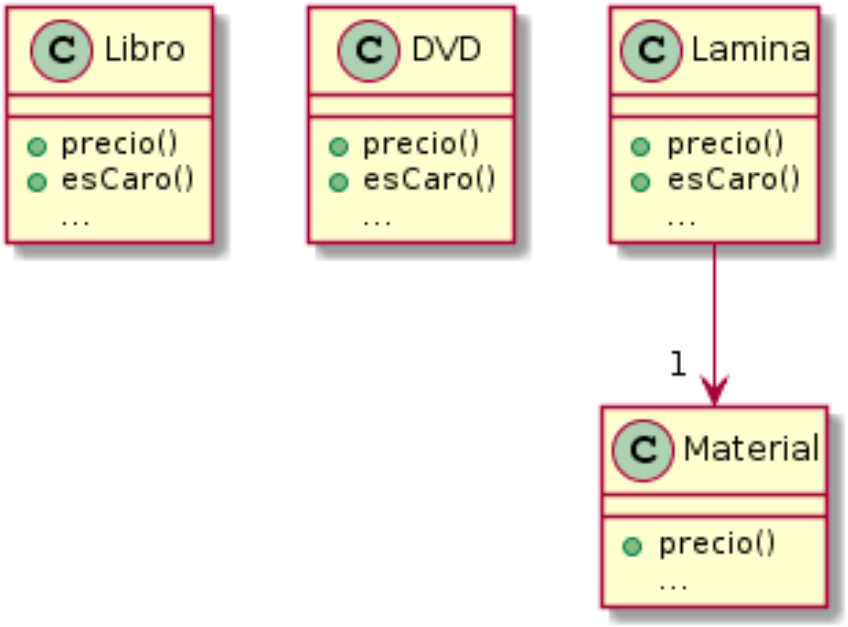
```
class DVD {
      var property precioMinuto = 10
      var property cantidadMinutos = 90

      method precio() = precioMinuto * cantidadMinutos
      method esCaro() = cantidadMinutos > 100
}

class Lamina {
      var property ancho = 29.7
      var property alto = 42
      var property material = new Material()

      method precio() = alto * ancho * self.precioBase()
      method esCaro() = self.precioBase() > 50
      method precioBase() = material.precio()
}
```

`self.precioBase()` se refiere al objeto receptor del mensaje: una instancia de la clase Lamina.

Además necesitamos en nuestro modelo una clase Material de la cual conocemos su precio.

```
class Material {
      var property precio = 5
}
```

El diagrama de clases nos quedaría[2]:



Vamos a crear las instancias de la clase DVD y de la clase Lamina. Ambos objetos entienden el mensaje precio().

```
const jurassicPark = new DVD()
const laGioconda = new Lamina()
```

De esta manera, ya están los objetos instanciados correctamente y se les puede enviar a cualquiera de ellos el mensaje precio().

```
jurassicPark.precio()
laGioconda.precio()
```

aunque el comportamiento de cada método precio() sea diferente.

[2] El lector podrá preguntarse: ¿no falta documentar las variables? ¿y algunos métodos?
La recomendación que hacemos es: no todo tiene que escribirse en el diagrama, solo lo que consideremos importante. Al fin y al cabo, la implementación puede verse en el código mismo. El diagrama está mostrando relaciones y comportamiento desde otro punto de vista.

7 Tipos y polimorfismo

Hemos visto de esta manera que una lámina, un DVD y un libro son polimórficos, porque aunque cada uno conserva su propia implementación, quien le envía mensajes no nota la diferencia y es el que aprovecha ese beneficio:

```
class Comprador {
	method comprar(algo) {
		if (algo.esCaro()) {
			...
```

Para hacer la pregunta esCaro(), la referencia a *algo* puede apuntar a

- un libro
- un DVD
- una lámina
- o a cualquier otro objeto autodefinido (wko) o de una clase que implemente el método esCaro()

El comprador, entonces, ni siquiera necesita conocer al objeto exacto al que le está hablando. Sabe solamente que le puede enviar el mensaje esCaro(), y eso

- abre el juego a que aparezcan nuevos objetos sin necesidad de modificar al Comprador (por eso decimos que hay un bajo acoplamiento entre ambos componentes: comprador por un lado y producto por el otro)
- permite dividir tareas: una persona puede codificar al Comprador y otra/s a los productos
- el polimorfismo entre estos productos se da en el contexto de la compra: esto no implica que se comportan por igual en todos los demás contextos. Ciertamente la forma de inicializar una lámina, un DVD y un libro es totalmente diferente. Además, en este modelo, los libros pueden aumentar su precio pero los demás elementos no.

8 Repaso: objetos anónimos, wko y clases

Ahora que incorporamos a las clases como una forma más de representar conceptos, podemos preguntarnos ¿cuándo conviene modelar a través de objetos anónimos, cuándo usar wko y cuándo clase?

- los **objetos anónimos** tienen un alcance acotado, por ejemplo un test, o la resolución de un requerimiento (un método y todo el encadenamiento de mensajes que sale de ese método). Lo mismo pasa cuando necesitamos construir un bloque para pasárselo a filter, o al map de una colección: el

objeto que representa al bloque no tiene nombre y no nos importa que lo tenga, existe para resolver el requerimiento puntual.

```
method taxistasLocales() =
      taxistas.filter { taxista => taxista.viveEn(zona) }
      // las llaves encierran un objeto anónimo que representa el
      // criterio de selección de un taxista (es un bloque)
```

- **objetos autodefinidos o wko**: son objetos conocidos, porque están representando un concepto de negocio dentro de la aplicación. Esto ocurre cuando un objeto tiene un comportamiento específico, y nos interesa modelarlo en forma separada de otros objetos.

- por último, las **clases** son importantes cuando existen múltiples objetos que comparten comportamiento y no tiene sentido que los nombre por separado: el viaje que hice ayer en colectivo, se parece mucho al viaje de la semana pasada. Si solo difieren en la información que guardan las referencias, el comportamiento se debe ubicar en un solo lugar para no repetir la misma idea una y otra vez.

9 Resumen

Hemos presentado a la clase como un concepto que permite agrupar los objetos con el mismo comportamiento y la misma estructura, más allá de que cada instancia guarde en cada referencia su propio juego de valores. Todo objeto debe pertenecer a una clase durante todo su ciclo de vida, y a su vez una clase tiene diferentes tipos, en base a los mensajes que define. Cuando dos objetos de diferente clase comparten el mismo tipo, son polimórficos en ese contexto: un observador le envía mensajes a los dos y no percibe cambios, aunque cada clase define un método propio.

Paradigma Orientado a Objetos

Módulo 09: Herramientas de instanciación

por Fernando Dodino
revisado por Lucas Spigariol
Versión 1.1
Julio 2019

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Objetos que no se instancian explícitamente

2 Instanciación sin parámetros

3 Instanciación con parámetros

4 Resumen

1 Objetos que no se instancian explícitamente

Los objetos anónimos se crean en forma programática:

```
const alumno = object {
	method estudia() = true
}
```

Mientras que los objetos nombrados (*wko*, como pepita) se instanciaban solos en el momento en que uno ejecuta la consola REPL, o bien un test, o un programa.

También están los literales de Wollok: objetos que tienen una sintaxis propia para construirse:

```
2          // permite referenciar al número 2
"hola"     // construye un String cuyo valor es "hola"
[7]        // construye una lista con un único elemento: el número 7
```

2 Instanciación sin parámetros

Si tenemos esta definición

```
class Ave {
	var energia = 100

	method volar(kms) { energia = energia - (kms * 5) }
}
```

sabemos que para crear un ave en la consola podemos hacerlo de esta manera:

```
>>> const cormoran = new Ave()
```

¿Cómo queda inicializado el cormorán? En este caso hay una sola referencia, energía, que tiene como valor 100. Podemos enviar el mensaje volar sin problemas, el objeto está correctamente construido.

3 Instanciación con parámetros

Como vimos en el ejemplo del Ave, cuando el objeto es lo suficientemente simple, nos alcanza con tener variables previamente inicializadas en su definición. Pero muchas veces esto es insuficiente o no es posible de realizar.

Wollok permite inicializar las referencias al construir el objeto, lo que ayuda a darle más claridad y precisión a la instanciación y garantizar que quede bien construido. Por ejemplo, queremos modelar la presentación de un recital, que define

- en qué lugar se realizará (puede cambiar)
- qué músicos tocarán
- el valor de la entrada general
- y cuántas entradas se vendieron hasta el momento

```
class Presentacion {
	var property lugar
	const property musicos
	var property valorEntrada
	var property entradasVendidas
	...
}
```

La ocasión para darle un valor coherente a cada variable es en el momento de la instanciación, y lo hacemos de esta manera:

```
const seruEnLuna = new Presentacion(
	lugar = lunaPark,
	musicos = [seruGiran],
	valorEntrada = 2000,
	entradasVendidas = 180)
```

Estos parámetros revelan claramente cuáles son las referencias a inicializar. Además, se puede hacer en cualquier orden:

```
const seruEnLuna = new Presentacion(
	entradasVendidas = 180,
	lugar = lunaPark,
	musicos = [seruGiran],
	valorEntrada = 2000)
```

Por otra parte

- Dado que es una sintaxis un tanto verbosa el IDE ayuda proponiendo las referencias a inicializar.
- Buena parte de la responsabilidad de inicializar bien depende de quien instancia a la presentación.

Para aquellas referencias, ya sean variables o constantes, que tengan definido un valor inicial, es opcional enviar un valor por parámetro.

- En caso que no se le envíe, mantiene en valor con que se inicializó.
- En caso que se envíe un valor por parámetro, se asigna éste en vez del definido en la inicialización.

Si se define a la presentación de esta manera

```
class Presentacion {
      var property lugar
      const property musicos = []
      var property valorEntrada
      var property entradasVendidas = 0
}
```

Y si se crea el objeto seruEnLuna como se lo hizo anteriormente, con todos los parámetros, el objeto queda instanciado de igual manera, pero ahora se puede instanciar sin necesidad de pasar por parámetros todos los valores.

```
const vamosAVerQuienEnLuna = new Presentacion(
      lugar = lunaPark,
      valorEntrada = 2000)
```

La presentación tiene un estado consistente, donde aún no se han vendido entradas y la lista de músicos está vacía. Todos los mensajes deberían funcionar correctamente.

En el caso particular en que todas las referencias están inicializadas, se puede hacer new sin parámetros.

La instanciación del objeto es un proceso de una sola pasada, que incluye la inicializacion de variables y su asignación mediante los valores que le pasamos a los parámetros. De esta manera, en el caso de una referencia constante que tiene un valor inicial y a la vez recibe un valor por parámetro, no se considera este seteo una modificación de la referencia, sino que ambas asignaciones se asumen como la asignación inicial.

En general, el formato para utilizar los parámetros nombrados es:

```
new ClaseAInstanciar(
      referencia1 = valor1,
      referencia2 = valor2,
      ...,
```

```
    referenciaN = valorN)
```

Las referencias no necesariamente deben ser propiedades (ni es necesario generarles *setters*), sí deben estar definidas como atributos válidos y pueden tener o no valores por defecto.

La buena noticia es que el IDE de Wollok nos alerta cuando hay referencias que no se inicializan ni se pasa el valor por parámetro.

4 Resumen

A lo largo de este capítulo hemos visto cómo instanciar un objeto e inicializar sus referencias para permitir un correcto uso del objeto. Las alternativas para inicializar son:

- En la declaración de las referencias de la clase se le puede asignar un valor inicial, que actúa como valor por defecto.
- Se le puede dar un valor a cada referencia haciendo una instanciación con parámetros con el new. Se especifica cada referencia con su correspondiente valor.

Las referencias que no se inicializan por parámetro, asumen por defecto su valor asignado en la declaración. Si una referencia fue inicializada en su declaración y luego recibe por parámetro un nuevo valor, queda con este último. Esta combinación de opciones permite flexibilidad, mantiene claridad en el código y garantiza que el objeto quede bien construido.

Paradigma Orientado a Objetos

Módulo 10: Manejo de errores

por Fernando Dodino, Pablo Tesone
revisión por Lucas Spigariol
Versión 2.2
Agosto 2019

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Errores y excepciones

2 Breve ejemplo: modelar un monedero

3 Nuestro objetivo

4 Solución en Wollok

5 Alternativas para modelar una excepción

- 5.1 Mirar a un costado
- 5.2 Códigos de error
- 5.3 Excepciones

6 Cómo generar una excepción

7 Dónde se atrapa

8 Excepciones de usuario y de programa

9 Tareas para el lector

10 Buenas y malas prácticas

- 10.1 To throw or not to throw
- 10.2 Consistencia en el valor de retorno de un método
- 10.3 Mejorando los mensajes de error

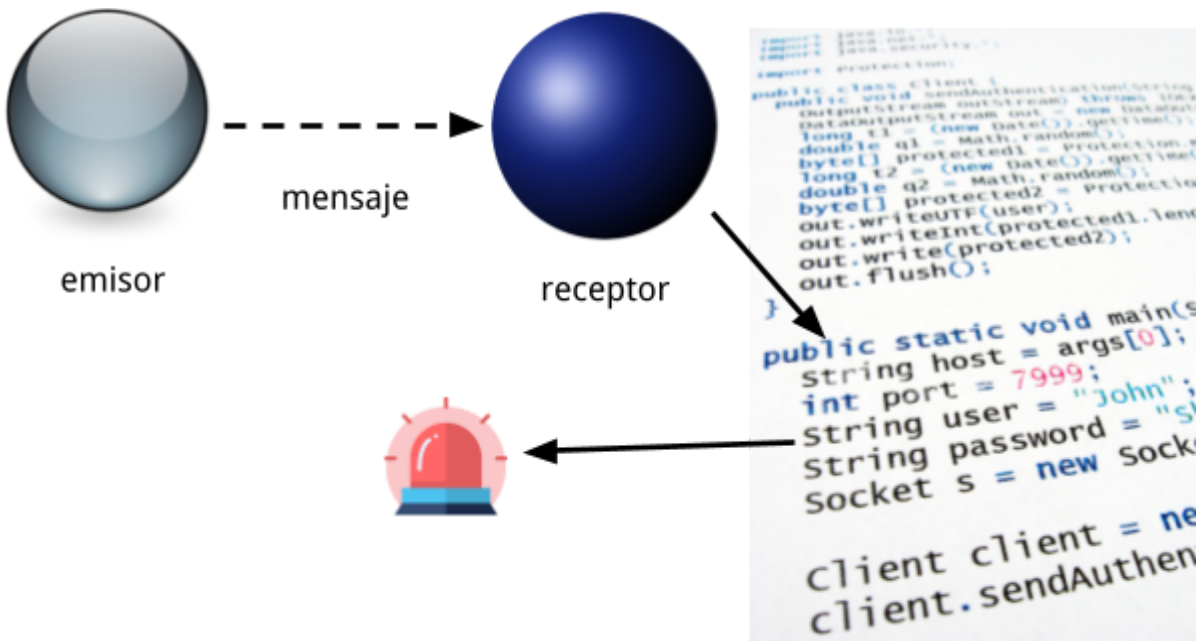
11 Resumen

1 Errores y excepciones

Recordemos la definición de sistema dentro del paradigma de objetos

Sistema = conjunto de objetos que colaboran para un objetivo común.

Entonces precisamos que haya comunicación entre los objetos. Cuando un objeto le envía un mensaje a otro,



el emisor espera que el receptor pueda realizar esa tarea. Ahora ¿qué sucede si falla algo en el método x?

Una ***excepción*** es un evento que altera el flujo normal de mensajes entre objetos.

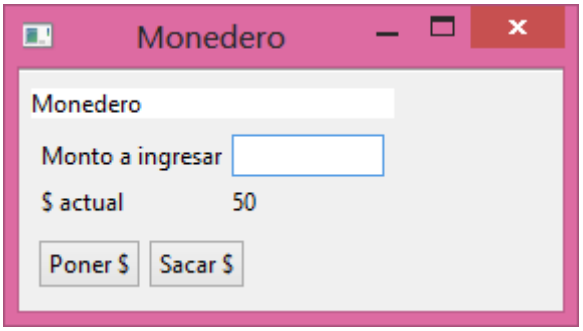
Las excepciones tienen orígenes diferentes: desde errores serios de hardware, como la ruptura del disco rígido, hasta errores simples de programación, como tratar de acceder a un elemento fuera de los límites de un vector, dividir por cero o enviar un mensaje a una referencia que no apunta a ningún objeto.

2 Breve ejemplo: modelar un monedero

Se quiere modelar un monedero al cual se le puedan realizar dos operaciones:

- Poner plata
- Sacar plata

Tenemos una interfaz de usuario que tiene el siguiente formato:



En el campo "$ actual" figurará la cantidad de plata que tiene el monedero. Inicialmente el monedero comienza con 500 (quinientos) pesos. El sistema debe validar lo siguiente:

- para sacar o poner plata en el monedero, el monto debe ser positivo, mayor a cero
- no se puede sacar más plata de lo que tiene el monedero

3 Nuestro objetivo

- entender cómo hay que manejar errores en el dominio
- discriminar el tratamiento de errores de usuario y los de sistema

4 Solución en Wollok

Modelamos en Wollok al objeto de dominio monedero[1]

```
object monedero {
	var plata = 500

	method plata() = plata

	method poner(cantidad) { plata = plata + cantidad }

	method sacar(cantidad) { plata = plata - cantidad }
}
```

Nada extraño, pero todavía no contemplamos las validaciones que el usuario nos pidió. Tenemos que definir qué vamos a hacer si la cantidad es un número negativo...

[1] El ejemplo completo se puede descargar en https://github.com/wollok/excepciones-monedero

5 Alternativas para modelar una excepción

5.1 Mirar a un costado

La peor decisión que podemos tomar es ignorar un valor de entrada que no satisface las condiciones que pide el negocio[2]:

```
method poner(cantidad) {
      if (cantidad > 0) {
            plata += cantidad
      }
}
```

¿Por qué decimos que se ignora? Porque el resultado es desconcertante para quien envía un mensaje

```
>>> monedero.poner(-2)
>>>
```

No da error, pero tampoco se produce efecto colateral en el monedero, que sigue teniendo la misma plata. Y lo que pasa en la interfaz de usuario es que el usuario presiona una y otra vez el botón “Poner $”, sin notar cambios en el monedero.

Esta opción por lo tanto queda descartada de plano.

5.2 Códigos de error

Una segunda opción podría ser utilizar códigos de retorno numéricos:

```
method poner(cantidad) {
      if (cantidad > 0) {
            plata = plata + cantidad
            return 0
      } else {
            return -1
      }
}
```

Entonces el método poner ya no solo tiene efecto colateral, sino un valor de retorno que hay que inspeccionar a su salida.

[2] Con la palabra “negocio” o “dominio” nos referimos a los requerimientos que estableció el usuario en alguno de los relevamientos

Esta estrategia tiene algunas desventajas

- obliga a quien llama a preguntar el valor de retorno antes de continuar con la siguiente línea de una secuencia de instrucciones,
- cuando hay varios mensajes de error posibles las preguntas se encadenan en ifs que dificultan el seguimiento
- los números mágicos -1, -2, no tienen descripciones representativas. Por eso a veces se reemplazan los números con mensajes de error alfanuméricos, aunque sigue siendo difícil rastrear el origen del problema.

Todos estos problemas lo vemos en el siguiente ejemplo, donde queremos verificar si un alumno puede inscribirse a una materia:

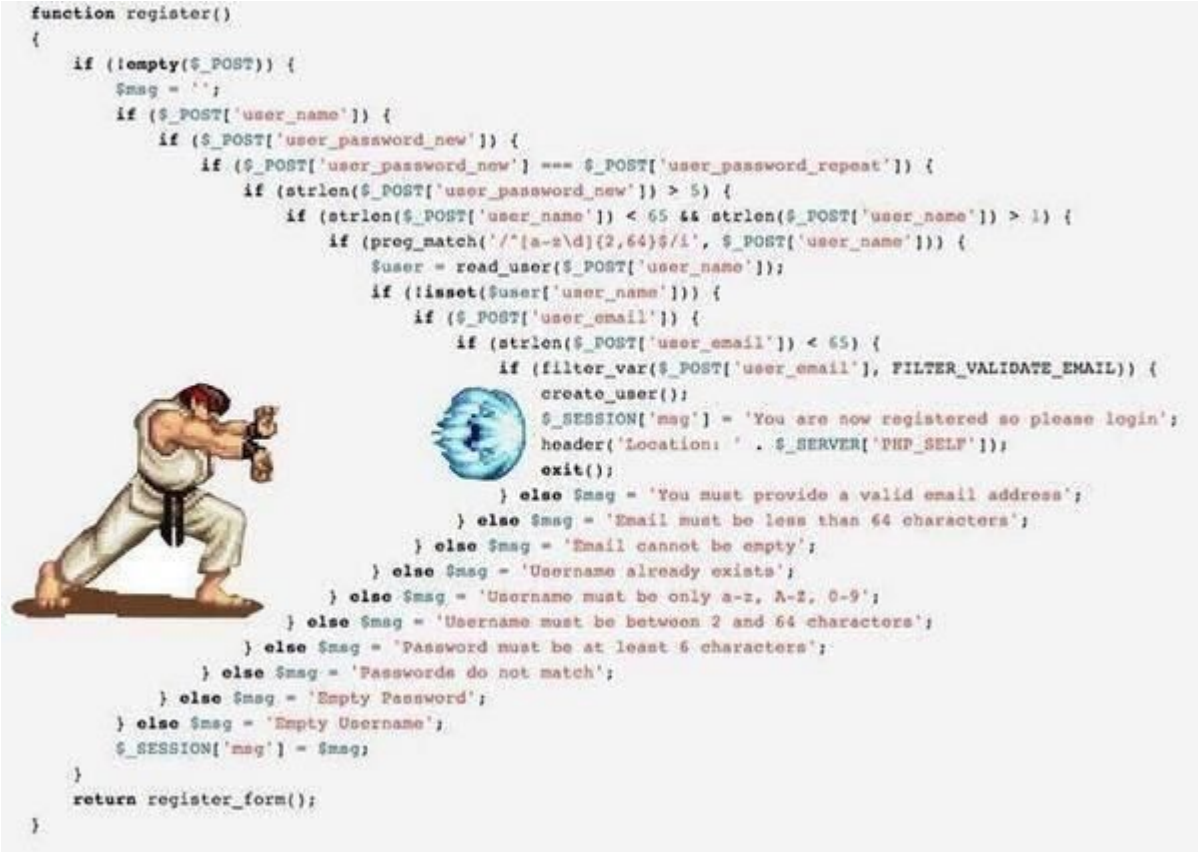
```
method inscribirse(alumno, materia) {
	var codigoError = OK
	const codErrorMateria = materia.validarCupo()
	if (codErrorMateria == 0) {
		const codErrorAlumno = alumno.validarCorrelativas(materia)
		if (codErrorAlumno == 0) {
			... generamos la inscripción ...
		}
		else {
			codigoError = ERROR_ALUMNO_NO_CUMPLE_CORRELATIVAS
		}
	} else {
		codigoError = ERROR_MATERIA_SIN_CUPO
	}
	return codigoError
}
```

La lógica de la inscripción podría complicarse aún más:

- si agregamos una fecha límite para la inscripción
- si aparecen múltiples validaciones sobre la materia o el alumno, por ejemplo que el alumno haya aprobado al menos dos finales el último año

Aquí vemos que no solo se recibe el código de error como valor de retorno del mensaje materia.validarCupo(), sino que este código también debe propagarse hacia quien esté llamando al método inscribirse()[3].

[3] esto produce un grado de acoplamiento mayor entre los objetos involucrados



```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

5.3 Excepciones

La tercera opción es lanzar una excepción por cada condición que salga del flujo normal, en particular

1. al sacar o poner una cantidad menor a cero
2. al sacar más de lo que permita el monedero

Esto lo haremos a continuación.

6 Cómo generar una excepción

Hay dos formas de expresar la generación de una excepción, una con un mensaje que entiende todo objeto y otra instanciando una excepción, utilizando el concepto de clase.[4]

- Enviando un mensaje
  ```
  self.error("Pasó tal cosa")
  ```

- Instanciando una excepción
  ```
  throw new UserException(message = "Pasó tal cosa")
  ```

[4] Continuaremos la explicación y los ejemplos con la opción de instanciar un objeto, que suele ser más utilizada que la otra, pero ambas alternativas son válidas.

Retomando el ejemplo, en el método poner() tenemos que contemplar que la cantidad sea positiva. Si la cantidad es menor a cero, se lanza una excepción cortando la secuencia de envío de mensajes que tenga el método: nunca llegarán a incrementar la cantidad de plata al monedero. Entonces quien envió el mensaje poner() al objeto monedero recibirá la excepción.

```
method poner(cantidad) {
  if (cantidad < 0) {
      throw new UserException(message = "La cantidad debe ser positiva")
      }
      plata += cantidad
}
```

¿Cómo funciona? La clase UserException la definimos utilizando herencia[5]:

```
class UserException inherits Exception { }
```

En el método sacar, una vez más, tenemos que validar la cantidad:

```
method sacar(cantidad) {
   if (cantidad < 0) {
      throw new UserException(message = "La cantidad a retirar debe ser positiva")
   }
   if (cantidad > plata) {
      throw new UserException(message = "Debe retirar menos de " + plata)
   }
   plata = plata - cantidad
}
```

El lector podrá pensar, ¿no se repite la misma validación para poner y sacar? Sí, podemos definir una validación común para ambas operaciones:

```
method poner(cantidad) {
      self.validarMontoPositivo(cantidad)
      plata = plata + cantidad
}
```

[5] Por el momento no nos detendremos a explicar qué implica escribir “inherits”, para el lector curioso recomendamos leer este apunte

```
method sacar(cantidad) {
      self.validarMontoPositivo(cantidad)
      if (cantidad > plata) {
         throw new UserException(message = "Debe retirar menos de "
+ plata)
      }
      plata = plata - cantidad
}

method validarMontoPositivo(cantidad) {
      if (cantidad < 0) {
            throw new UserException(message = "La cantidad debe ser
positiva")
      }
}
```

7 Dónde se atrapa

Por lo general, los objetos de dominio (el cliente, el alumno, el héroe, un ave) no suelen atrapar los errores. Más bien lo único que hacen es lanzar los errores o dejar que ocurran, porque es raro poder hacer algo en el dominio cuando recibimos una excepción.

Volvamos al ejemplo anterior de la inscripción, ahora codificado mediante excepciones:

```
method inscribirse(alumno, materia) {
      materia.validarCupo()
      alumno.validarCorrelativas(materia)
      ... inscripción propiamente dicha ...
}
```

El método quedó bastante más corto, porque si la validación del cupo de la materia falla, no hay mucho que podamos hacer. Tampoco podremos hacer mucho si el alumno no tiene las correlativas al día. Si bien Wollok ofrece la posibilidad de envolver el código que puede fallar en un bloque try/catch, de manera de “atrapar” la excepción y poder definir comportamiento por el flujo alternativo...

```
method inscribirse(alumno, materia) {
      try {
            materia.validarCupo()
            alumno.validarCorrelativas(materia)
            ... inscripción propiamente dicha ...
      } catch e : UserException {
```

```
            ¿¿¿¿¿ Qué hacer ?????
        }
}
```

... no sabemos muy bien qué hacer en el bloque catch. Por eso, lo mejor es simplemente dejar que la excepción que se dispare en materia o alumno siga su curso hacia quien nos mandó el mensaje inscribirse().

```
method inscribirse(alumno, materia) {
        materia.validarCupo()
        alumno.validarCorrelativas(materia)
        ... inscripción propiamente dicha ...
}
```

¿Quién puede atrapar **realmente** una excepción?

- un test unitario[6]
- la interfaz de usuario, que está fuera del alcance del presente apunte

Ellos deben ser los encargados de atrapar la excepción antes de que la aplicación falle y se cierre.

8 Excepciones de usuario y de programa

En base al cliente que va a recibir la excepción, podemos clasificar a las excepciones en dos tipos diferentes:

- **Excepciones de usuario o de dominio:** ocurren en el uso de la aplicación y son entendibles para el usuario final ("no hay saldo en la cuenta corriente", "no hay stock del producto a facturar", "no hay precio del producto a facturar", etc.)
- **Excepciones de programa:** se producen cuando se ejecuta código de la aplicación y las puede analizar un especialista técnico ("falló el acceso al motor de la base de datos", "hubo división por cero", "el objeto no entiende este mensaje", etc.)

La naturaleza de ambos tipos de excepción son diferentes: en general las excepciones de negocio (o de aplicación) requieren que el usuario corrija la información que quiere ingresar al sistema (y valorice el producto, o bien seleccione un producto alternativo para facturar, o trate de sacar plata de otra cuenta bancaria), en tanto que las excepciones de programa requieren una corrección por parte de un usuario técnico (que chequeará la conexión a la base de datos o bien corregirá el

[6] El lector interesado puede estudiar el capítulo de Testing avanzado que cubre los tests que buscan cubrir situaciones excepcionales

código que originó el error).

Por lo tanto, las acciones a tomar cuando armamos cada tipo de excepción son diferentes: en las excepciones de negocio intentamos que el usuario vea una pantalla amigable donde le mostramos el problema que hubo al tratar de completar una acción con un mensaje representativo (e incluso proponiéndole soluciones alternativas para que la tarea se realice), mientras que en las excepciones de programa también mostramos una pantalla amigable al usuario, pero reservamos todos los detalles internos al desarrollador. De esa manera, las excepciones terminan siendo una herramienta más que ayuda a que nuestra aplicación se vuelva más robusta y confiable.

Entonces, para el componente de UI (interfaz de usuario), quizás sea más conveniente no tener que atrapar distintas excepciones, sino diferenciar las que son de negocio y las que no.

9 Tareas para el lector

Hacer los cambios necesarios para incorporar las siguientes validaciones:

- no se pueden hacer más de 3 depósitos el mismo día
- no se puede extraer más de 1.000 pesos diarios

10 Buenas y malas prácticas

Ya sabemos que no hay recetas, pero sí libros de auto-ayuda... aquí van algunas palabras que pueden servir de puntapié inicial para debates a la hora de trabajar con excepciones:

- Sólo las atrapa el que las sabe tratar. Dado que en el contexto de una materia inicial de objetos estamos concentrados en modelar el negocio, es poco frecuente que necesitemos usar el bloque try/catch.
- Las excepciones deberían tratarse en pocos lugares, excederse en el control produce código difícil de leer (y de mantener).
- Utilizar mensajes claros en las excepciones, tanto para los usuarios como para los desarrolladores.
- Representamos como excepciones las condiciones que salen del flujo normal del negocio.
- No debemos nunca atrapar una excepción y no hacer nada. No hacer nada incluye imprimir por consola, poner un comentario TODO o cualquier otra cosa similar, donde **no se avisa del error al componente llamador, con lo cual es probable que este error propague errores posteriores en otros componentes de la aplicación y estemos buscando el error en otro lugar durante un buen tiempo**.
- Este ejemplo muestra cómo el uso de excepciones se desaprovecha para

trabajar con valores de retorno, algo que no es aconsejable:

```
method compraDirecta() {
      try {
            const cliente = new Cliente()
            cliente.comprar(25)
            ...
            return 0
      } catch e : NoTengoUnMangoException    {
            return -1
      }
}
```

¿Cómo debería quedar? Mucho más simple si no tratamos de controlar las cosas y dejamos que fallen:

```
method compraDirecta() {
      const cliente = new Cliente()
      ...
      chiara.comprar(25)            // sabemos que puede fallar
}
```

10.1 To throw or not to throw

En el siguiente ejemplo que modela la contratación de servicios para arreglar una casa (como electricistas, plomeros, albañiles), hay una regla de negocio que pide que validemos que haya suficiente presupuesto para pagarle al contratista. Esto se implementa con una excepción de negocio, y está bien que así sea:

```
method contratarA(contratista) {
      if (contratista.costoArreglo(casa) > self.presupuesto())
            throw new UserException(message = "El presupuesto es
insuficiente")
      ...
```

No es posible contratar a un contratista sin pasar esta validación.
Ahora al querer conocer cuáles fueron los contratistas a los que llamé...

```
method contratistasContratados() {
      if (serviciosContratados.isEmpty()){
            throw new UserException(message = "No contrató servicios
aún")
      }
```

```
	return serviciosContratados.map({ ... })
}
```

Esto es conceptualmente incorrecto:

- si quiero enviar el mensaje *contratistasContratados*, el diseño me obliga a preguntar antes si un cliente contrató o no servicios para evitar la excepción.
- todos los clientes comienzan sin servicios contratados, no es una condición excepcional del negocio.
- ¡Y el map funciona bien cuando no hay elementos en la colección! No necesito en ningún punto el if...

```
>>> [].map({ element => element * 2 })
[]
```

10.2 Consistencia en el valor de retorno de un método

Consideremos el siguiente ejemplo, donde tenemos un taxista Daniel que toma viajes si el destino es Alejandro Korn:

```
object dani {
	method puedeTomar(viaje) {
		if (viaje.destino() == "Korn") return true
		throw new Exception(message = "No puede tomar el viaje")
	}
```

Si la pregunta es "¿puede tomar un viaje?", es inconsistente que el método

- devuelva true si Daniel puede tomar el viaje
- o tire error en caso contrario. Lo que debe hacer es... devolver false si no lo puede tomar.

```
method puedeTomar(viaje) = viaje.destino() == "Korn"
```

Ahora **sí** estamos siendo consistentes, el método devuelve siempre un valor booleano. Y devolvemos una expresión booleana como valor de retorno del método.

Distinto es el comportamiento cuando buscamos agregar un viaje a dani:

```
method tomarViaje(viaje) {
	if (!self.puedeTomar(viaje)) {
		throw new Exception(message = "No puede tomar el viaje")
	}
	viajes.add(viaje)
}
```

Claro, aquí el método tomarViaje no devuelve nada, entonces está bien que

- no retorne nada si funciona ok
- o tire un error en caso de que no se pueda tomar el viaje

10.3 Mejorando los mensajes de error

Ahora Daniel toma viajes

- si son a Alejandro Korn (que es un destino cómodo)
- y si el pasajero tiene más de 50 años

Elaboramos una primera versión

```
method puedeTomar(viaje) = viaje.destino() == "Korn" &&
                           viaje.pasajero().edad() > 50

method agregarViaje(viaje) {
      if (!self.puedeTomar(viaje)) {
            throw new Exception(message = "No puede tomar el viaje")
      }
      viajes.add(viaje)
}
```

¿Cuál es el motivo por el cual no puede tomar un viaje? No lo sabemos. Cuando tenemos que pasar muchas validaciones a la vez, está bueno que el mensaje de la excepción refleje el motivo por el cual la operación no puede satisfacerse.

Entonces

- por un lado trataremos de evitar duplicaciones en las preguntas
- por otro que los mensajes de error sean representativos

```
method destinoComodo(viaje) = viaje.destino() == "Korn"
method edadPasajeroCorrecta(viaje) =
      viaje.pasajero().edad() > EDAD_MINIMA_PASAJERO

method puedeTomar(viaje) = self.destinoComodo(viaje) &&
      self.edadPasajeroCorrecta(viaje)

method agregarViaje(viaje) {
      if (!self.destinoComodo(viaje)) {
            throw new Exception(message = "No puede tomar el viaje
porque el destino " + viaje.destino() + " no es cómodo")
      }
      if (!self.edadPasajeroCorrecta(viaje)) {
            throw new Exception(message = "El pasajero debe tener " +
EDAD_MINIMA_PASAJERO  + " años o más")
```

```
        }
        viajes.add(viaje)
}
```

De todas maneras, esta técnica solo necesitamos implementarla cuando queremos tener dos métodos: la pregunta puedeXXX, sin efecto que debe devolver un valor booleano y la acción hacerXXX, que debe producir efecto o tirar excepción.

11 Resumen

Las excepciones nos sirven para modelar condiciones que salen del flujo normal del negocio. Para los objetos de dominio las formas correctas de usar este concepto es muy simple:

- en el caso de los errores de programa, no se pueden prever, así que no debemos hacer nada
- en el caso de los errores de usuario, debemos detectarlos y alertar al usuario lo más tempranamente posible, tratando de dar un mensaje de las acciones que debe realizar para poder completar la acción que desea.

Por último, hay mecanismos para atrapar las excepciones y volver al sistema a un estado consistente, aunque por lo general son raros los casos en el que un objeto de dominio puede hacer algo al respecto.

Paradigma Orientado a Objetos

Módulo 11: Testeo unitario automatizado avanzado

por Lucas Spigariol
Fernando Dodino
y colaboración de
Matías Freyre
Versión 2.2
Septiembre 2018

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Introducción

2 Código compartido entre tests
- 2.1 Ejemplo y solución
- 2.2 Inicialización del describe
- 2.3 Initialize
- 2.4 Métodos auxiliares
- 2.5 Tests completos

3 Testeo de errores
- 3.1 Esperando un error

4 ¿Qué probar? ¿Cuánto probar?
- 4.1 Code coverage
- 4.2 Clases de equivalencia en los tests
- 4.3 ¿Pruebas unitarias?

5 Resumen

1 Introducción

Hasta el momento hemos conocido el testeo unitario automatizado que tiene Wollok, que requiere definir **casos de prueba**, e implementarlos mediante **tests** que se agrupan en construcciones llamadas **describe**. La ejecución de los tests tiene tres resultados posibles: el test puede satisfacer las comprobaciones, no satisfacerlas o arrojar un error antes de poder determinar un resultado.

Continuaremos la explicación con conceptos que permitirán mejorar la calidad de nuestras pruebas automatizadas.

2 Código compartido entre tests

Cuando la complejidad de los casos de prueba crece, aparece la necesidad de compartir estado procurando mantener el carácter unitario de cada test. Si bien nunca hay dos tests idénticos, es frecuente que haya diferentes tests que guardan grandes coincidencias con otros, que necesiten realizar acciones en común, o que prueban diferentes aspectos sobre los mismos conjuntos de datos. Para lograr armonizar estas situaciones coherentemente con los criterios clásicos de la programación en cuanto a no repetir lógica, *Wollok* provee algunas herramientas adicionales para organizar los tests.

2.1 Ejemplo y solución

Para que se vea mejor la utilidad de las herramientas que vamos a presentar, reformulamos y ampliamos el ejemplo de pepita.

El planteo del problema ahora agrega:

> *"En vez de volar una cierta cantidad de kilómetros, vuela a diferentes ciudades y cada una de ellas implica un consumo de energía diferente. Pepita debe poder cumplir su deseo, que consiste en volar a la más reciente de sus ciudades favoritas. Una vez cumplido su deseo, esa ciudad deja de estar entre sus favoritas. También se quiere saber si una determinada ciudad es favorita para pepita"*

La solución ahora incluye la clase ciudad (muy trivial) y cambios en el objeto pepita:

```
class Ciudad {
	const consumo

	method consumo() {
		return consumo  // o consumo podría ser una propiedad...
	}
}

object pepita {
	var energia = 100
	const ciudadesFavoritas = [ ] // Nuevo

	method energia() { return energia }

	method esFuerte() { return energia > 50 }

	method volar(ciudad) { // Modificado para recibir una ciudad
		energia = energia - ( ciudad.consumo() + 10 )
	}

	method comer(gramos) {
		energia = energia + 4 * gramos
	}

	method cumplirDeseo() { // Nuevo
		const destino = ciudadesFavoritas.last()
		self.volar(destino)
		ciudadesFavoritas.remove(destino)
	}

	method agregarCiudad(ciudad) { // Nuevo
		ciudadesFavoritas.add(ciudad)
	}

	method esFavorita(ciudad) { // Nuevo
		return ciudadesFavoritas.contains(ciudad)
	}
}
```

Para probar la nueva versión, se mantienen algunos casos de prueba, se reformulan otros y se agregan nuevos:

Acciones	Verificación
ninguna	pepita tiene energía 100
ninguna	pepita es fuerte
que pepita vuele a San Martín (consume 5 unidades extra)	pepita queda con energía 85
que pepita coma 120 gramos	pepita queda con energía 580
que pepita coma 0 gramos	pepita queda con energía 100
que pepita vuele a una ciudad que consume 0 unidades extra	pepita queda con energía 90
que pepita vuele a San Martín (consume 5 unidades extra)	pepita sigue siendo fuerte
que pepita vuele a Campana (consume 60 unidades extra)	pepita no es fuerte

Planteamos un escenario con dos ciudades favoritas, Campana y San Martín. En algún momento pepita puede incorporar como nueva ciudad a Quilmes, que consume 1 unidad de energía.

Que pepita cumpla su deseo (volar a San Martín)	pepita queda con energía 85
Que pepita cumpla su deseo, luego conozca Quilmes y vuelva a cumplir su deseo dos veces más (volar a San Martín, luego a Quilmes por último a Campana)	pepita queda con energía 4

También puede ser conveniente verificar qué pasa con las ciudades favoritas de pepita.

Ninguna acción previa	pepita tiene a San Martín como ciudad favorita
Que pepita cumpla su deseo (volar a San Martín)	pepita no tiene a San Martín como ciudad favorita. pepita tiene a Campana como ciudad favorita.

Que pepita cumpla su deseo, luego conozca Quilmes y vuelva a cumplir su sueño dos veces más (volar a San Martín, luego a Quilmes y por último a Campana)	pepita no tiene a Campana como ciudad favorita.

Probablemente podrían hacerse otras pruebas diferentes o plantearlas de otra manera, pero para lo que queremos mostrar es suficiente.

2.2 Inicialización del describe

A continuación de la especificación del **describe** con su correspondiente nombre, y previo a la enumeración de los tests que abarca, se pueden declarar variables y constantes. Al igual que cuando se definen en objetos o clases, pueden inicializarse ya sea con valores puntuales, instanciando clases existentes o con el valor de retorno de cualquier mensaje. El alcance de estas variables y constantes son todos los tests del describe (y también *initialize* y métodos). Se permite definir constantes sin un valor inicial, siempre que sean inicializadas en el initialize.

Importante: Tener en cuenta que el estado de estas variables (y constantes) se reinicia con el valor especificado previo a ejecutar cada uno de los tests, al igual que todos los objetos existentes en el ambiente.

2.3 Initialize

Cuando se requiere realizar previo a cada test otras acciones de configuración de la situación inicial más complejas, para las cuales no es suficiente la inicialización de variables, se puede definir un método initialize que no espera parámetros. Se escribe como cualquier otro método. Escribimos la firma **method initialize()** y luego entre { } se detalla la secuencia de mensajes que se necesite. Se especifica luego de la declaración e inicialización de variables y previo a los tests.

2.4 Métodos auxiliares

Para ciertas acciones que se repiten entre algunos tests pero no son comunes a todos, se pueden definir métodos auxiliares, de igual manera que los métodos de cualquier objeto. Su nombre es cualquier identificador válido, puede tener parámetros,

retornar valores, y ser invocado desde cualquier lugar del describe, -un test, otro método, la sección de inicialización-, utilizando self.

2.5 Tests completos

Considerando todos los elementos mencionados, el orden de ejecución de cada uno de los tests del describe será:

- Reseteo del ambiente
- Inicialización de las variables de la descripción
- Initialize
- Test propiamente dicho

En el ejemplo de pepita, los combinamos de la siguiente manera:

```
import pepita.*

describe "Tests de Pepita" {
      // son objetos que se usan en todos los tests
      const campana = new Ciudad(consumo = 60)
      const sanMartin = new Ciudad(consumo = 5)

      // todos los tests parten con pepita teniendo como
      // favoritas a estas dos ciudades
      method initialize() {
            pepita.agregarCiudad(campana)
            pepita.agregarCiudad(sanMartin)
      }

      // hay tests que prueban cosas diferentes a partir de la misma
      // situación, por lo que una opción es definir un método
      method viajeLoco() {
            // dado que quilmes solo se utiliza aquí,
            // es conveniente no definir la referencia en el describe
            // sino que quede dentro del test. Además, dentro del mismo
            // test podemos cómo se define o qué representa quilmes,
            // lo que ayuda a entender mejor lo que estamos probando.
            const quilmes = new Ciudad(consumo = 1)
            pepita.cumplirDeseo()
            pepita.agregarCiudad(quilmes)
            pepita.cumplirDeseo()
            pepita.cumplirDeseo()
      }
```

```
    test "pepita comienza con 100 unidades de energía" {
        assert.equals(100, pepita.energia()) // igual al anterior
    }

    test "pepita comienza siendo fuerte" {
        assert.that(pepita.esFuerte()) // igual al anterior
    }

    test "pepita vuela a una ciudad con consumo mayor a cero y ese
consumo le quita energia" {
        pepita.volar(sanMartin) // Modificado, ahora con una ciudad
        assert.equals(85, pepita.energia())
    }

    test "cuando pepita come su energia baja" {
        pepita.comer(120) // igual al anterior
        assert.equals(580, pepita.energia())
    }

    test "pepita vuela a una ciudad que no tiene consumo, la energía
baja un valor constante" {
        const ciudadSinConsumo = new Ciudad(consumo = 0)

        pepita.volar(ciudadSinConsumo)
        // Modificado, ahora con una ciudad

        assert.equals(90, pepita.energia())
    }

    test "pepita come 0 gramos y la energía se mantiene" {
        pepita.comer(0)
        assert.equals(100, pepita.energia())
    }

    test "pepita vuela a una ciudad que le consume mucho, ya no es
fuerte" {
        pepita.volar(campana) // Modificado, ahora con una ciudad
        assert.notThat(pepita.esFuerte())
    }

    test "pepita cumple su deseo, al volar queda con menos energia" {
        pepita.cumplirDeseo() // Nuevo
        assert.equals(85, pepita.energia())
    }
```

```
	test "pepita hace su viaje loco y queda con energia 4" {
		self.viajeLoco() // Nuevo. Usa método del describe
		assert.equals(4, pepita.energia())
	}

	test "al inicio pepita tiene a San Martin por ciudad favorita" {
		assert.that(pepita.esFavorita(sanMartin)) // Nuevo
	}

	test "pepita cumple su deseo y cambia su ciudad favorita" {
		pepita.cumplirDeseo() // Nuevo
		assert.notThat(pepita.esFavorita(sanMartin))
		assert.that(pepita.esFavorita(campana))
	}

	test "pepita hace su viaje loco y no tiene a campana como
favorita" {
		self.viajeLoco() // Nuevo. Usa método del describe
		assert.notThat(pepita.esFavorita(campana))
	}
}
```

Algunas cosas que podríamos mejorar sobre nuestra versión inicial:

- hay tests que prueban cómo queda la energía al comer o volar y otros que verifican el comportamiento asociado a las ciudades favoritas, quizás no todos necesitan compartir la misma inicialización. Podríamos separarlos en dos describes distintos, uno que tenga el método initialize (para las ciudades) y otro que no (para el chequeo de la energía al comer o volar)
- podrán preguntarnos qué sucede si 1) decido separar los casos de prueba en varios describes, pero 2) no puedo reutilizar el mismo método entre diferentes describes (porque no se pueden referenciar los describes). Una forma simple de solucionarlo es mediante un objeto que se encargue de crear los objetos que necesito, una ciudad con poco o mucho consumo, un viaje loco, etc.: en general cuando necesitemos reutilizar mecanismos para instanciar objetos, lo mejor es poner esa responsabilidad en un objeto aparte.

Les dejamos la solución con dos describes dentro del mismo archivo, lo que permite que no se mezclen dos casos de prueba diferentes y simplifica el armado del juego de datos que algunos tests necesitan (así además no penalizamos a todos los tests para correr una inicialización si no se necesita):

```
describe "Tests de Pepita - ciudades" {
```

```
    const campana = new Ciudad(consumo = 60)
    const sanMartin = new Ciudad(consumo = 5)

    method initialize() {
        pepita.agregarCiudad(campana)
        pepita.agregarCiudad(sanMartin)
    }

    method realizarViajeLoco() {
        const quilmes = new Ciudad(consumo = 1)
        pepita.cumplirDeseo()
        pepita.agregarCiudad(quilmes)
        pepita.cumplirDeseo()
        pepita.cumplirDeseo()
    }

    test "al inicio pepita tiene a San Martin por ciudad favorita" {
        assert.that(pepita.esFavorita(sanMartin))
    }

    test "pepita cumple su deseo y cambia su ciudad favorita" {
        pepita.cumplirDeseo()
        assert.notThat(pepita.esFavorita(sanMartin))
        assert.that(pepita.esFavorita(campana))
    }

    test "pepita hace su viaje loco y no tiene a campana como
favorita" {
        self.realizarViajeLoco()
        assert.notThat(pepita.esFavorita(campana))
    }

    test "pepita cumple su deseo, al volar queda con menos energía" {
        pepita.cumplirDeseo()

        assert.equals(85, pepita.energia())
    }

    test "pepita hace su viaje loco y queda con una cierta energía" {
```

```
            self.realizarViajeLoco()
            assert.equals(4, pepita.energia())
      }
}

describe "Tests de Pepita - energía" {
      test "pepita comienza con 100 unidades de energía" {
            assert.equals(100, pepita.energia()) // igual al anterior
      }

      test "pepita comienza siendo fuerte" {
            assert.that(pepita.esFuerte())
      }

      test "pepita vuela a una ciudad con consumo mayor a cero y ese
consumo le quita energia" {
            const sanMartin = new Ciudad(consumo = 5)
            pepita.volar(sanMartin)
            assert.equals(85, pepita.energia())
      }

      test "cuando pepita come su energia baja" {
            pepita.comer(120)
            assert.equals(580, pepita.energia())
      }

      test "pepita vuela a una ciudad que no tiene consumo, la energía
baja un valor constante" {
            const ciudadSinConsumo = new Ciudad(consumo = 0)

            pepita.volar(ciudadSinConsumo)
            assert.equals(90, pepita.energia())
      }

      test "pepita come 0 gramos y la energía se mantiene" {
            pepita.comer(0)
            assert.equals(100, pepita.energia())
      }
```

```
    test "pepita vuela a una ciudad que le consume mucho, ya no es fuerte" {
        const campana = new Ciudad(consumo = 60)
        pepita.volar(campana)
        assert.notThat(pepita.esFuerte())
    }

}
```

3 Testeo de errores

En el capítulo de manejo de errores, hemos presentado el ejercicio del monedero, donde podemos

- poner plata: un valor positivo
- sacar plata: un valor positivo sin que exceda lo que hay en el monedero

Dado un monedero que inicialmente tenga $ 500, ¿qué pruebas podemos hacer?

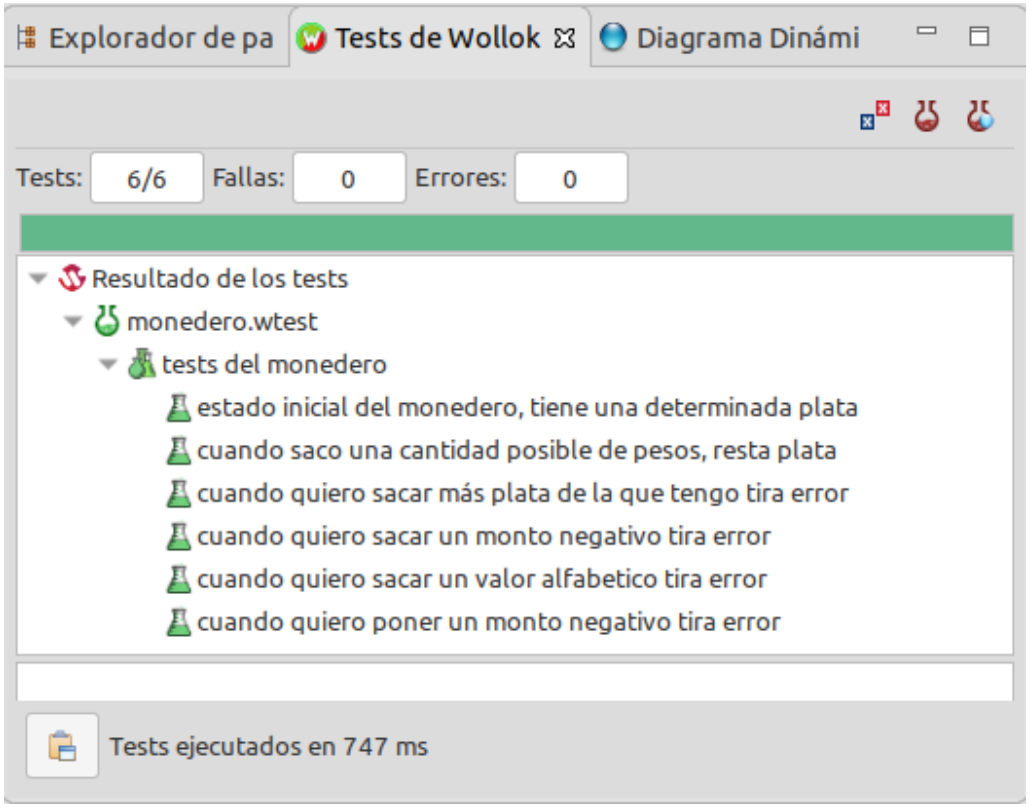
Precondiciones	Verificación
ninguna	El monedero tiene $ 500
sacamos $ 15 del monedero	Al monedero le quedan $ 485
quiero sacar $ 1000 del monedero (hay $ 500)	Tiene que dar error con mensaje “Debe retirar menos de 500”
sacar $ -20 del monedero	Tiene que dar error con mensaje “La cantidad debe ser positiva”
sacar “A” del monedero	Tiene que dar error
poner $ -20 en el monedero	Tiene que dar error con un mensaje “La cantidad a retirar debe ser positiva”

El lector podrá notar que en los últimos 4 casos lo que se espera es lo mismo: un error. Vemos cómo se implementa[1]:

[1] Esta variante permite mostrar que el uso de la construcción *describe* es opcional. Si no se necesita compartir estado entre los tests, se pueden definir tests aislados en un archivo *.wtest*

```
import monedero.*

test "estado inicial del monedero tiene 500 $" {
      assert.equals(500, monedero.plata())
}

test "cuando saco una cantidad posible de pesos, resta plata" {
      monedero.sacar(15)
      assert.equals(485, monedero.plata())
}

test "cuando quiero sacar más plata de la que tengo tira error" {
      assert.throwsExceptionWithMessage("Debe retirar menos de 500",
      { monedero.sacar(1000)})
}

test "cuando quiero sacar un monto negativo tira error" {
      assert.throwsExceptionLike(
         new UserException(message = "La cantidad a retirar debe ser
positiva"),
         { monedero.sacar(-20) }
      )
}

test "cuando quiero sacar un valor alfabetico tira error" {
      assert.throwsException({ monedero.sacar("A") })
}

test "cuando quiero poner un monto negativo tira error" {
      assert.throwsExceptionWithMessage("La cantidad a retirar debe ser
positiva", { monedero.poner(-20) })
}
```

Ejecutamos los tests y vemos que todo está ok:



3.1 Esperando un error

Repasemos el test que saca un monto negativo. Allí el test está escrito dentro de un bloque de código que le pasamos al objeto assert enviando el mensaje `throwsExceptionLike`. Nos podemos imaginar lo que hace: esperamos que dentro de ese código se produzca una excepción.

Entonces, si el bloque de código tira una excepción, eso significa que está funcionando de acuerdo a lo "esperado".

Si no recibimos una excepción, esto es una mala señal. Por ese motivo si el bloque de código no falla, el test está mal.
Así es, el test falla.

El objeto assert nos permite hacer diferentes chequeos por error:

- **`throwsException(block)`:** es el más básico, en el test esperamos un error (no importa cuál).
- **`throwsExceptionWithMessage(errorMessage, block)`:** esperamos un error con un mensaje específico (si no coincide el mensaje el test falla)

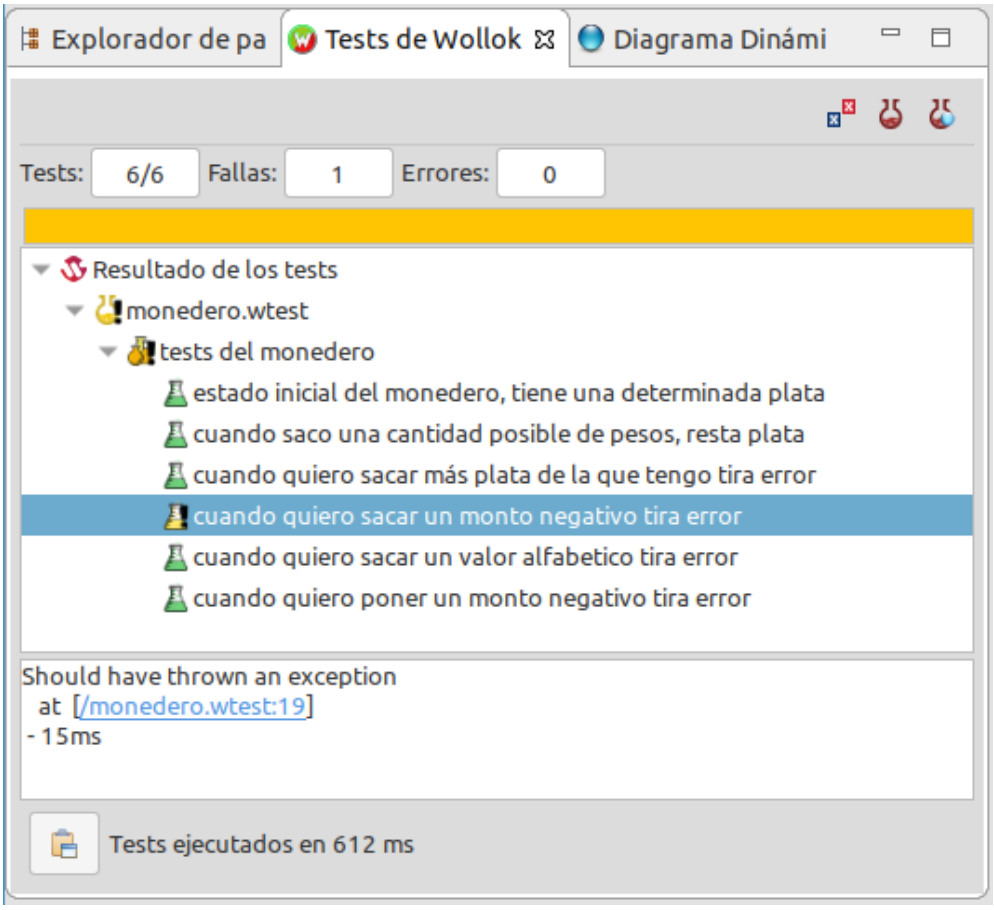
- **`throwsExceptionLike(exceptionExpected, block)`:** es la opción más restrictiva, ya que tienen que coincidir
  - el tipo de excepción (UserException en este caso)
  - y el mensaje de error

Para comprobar que el test está funcionando correctamente, vamos a modificar el monedero, comentando la validación de los importes negativos:

```
object monedero {
	...

	method sacar(cantidad) {
//		self.validarMonto(cantidad)
		if (cantidad > plata) {
			throw new UserException(message = "Debe retirar menos
de " + plata)
		}
		plata = plata - cantidad
	}
```

Al ejecutar el test, se produce un error, porque monedero.sacar(-20) no cortó la secuencia, y la siguiente línea dispara un error en el test:

```
test "cuando quiero sacar un monto negativo tira error" {

	assert.throwsExceptionLike(
		new UserException("La cantidad a retirar debe ser
	positiva"),
		{ monedero.sacar(-20) }) // no tira error!
}
```



En cambio, cuando `monedero.sacar(-20)` dispare la excepción, el assert.throwsExceptionLike internamente atrapará el error y no hará nada, porque es lo que nosotros queríamos que pasara:

```
test "cuando quiero sacar un monto negativo tira error" {
	assert.throwsExceptionLike(
		new UserException(message = "La cantidad a retirar debe ser
	positiva"),
		{ monedero.sacar(-20) }) // tira error y se atrapa en el
	assert
}
```

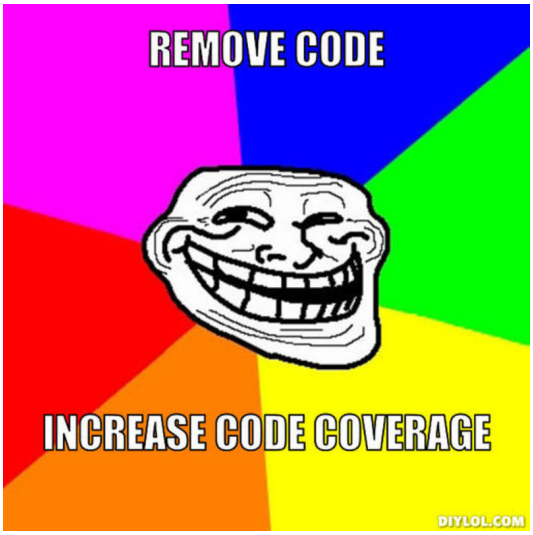
4 ¿Qué probar? ¿Cuánto probar?

Así como vamos tomando conciencia de la importancia de tener tests para probar nuestro código, también vamos notando que lleva tiempo y esfuerzo. Por lo tanto empieza a sobrevolar la pregunta ¿vale la pena probar absolutamente todo?

4.1 Code coverage

¿Qué porcentaje de nuestro código estamos testeando? En estos casos, el 100%, porque se trata de ejemplos breves. Es una condición que rara vez encontraremos en las aplicaciones productivas.

Acabamos de presentar el *code coverage* que es una métrica utilizada para medir qué nivel de cobertura de código tienen nuestros tests[2]. Es un número que va de 0 a 100, donde 1 es muy poco y 100 es *demasiado*. ¿Por qué demasiado? Porque a medida que nos acercamos al 100% nuestros tests pierden valor (por ejemplo porque nos obsesionamos con probar getters, setters, métodos con lógica trivial, etc.) Además, conforme aparecen nuevos métodos de negocio no siempre podemos contemplar y generar los casos de prueba para todos ellos.



No hay un número mágico, es una cuestión de equilibrio y razonabilidad en función de las características del desarrollo y del equipo que lo lleva adelante. Si bien se busca que sea un porcentaje elevado, llega un punto donde dedicar demasiado esfuerzo a incrementar levemente el porcentaje de cobertura se vuelve contraproducente.

4.2 Clases de equivalencia en los tests

Retomando lo que decíamos con el ejemplo de pepita, teniendo un test que le pide a pepita que vuele 5 kilómetros. ¿Tiene sentido hacer otro test que le pida volar 10 kilómetros? ¿y luego otro que le pida a pepita que vuele 6 kilómetros?

No en tanto sabemos que pepita queda con una energía positiva. En cambio, al pedirle a pepita que vuele 91 kilómetros, su energía quedaría negativa: 100 - (91 + 10) = -1. Así como entonces nos pareció que volar 0 km era un caso singular que ameritaba probarse, con más razón éste que deja a pepita en una situación extraña.

Lo que sucede entonces es que los casos de prueba forman **clases de equivalencia** [3], lo que ayuda a definir pruebas representativas que sean de nuestro interés.

[2] Para más información recomendamos leer http://c2.com/cgi/wiki?CodeCoverage
[3] para más información véase https://es.wikipedia.org/wiki/Relaci%C3%B3n_de_equivalencia

El **análisis de los valores límite** de un método nos permite diferenciar dos clases de equivalencia diferentes. En el ejemplo de pepita, pedimos que

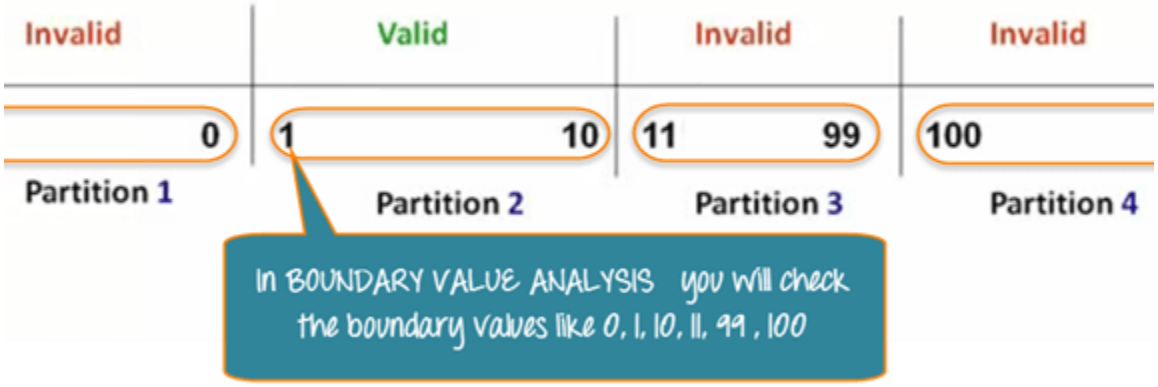
- coma una distancia valor mayor a cero
- y que coma una cantidad nula

Para el método esFuerte(),

```
method esFuerte() { return energia > 50 }
```

una buena prueba consiste en hacer la prueba considerando 50 como valor límite para la energía. Tendríamos entonces

- una clase de equivalencia a “las pepitas que tienen 50 unidades de energía” (o menos)
- otra clase de equivalencia a “las pepitas que tienen 51 unidades de energía” (o más)



Por último, los **casos de error** también constituyen una importante clase de equivalencia en los tests, como hemos visto en el caso del monedero.

4.3 ¿Pruebas unitarias?

La prueba se considera unitaria en tanto y en cuanto estén probando una unidad funcional. Podríamos asociar la prueba unitaria a la prueba de una única clase/objeto, pero los ejemplos expuestos hasta el momento no son generalizables: ni pepita, ni un empleado ni el monedero tenían dependencias con otros objetos de nuestro dominio. ¿Qué pasaría si estuviéramos probando un cliente que tiene un conjunto de facturas, conformadas por una serie de ítems, cada una con una cantidad y un producto?

Cuando tenemos dependencias entre objetos (no entre tests, que siguen siendo independientes), tenemos que elegir una estrategia que nos indique hasta dónde

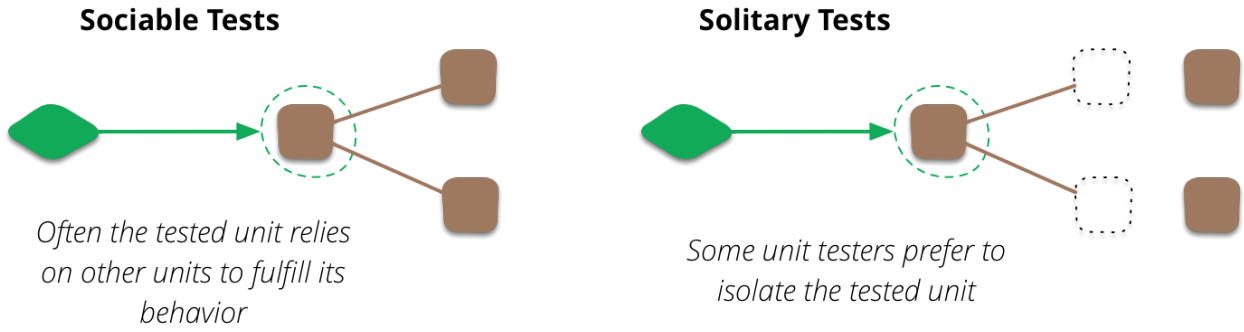
estaremos testeando. Por ejemplo podríamos separar al cliente, a la factura y al producto como tres unidades que cumplen funciones en forma independiente. Más allá de esta elección, no resulta sencillo armar un juego de datos para probar un cliente: ¿puedo encarar los tests de cliente si todavía no construí la factura? ¿puedo hacer pruebas sobre una factura si no terminé de definir el comportamiento del producto?

De la misma manera,

- cuando tenemos efecto colateral (como en el caso de las bases de datos, o al enviar un mail)
- cuando hay un proceso no-determinístico (como un error de red, un momento del día, o una temperatura dada)
- cuando la construcción del juego de datos es compleja
- cuando hay objetos que son poco performantes

nos conviene utilizar objetos que sean polimórficos con los que usaremos en la aplicación "real", pero que tienen un comportamiento simple, rápido, determinístico y fácilmente configurable.

Martin Fowler diferencia los tests en "sociales" y "solitarios"[4]. Mientras que en los primeros se prueba toda la unidad y sus dependencias (en el caso del cliente se estaría utilizando la factura y los productos, y deberíamos esperar a que estén codificadas dichas clases), en los segundos se utilizan implementaciones "de resguardo", lo que facilita mantener la unitariedad del test del objeto cliente.



Ej: Tenemos una clase cliente, y una implementación de prueba de la clase Factura

```
class Cliente {
      const facturas = []

      method agregarFactura(factura) {
```

[4] http://martinfowler.com/bliki/UnitTest.html

```
            facturas.add(factura)
      }

      method saldo() {
            return facturas.sum { factura => factura.saldo() }
      }
}

class FacturaPrueba {
      method saldo() = 100
}
```

¿Para qué sirve generar una factura de prueba que devuelva 100 cuando le pregunto el saldo? Para poder hacer el test del cliente. Si el cliente tiene una factura de prueba, debería asumir que su saldo es 100:

```
import cliente.*

describe "Test de clientes" {
      const cliente = new Cliente()
      const facturaPrueba = new FacturaPrueba()

      method initialize() {
            cliente.agregarFactura(facturaPrueba)
      }

      test "el saldo del cliente de una sola factura de prueba es el
saldo de esa factura" {
            assert.equals(100, cliente.saldo())
      }

      ...
}
```

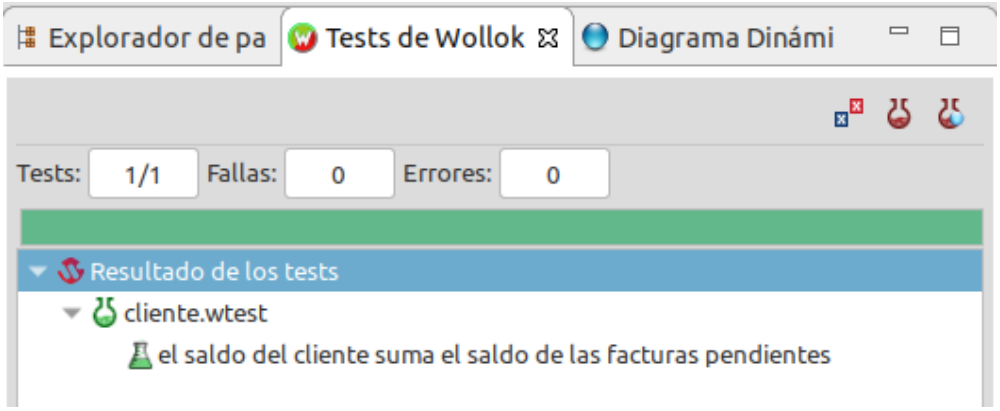
De la misma manera podríamos haber definido un FacturaPrueba cuyo saldo sea configurable en la instanciación:

```
class FacturaPrueba {
      const property saldo
      method estaPendiente() = true // o self.saldo() > 0
}
```

Esto nos permitiría tener varias facturas con montos diferentes, y cambiar el describe:

```
import cliente.*

describe "Test de clientes" {
   const cliente = new Cliente()
   const facturaPrueba1 = new FacturaPrueba(saldo = 150)
   const facturaPrueba2 = new FacturaPrueba(saldo = 250)

   method initialize() {
      cliente.agregarFactura(facturaPrueba1)
      cliente.agregarFactura(facturaPrueba2)
   }

   test "el saldo del cliente suma el saldo de las facturas
pendientes" {
      assert.equals(400, cliente.saldo())
   }
}
```

Corremos el test, y efectivamente funciona, aun cuando no tenemos la implementación real de la factura[5]:



Otra forma de definir objetos de resguardo es utilizando objetos anónimos, como hemos contado previamente:

```
describe "Test de clientes" {
      const cliente = new Cliente()
      const facturaPrueba1 = self.facturaPrueba(saldo = 150)
      const facturaPrueba2 = self.facturaPrueba(saldo = 250)

      method facturaPrueba(saldo) = object {
            method estaPendiente() = true
            method saldo() = saldo
```

[5] Para una visión más profunda de estos conceptos recomendamos el siguiente apunte de Testing

```
    }

    method initialize() ... (idem)
}
```

En este caso nuestras facturas de prueba cumplen el rol de objetos impostores o *stubs*. Su sentido es proveer una implementación de resguardo para poder testear un cliente.

5 Resumen

En este capítulo profundizamos el mecanismo de testeo unitario automatizado que provee Wollok. En particular, el describe no solo agrupa los casos de prueba, sino que permite definir variables que pueden ser utilizados por todos los tests. Esas variables referencian a objetos que pueden inicializarse en la misma definición, en un **initialize** o utilizando un **método** dentro del mismo describe. También vimos cómo implementar pruebas cuando el comportamiento esperado es un error.

El diseño de los casos de prueba es una tarea que está lejos de ser repetitiva y mecánica: no hay una metodología que nos asegure que nuestros tests son buenos, aunque algunas métricas como el **code coverage** pueden ayudar (siempre que tengamos en cuenta que es una medida y no un fin en sí mismo).

Por último, para mantener la unitariedad de los tests podemos usar objetos de resguardo (a veces llamados **stubs** o **mocks**) y facilitar así el testeo de una unidad funcional.

El apunte concluye aquí, no así el aprendizaje de la actividad de testing que es mucho más abarcativa, y que dejamos a cargo del lector.

Paradigma Orientado a Objetos

Módulo 12: Colecciones con clases. Bloques de código.

por Fernando Dodino
revisión por Lucas Spigariol
Versión 3.2
Julio 2019

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Colecciones revisited
- 1.1 Interfaz de una colección

2 Misiones de un héroe, parte 2
- 2.1 Definiciones: de objetos a clases
- 2.2 Diagrama de clases de la solución
- 2.3 Referencias: algunas pruebas
- 2.4 ¿Colecciones sin polimorfismo?

3 Mensajes de colección
- 3.1 Repaso de filter
- 3.2 Repaso de la interfaz de las colecciones
- 3.3 Encontrar un elemento que satisfaga una condición
- 3.4 Ordenar una colección
- 3.5 For-each

4 Tipos de colecciones Set vs. List

5 Dictionary (BONUS)

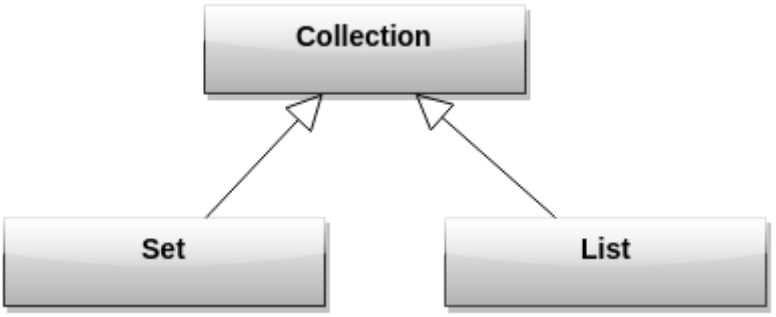
6 Resumen

1 Colecciones revisited

Hasta ahora, definíamos colecciones de dos maneras, utilizando literales:

- `#{}` definía un conjunto, en el cual no había orden ni tampoco elementos repetidos
- `[]` definía una lista, en la cual los elementos conservaban el mismo orden en el que fueron ingresados y el mismo elemento podía estar más de una vez

Bien, ahora que conocemos el concepto de clase como un mecanismo de agrupación de objetos, presentamos dos tipos de colecciones de Wollok: las clases **Set** y **List**. La primera define el conjunto matemático (sin orden ni duplicados) y la segunda la lista (cuyos elementos respetan el orden de ingreso y donde puede haber repetidos).



1.1 Interfaz de una colección

La interfaz de Set y List es la misma que conociste en el módulo 4, así que por este lado no hay cambios:

```
const unConjunto = new Set()
unConjunto.add(1)
unConjunto.add(3)
unConjunto.size()
```

2 Misiones de un héroe, parte 2

2.1 Definiciones: de objetos a clases

Para utilizar clases en nuestro ejercicio de los héroes, tenemos que reformular algunas cosas[1]:

- liberarAFiona es una instancia de LiberarDoncella
- buscarPiedraFilosofal es una instancia de BuscarItemMagico

[1] El ejemplo completo se puede descargar de https://github.com/wollok/heroes-con-clases/

- aprovecharemos las propiedades de Wollok para eliminar código *boilerplate*

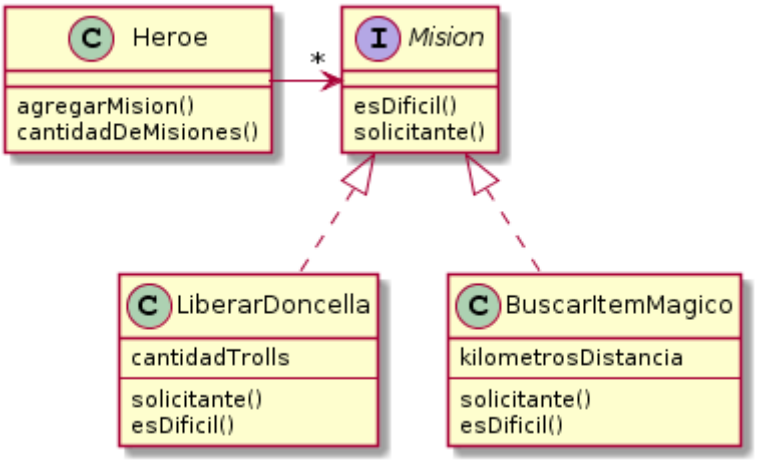
```
class LiberarDoncella {
	var property solicitante = ""
	const cantidadTrolls

	method esDificil() = cantidadTrolls.between(4, 5)

	method puntosRecompensa() = cantidadTrolls * 2
}

class BuscarItemMagico {
	var property solicitante = ""
	const property kilometrosDistancia

	method esDificil() = kilometrosDistancia > 100

	method puntosRecompensa() =
		if (kilometrosDistancia > 50) 10 else 5
}
```

El objeto shrek se transforma en una clase Héroe, con un atributo misiones que podría definirse con el literal #{}, o también construyendo explícitamente una instancia de Set:

```
class Heroe {
	var misiones = new Set()
	... el resto, igual
```

2.2 Diagrama de clases de la solución

Realizamos un diagrama de clases de la solución generada:



Este es un diagrama con características *estáticas*, porque no depende de un caso particular sino que muestra las relaciones entre las clases.

¿Qué es lo que cuenta el diagrama?

- Que un héroe tiene **muchas** misiones, definido por el asterisco (*).
- ¿Qué significa la caja Misión? Una interfaz, ya que LiberarDoncella y BuscarItemMagico no comparten código, pero sí tienen en común que ambos implementan los mensajes solicitante() y esDificil().
- La línea punteada que termina en un triángulo cerrado marca una relación de **Realización** (implementa) entre LiberarDoncella/Misión y BuscarItemMagico/Misión.
- La línea continua con una flecha abierta marca una relación de **Asociación** entre Héroe y Misión.
- Para más información, recomendamos leer [este apunte].

La dirección marca qué objeto conoce a los otros: como la flecha va de Héroe a Misión sabemos que cada héroe tiene n misiones, no conocemos qué características tiene la relación de Misión a Héroe, pero se pueden dar dos opciones:

- una misión puede ser realizada por un héroe solo, en ese caso la relación Héroe-Misión es de uno a muchos
- una misión puede ser encarada por varios héroes. En ese caso la relación es de muchos a muchos.

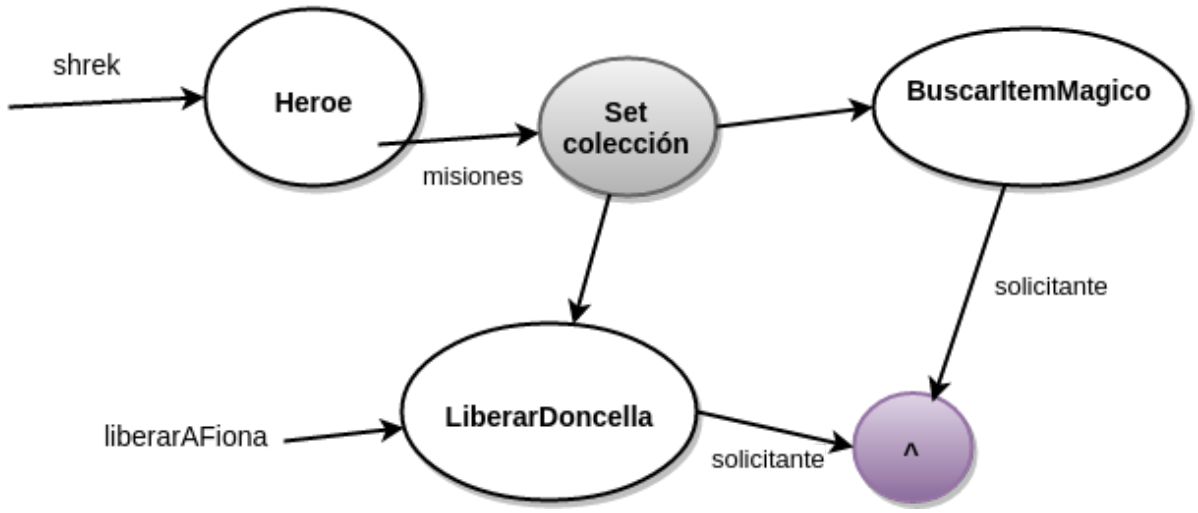
2.3 Referencias: algunas pruebas

Generamos las misiones de shrek

```
Wollok interactive console (type "quit" to quit):
>>> const shrek = new Heroe()
a Heroe[misiones=#{}]
```

```
>>> const liberarAFiona = new LiberarDoncella(cantidadTrolls = 5)
a LiberarDoncella[cantidadTrolls=5, solicitante=]
>>> shrek.agregarMision(liberarAFiona)
>>> shrek.agregarMision(new BuscarItemMagico(kilometrosDistancia =
40))
>>> shrek.cantidadDeMisiones()
2
```

Vemos el gráfico de los objetos en el ambiente (aquí vemos que la instancia de la clase LiberarDoncella tiene 2 referencias):



Este es un diagrama con características *dinámicas*, porque muestra el estado de los objetos en un momento determinado.

2.4 ¿Colecciones sin polimorfismo?

Si quiero cambiar la referencia de liberarAFiona

```
>>> liberarAFiona = 5
```

Eso produce un error:

```
ERROR: No se pueden modificar las referencias constantes (linea: 1)
```

Esto es debido a que definimos nuestra referencia como constante (inmutable):

```
>>> const liberarAFiona = new LiberarDoncella(cantidadTrolls = 5)
```

¿Y si agregamos una referencia a un valor que no sea una misión?

```
>>> var liberarAPepe = 4
4
>>> shrek.agregarMision(liberarAPepe)
>>> shrek.cantidadDeMisiones()
3
```

Esto parece funcionar, el problema es cuando intente usar polimórficamente al entero con las misiones, es entendible que se produzca un error:

```
>>> shrek.solicitantesDeMisMisiones()
```

el mensaje de error es largo, la línea que nos marca el problema es esta:

```
wollok.lang.MessageNotUnderstoodException: 4 no entiende el mensaje solicitante()
   at wollok.lang.Object.messageNotUnderstood(messageName,parameters) (classpath:/wollok/lang.wlk:228)
```

3 Mensajes de colección

3.1 Repaso de filter

Volvamos al método misionesDificiles, antes del objeto shrek y ahora de Heroe:

```
method misionesDificiles() =
      misiones.filter({ mision => mision.esDificil() })
```

Sabemos lo que hace, filtra las misiones que son difíciles, lo importante es que Héroe delega el algoritmo de filtrado a la colección (el objeto que representa el conjunto de misiones). Y para eso, construye un **bloque de codigo** que recibe una misión y devuelve un valor booleano.

Lo podemos probar en el REPL:

```
>>> { mision => mision.esDificil() }.apply(new LiberarDoncella(cantidadTrolls = 4))
true
>>> { mision => mision.esDificil() }.apply(new LiberarDoncella(cantidadTrolls = 7))
false
```

Y podemos construir un filter *wollokiano*:

```
>> clase Set
      method filter(condicionFiltro) {
            const result = #{ }
            self.forEach({ elemento =>
                  if (condicionFiltro.apply(elemento)) {
                        result.add(elemento)
                  })
            }
            return result
```

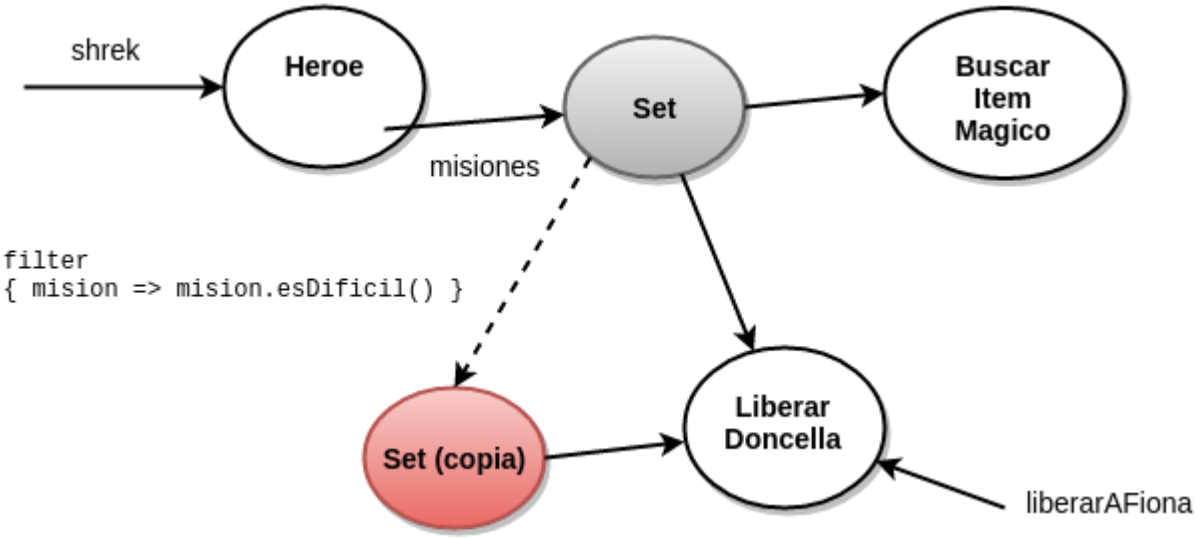
```
}
```

Claro que, al tenerlo resuelto, yo sólo me concentro en

- decir que quiero usar **filter**
- generar el **bloque de código** que defina el criterio de filtrado de los elementos

Así -como dijimos anteriormente- subimos el nivel de **declaratividad**.

Recordamos que el filter no tiene efecto colateral, se devuelve una nueva colección con los elementos que cumplen dicha condición:



3.2 Repaso de la interfaz de las colecciones

Recordemos que map permite transformar los elementos de una colección, generando una nueva colección.

```
method solicitantesDeMisMisiones() =
      misiones.map({ mision => mision.solicitante() })
```

El método solicitantesDeMisMisiones utiliza un bloque que transforma una misión en un solicitante.

Para obtener el total de puntos de recompensa, utilizamos el mensaje sum:

```
method totalPuntosDeRecompensa() =
      misiones.sum({ mision => mision.puntosRecompensa() })
```

Wollok también trae el mensaje fold, para reducir los elementos de una colección a un valor:

```
method totalPuntosDeRecompensa() =
      misiones.fold(0,
            { acum , mision => acum + mision.puntosRecompensa() })
```

Para el lector: puede probar en una consola REPL

```
>>> { acum , mision => acum + mision.puntosRecompensa() }
      .apply(0, new LiberarDoncella(cantidadTrolls = 5))
10

>>> { acum , mision => acum + mision.puntosRecompensa() }
      .apply(25, new LiberarDoncella(cantidadTrolls = 5))
35
```

3.3 Encontrar un elemento que satisfaga una condición

Si queremos saber qué misión tiene más de 5 puntos de recompensa, ¿cómo hacemos?

```
shrek.misionQueTengaMayorRecompensaQue(5)
```

Y lo implementamos así:

```
method misionQueTengaMayorRecompensaQue(puntos) =
   misiones.find({ mision => mision.puntosRecompensa() > puntos })
```

Una vez más utilizamos el concepto de **bloques de código** para devolver el primer elemento que cumple una condición[2].

3.4 Ordenar una colección

Para ordenar las misiones por puntos de recompensa:

```
method misionesOrdenadasPorPuntos() =
      misiones.sortedBy({ mision1, mision2 =>
            mision1.puntosRecompensa() < mision2.puntosRecompensa()})
```

Utilizamos un bloque, que define el criterio de ordenamiento para dos elementos y una condición

- si la condición se cumple, el primer elemento se ordena antes que el segundo
- en caso contrario, es el segundo elemento el que queda antes que el primero.

¿Cómo hacemos si queremos ordenar de mayor a menor? Por supuesto, tenemos que invertir la condición, para hacer que los elementos queden ordenados al revés que en la solución original:

[2] El lector atento podrá notar la presencia de una variable libre: el parámetro puntos. Y aquí comienza a introducirse la distinción entre closures y lambdas que mencionamos anteriormente. (recomendamos leer la respuesta del usuario silvali)

```
method misionesOrdenadasPorPuntos() =
      misiones.sortedBy({ mision1, mision2 =>
            mision1.puntosRecompensa() > mision2.puntosRecompensa()})
```

3.5 For-each

Al presentar el mensaje filter, estuvimos presentando “sin querer” el mensaje forEach, que aplica una operación sobre todos los elementos de una colección. Esto tiene sentido cuando esa operación tiene efecto colateral. Por ejemplo, si queremos modificar todos los solicitantes de las misiones que tiene un héroe.

Para resolverlo, enviamos un mensaje:

```
shrek.cambiarSolicitanteDeMisiones("señor X")
```

E implementamos el método cambiarSolicitanteDeMisiones en la clase Héroe:

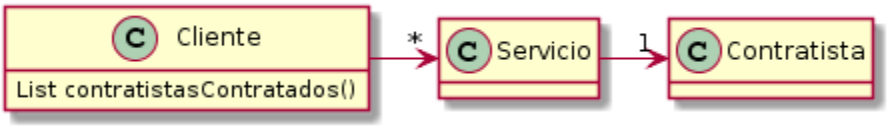
```
method cambiarSolicitanteDeMisiones(nuevoSolicitante) {
      misiones.forEach(
            { mision => mision.solicitante(nuevoSolicitante) })
}
```

- Hasta ahora, nos interesaba saber qué devolvía cada mensaje
- Al pedir shrek.cambiarSolicitante(...) lo que nos importa es que ese mensaje tenga efecto sobre shrek. Entonces para comprobar que el mensaje produjo ese efecto, lo tenemos que complementar con otro mensaje que muestre que las misiones tienen el nuevo solicitante. Una solución posible es generar el método getter para la colección de misiones.

Si queremos trabajar en forma declarativa, deberíamos utilizar forEach solo cuando necesitemos producir efecto sobre los elementos de una colección.

4 Tipos de colecciones Set vs. List

Consideremos el siguiente ejemplo:



Un cliente contrata servicios para arreglar su casa (pintura, plomería, albañilería, etc.). Cada servicio involucra diferentes contratistas. Tenemos un cliente Milena, y dos contratistas: Silvina y Eliana (modelados como instancias de la clase Contratista).

Cada vez que un cliente contrata a un contratista se genera un nuevo servicio donde se le asocia el contratista y la fecha de la contratación.

Un usuario nos reporta que este test falla, ¿qué podrá ser?

```
test "La lista de contratados por milena esta compuesta por silvina
y eliana" {
	const milena = ... creación de Milena ...
	milena.contratarA(silvina)
	milena.contratarA(silvina)
	milena.contratarA(eliana)
	milena.contratarA(eliana)
	assert.equals([silvina,eliana],
		milena.contratistasContratados().asSet())
}
```

Sabemos que contratistasContratados devuelve una lista. Como se que esa lista de contratistas va a devolver elementos duplicados, lo convierto a un conjunto enviando el mensaje asSet. Pero aun así, el test falla porque estoy comparando ese set contra... silvina y eliana que es... ¡**una lista!**

En Wollok

- dos conjuntos son iguales si tienen los mismos elementos **no importa el orden y no importa si para armar el conjunto repetimos varias veces los elementos**
- dos listas son iguales si tienen **los mismos elementos en el mismo orden** entonces ojo porque [1, 2, 3] es diferente de [1, 3, 2]

Algunas consultas para reforzar lo dicho:

```
>>> [1, 2, 3] == [1, 2, 3]
true
>>> [1, 2, 3] == [1, 3, 2]
false
>>> [1, 2, 3] == [1, 3, 2, 3]
false
```

Mientras que

```
>>> #{1, 2, 3} == #{1, 3, 2}
true
>>> #{1, 2, 3} == #{1, 3, 2, 3}
true
```

5 Dictionary (BONUS)

Tenemos el siguiente requerimiento: dada una colección de "Rendiciones de gastos", donde cada rendición conoce

- la persona que realizó el gasto (por el momento es solo un String)
- la fecha en la que hizo el gasto
- la cantidad de plata que gastó

queremos poder saber los totales por mes y por persona.

Pensemos qué queremos: si tenemos estas rendiciones

Fecha	Quién	Cuánto
hoy	Juan	100
hace un mes	Juan	250
ayer	Juan	120
hoy	Ricky	500
anteayer	Ricky	50

```
>>> const reporter = new Reporter()
>>> reporter.mostrarTotalesPorCliente()
Juan - $ 470
Ricky - $ 550
>>> reporter.mostrarTotalesPorFecha()
a Date[day = 21, month = 9, year = 2016] - $ 250
a Date[day = 19, month = 10, year = 2016] - $ 50
a Date[day = 20, month = 10, year = 2016] - $ 120
a Date[day = 21, month = 10, year = 2016] - $ 600
```

Bien, para lograr eso, necesitamos la clase Rendición[3]:

```
class Rendicion {
      const property persona
      const property fecha
      const property total

      }
```

Nada del otro mundo. La clase Reporter tendrá en el constructor la generación del juego de datos:

[3] El ejemplo completo se puede descargar en https://github.com/wollok/dictionary-totales

```
class Reporter {
	const hoy = new Date()
	const rendiciones = [
		new Rendicion(persona = "Juan",
			      fecha = hoy,
			      total = 100),
		new Rendicion(persona = "Juan",
			      fecha = new Date().minusMonths(1),
			      total = 250),
		new Rendicion(persona = "Juan",
			      fecha = new Date().minusDays(1),
			      total = 120),
		new Rendicion(persona = "Ricky",
			      fecha = hoy,
			      total = 500),
		new Rendicion(persona = "Ricky",
			      fecha = new Date().minusDays(2),
			      total = 50)
	]
}
```

Para sacar los totales por cliente, el Dictionary es una estructura auxiliar que nos sirve particularmente, porque permite acceder en forma directa al cliente que estoy buscando. El único tema es que la primera vez que averigüe el total de un cliente no va a estar en esa estructura, entonces debo asumir que el total es 0:

```
method totalPorCliente() {
	const totales = new Dictionary()
	rendiciones.forEach{rendicion =>
		var total = totales.getOrElse(rendicion.persona(), { 0 })
		total += rendicion.total()
		totales.put(rendicion.persona(), total)
	}
	return totales
}
```

Entonces de la lista generamos un conjunto de pares clave - valor, donde tenemos

- Persona
- Total

Lo mismo podemos hacer para la fecha:

```
method totalPorFecha() {
	const totales = new Dictionary()
	rendiciones.forEach{ rendicion =>
		var total = totales.getOrElse(rendicion.fecha(), { 0 })
```

```
            total += rendicion.total()
            totales.put(rendicion.fecha(), total)
        }
        return totales
}
```

Tenemos este método que nos ayuda a mostrar los resultados por pantalla, no es algo que queramos que hagan pero para este ejemplo el test es un poco engorroso de armar:

```
method mostrarTotalesPorCliente() {
        self.totalPorCliente().forEach({ persona, total =>
            console.println(persona + " - $ " + total)
        })
}
```

El forEach de un Dictionary recorre cada par de elementos clave y valor.

En general, los Dictionaries (también llamados mapas en otros lenguajes) son estructuras útiles para generar totales o bien para hacer cachés de valores, por su manera cómoda de acceder por clave.

6 Resumen

A lo largo de este capítulo hemos repasado el ejemplo de manejo de colecciones, ahora presentando las clases Set, List y Dictionary. La interfaz de las colecciones no ha cambiado, seguimos teniendo mensajes poderosos como map, filter, sum, find, etc. que nos permiten subir la declaratividad de nuestras soluciones, valiéndonos para ello de los bloques de código (o lambdas), que nos permiten definir comportamiento anónimo.

Paradigma Orientado a Objetos

Módulo 13: Herencia. Super. Redefinición.

por Fernando Dodino

Versión 2.2

Septiembre 2018

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Herencia
- 1.1 Enunciado base
- 1.2 Intro a superclase
- 1.3 Representación

2 Herencia en Wollok: ejemplo de las aves
- 2.1 Golondrina Tijerita
- 2.2 Method lookup de pepita

3 Segundo caso: el Petrel
- 3.1 Modelo y realidad
- 3.2 Volviendo al código de petrel
- 3.3 Redefinición de métodos
- 3.4 Super
- 3.5 Method lookup de unPetrel

4 Tercer ejemplo: la torcaza
- 4.1 Self
- 4.2 Method lookup de torcaza

5 Más sobre herencia
- 5.1 Clases abstractas y concretas
- 5.2 Herencia vs. composición
- 5.3 ¿Instancias o clases?
- 5.4 Cambiar la clase

6 Herencia de WKO
- 6.1 Method lookup de un WKO con herencia
- 6.2 Objetos y clases polimórficos

7 Resumen

1 Herencia

1.1 Enunciado base

Un ornitólogo, luego de estudiar el comportamiento de las golondrinas tijerita como pepita, se dio cuenta de que existen otros 2 tipos de aves que le interesan:

- el petrel
- y la torcaza

Todas saben volar y comer igual que pepita, pero

- al ornitólogo le interesa saber la cantidad de kilómetros que vuela un petrel
- la torcaza es medio atolondrada, antes de comer se pone tan contenta que vuela en círculos 1 kilómetro

1.2 Intro a superclase

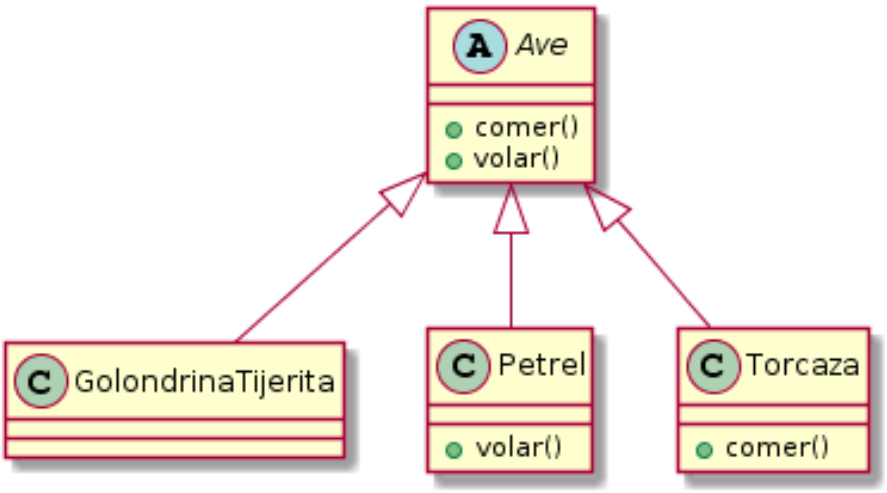
La golondrina tijerita, el petrel y la torcaza saben comer y volar. Esto las hace en principio **polimórficas**, pero además la forma en que comen y vuelan es muy similar, aunque no exactamente igual.

Sabemos que una clase es el concepto que permite agrupar el comportamiento (y la estructura) de cada una de las instancias. Pero como el petrel “vuela distinto” (para nuestro sistema) y la torcaza “come distinto”, esto nos lleva a modelar tres clases diferentes: GolondrinaTijerita, Petrel y Torcaza. Ahora bien, no queremos repetir el mismo código para las 3 clases al comer y al volar.

Afortunadamente, podemos relacionar una clase con otra: GolondrinaTijerita, Petrel y Torcaza pueden tener a Ave como clase madre o **superclase**. Las superclases tienden a representar un concepto más general, mientras que las subclases se enfocan a especializaciones o casos particulares.

1.3 Representación

Generamos la representación en el diagrama de clases:



La flecha con el triángulo cerrado marca la relación de herencia. Las subclases de Ave heredan a priori sus atributos y comportamiento.

2 Herencia en Wollok: ejemplo de las aves

Recordemos cómo es el comportamiento de una golondrina tijerita

- cuando vuela, consume un joule para cada kilómetro que vuela, más 10 joules de "costo fijo" en cada vuelo
- cuando come, adquiere 4 joules por cada gramo que come

Este comportamiento puede ahora ser la definición por defecto de un Ave. Creamos la clase Ave en Wollok[1]:

```
/**
* Definición por defecto de un Ave
*/
class Ave {
	var energia = 50

	method volar(kilometros) {
		energia = energia - (kilometros + 10)
	}

	method comer(gramos) {
		energia = energia + gramos * 4
	}
}
```

[1] El ejemplo completo puede descargarse de https://github.com/wollok/herencia-aves-pepita

2.1 Golondrina Tijerita

¿Qué sucede con la golondrina tijerita? Se comporta ahora “igual que un ave”. Esto lo definimos así:

```
class GolondrinaTijerita inherits Ave { }
```

Entonces una golondrina tijerita, por heredar de Ave

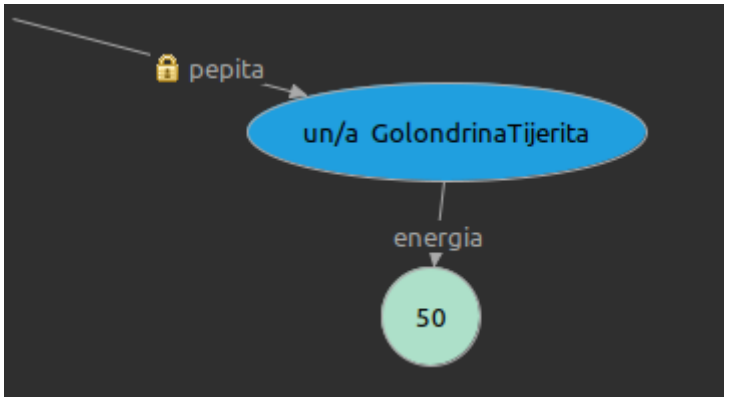
- tiene energía
- sabe comer y volar, como lo hacen todas las aves

Además agrega un comportamiento que no viene al caso en este momento:

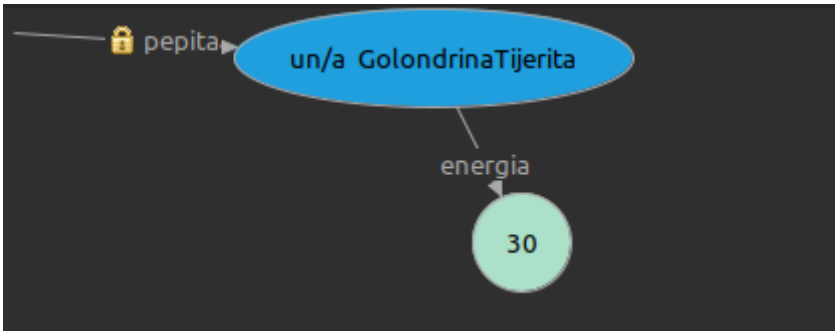
```
class GolondrinaTijerita inherits Ave {
     method hacerAlgo(conCosa) { ... }
}
```

Podemos verlo al instanciar una golondrina tijerita en la consola REPL:

```
>>> const pepita = new GolondrinaTijerita()
a GolondrinaTijerita
```




```
>>> pepita.volar(10)
```



Al instanciar a pepita, tenemos una referencia llamada energía, y podemos enviarle el mensaje volar. Pero ¿cómo es que pepita entiende volar?

2.2 Method lookup de pepita

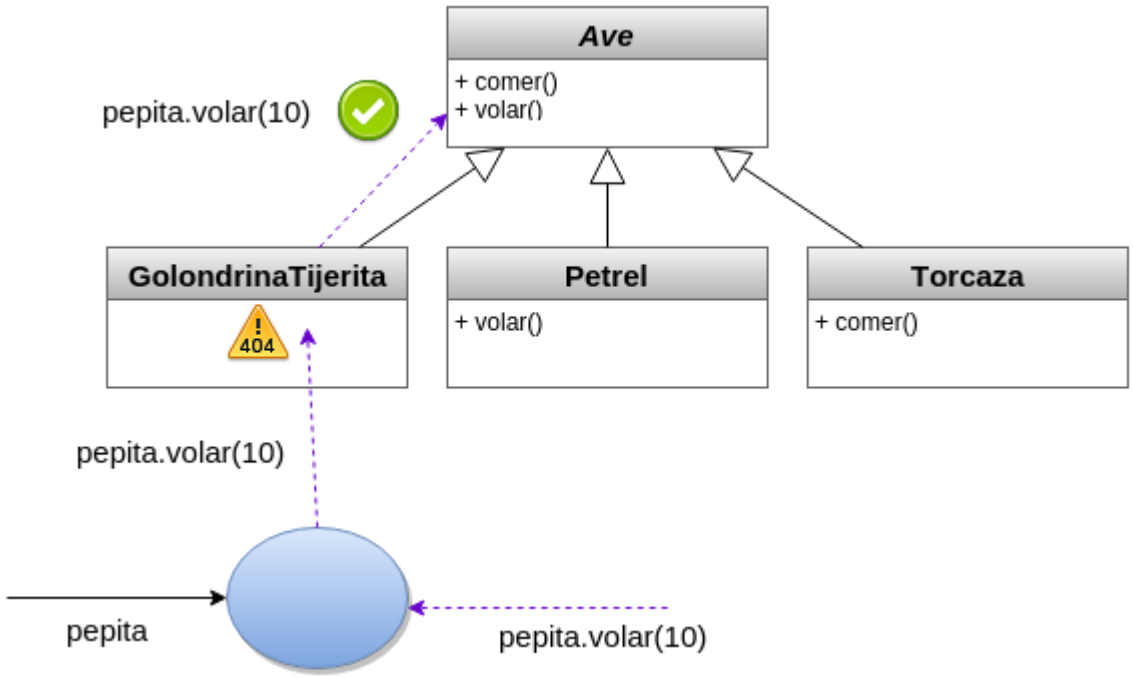
Tenemos que revisar la definición del *method lookup*. Hasta ahora, al enviar un mensaje a un objeto,

- si se trata de un objeto autodefinido o “well-known object”, se ejecuta el método que está en dicho objeto
- si es instancia de una clase, comenzamos buscando el método en la clase a la que pertenece dicha instancia. En el ejemplo de pepita, buscamos el método “volar(kilometros)” en la clase GolondrinaTijerita.

Ahora extendemos lo que sucede si no encontramos la definición en la clase del objeto receptor:

- la búsqueda continúa en la superclase, en este caso Ave. Aquí tenemos definido un método volar con un parámetro y se evalúa para los valores que tiene pepita.
- En el caso de no existir en la superclase, se sigue buscando en la clase superior, hasta llegar a Object (la clase madre de todas las clases). Y finalmente si no encontramos allí el método que buscamos, recibimos un mensaje de error.

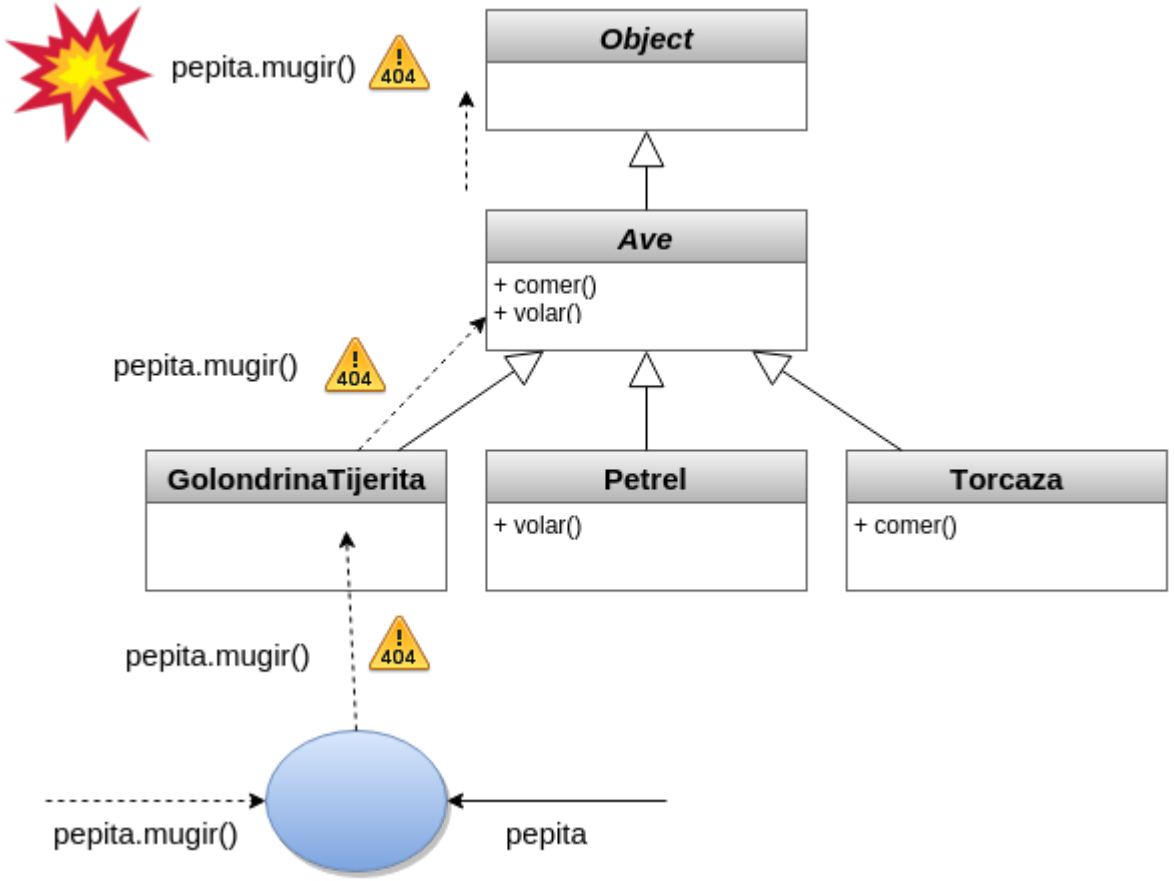
Repasamos el gráfico en el envío del mensaje pepita.volar(10):



Vemos qué pasa cuando enviamos un mensaje que no puede ser respondido:

```
>>> pepita.mugir()
wollok.lang.MessageNotUnderstoodException: un/a  GolondrinaTijerita
no entiende el mensaje mugir()
```

Y lo vemos gráficamente:



3 Segundo caso: el Petrel

El petrel

- come y vuela igual que un ave
- además el ornitólogo quiere registrar la cantidad de kilómetros que voló

La primera pregunta que podríamos hacernos es: ¿es responsabilidad del petrel registrar los kilómetros de vuelo? ¿no es el ornitólogo el que termina haciendo el trabajo?

3.1 Modelo y realidad

Claro, pero ¡¡ojo!! Acá nos estamos confundiendo modelo y realidad

- en la realidad, el petrel vuela, el ornitólogo anota en un cuaderno las veces que voló

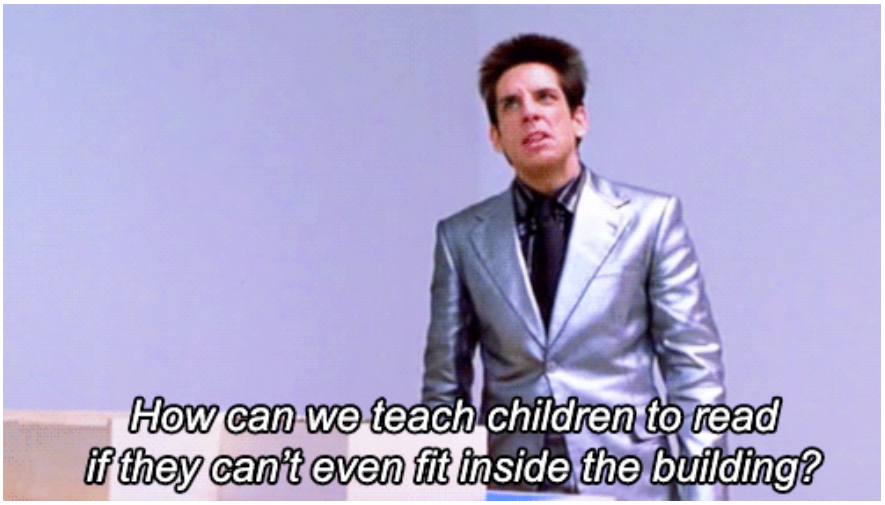
- en nuestro modelo, el petrel no es el petrel, sino una representación del petrel que es mucho menos compleja, a la que le sacamos todas las características que no son esenciales mediante el proceso de abstracción.

Dejamos al lector dos ideas asociadas:
El cuadro de René Magritte, “Esto no es una pipa”

En él, Magritte resalta la idea de que dibujar una pipa no constituye una pipa en sí, sino una representación de la pipa (acotada por el observador).

En la película Zoolander[2], al protagonista Ben Stiller le muestran una maqueta de un “Centro para niños que no saben leer bien”:

[2] Idea extraída de una clase de nuestro amigo Pablo Beláustegui



Indignado, el personaje Derek Zoolander replica: *"¿Qué es esto, un centro para hormigas? ¿Cómo vamos a enseñarle a los chicos a leer si es imposible que quepan en el edificio?"*

Zoolander también confunde modelo y realidad.

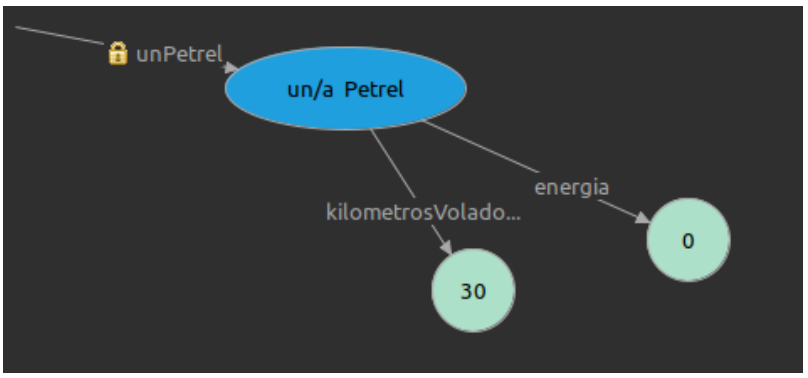
3.2 Volviendo al código de petrel

Entonces el petrel de nuestro ejemplo se puede implementar así:

```
class Petrel inherits Ave {
	var kilometrosVolados = 0

	override method volar(kilometros) {
		super(kilometros)
		kilometrosVolados = kilometrosVolados + kilometros
	}
}
```

Antes de explicar lo que hicimos arriba, lo probamos:

```
>>> const unPetrel = new Petrel()
>>> unPetrel.volar(10)
>>> unPetrel.volar(20)
```



Un detalle: además de los kilómetros volados (definidos en Petrel), el petrel tiene energía, porque dijimos que un Petrel "es un" Ave.

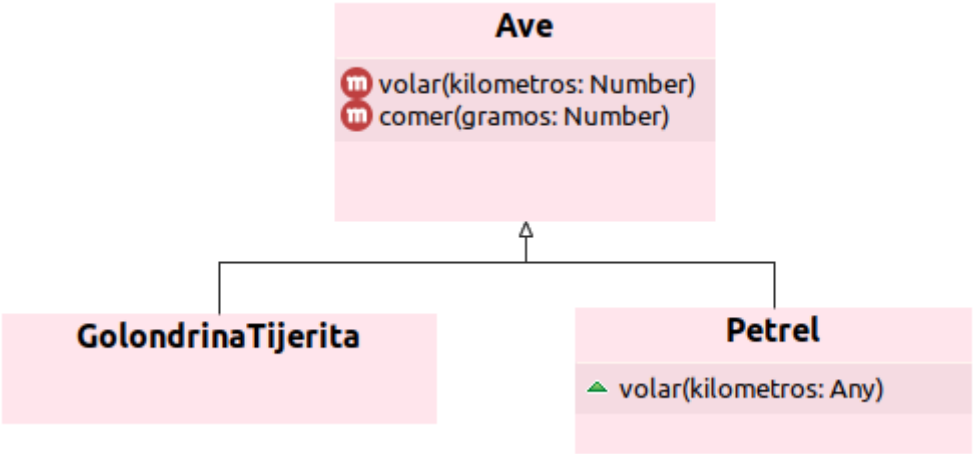
3.3 Redefinición de métodos

¿Por qué aparece la palabra override antes de method?
Porque la subclase está **redefiniendo** comportamiento de la superclase: ya teníamos una definición de volar(kilometros) pero nosotros queremos escribir otra, que va a pisar a la definición original de Ave.

Entonces al enviar el mensaje

```
>>> unPetrel.volar(10)
```

el method lookup comenzará a buscar en la clase receptora de la instancia unPetrel. Esto es... la clase Petrel, que ahora sí tiene una definición de volar. Pueden visualizarlo en el diagrama estático[3]:



La flecha verde sobre volar de Petrel indica que estamos redefiniendo el comportamiento de la superclase (en este caso Ave).

[3] Para más información pueden ver el apunte *Anexo 3: Diagrama estático de Wollok*

3.4 Super

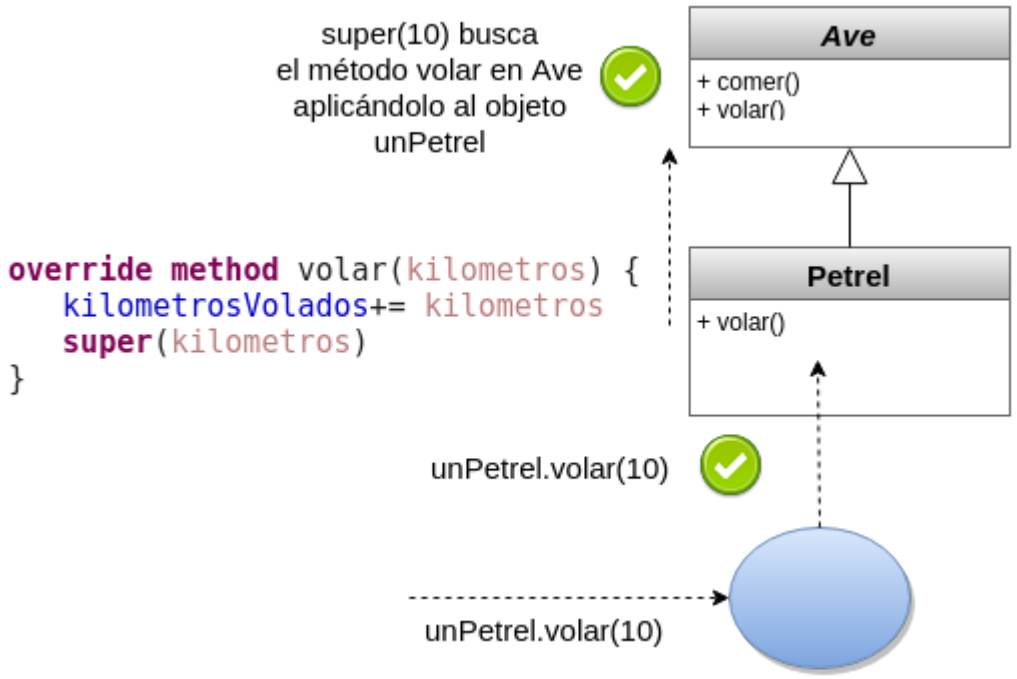
Queremos aprovechar el comportamiento de volar() que está definido en Ave, pero sin repetir el código. Por eso utilizamos la pseudo-variable super:

```
override method volar(kilometros) {
      super(kilometros)
      kilometrosVolados = kilometrosVolados + kilometros
}
```

Esto permite alterar el method lookup en el contexto en donde estamos, salteando la clase del objeto receptor y comenzando por su superclase. Esto es útil particularmente cuando estamos en un método redefinido y queremos evitar un loop infinito de llamadas al mismo método en el cual estamos.

3.5 Method lookup de volar par un petrel

Mostramos gráficamente cómo es el method lookup para el mensaje `unPetrel.volar(10)`:



4 Tercer ejemplo: la torcaza

La torcaza

- vuela como un ave

- y al ser medio atolondrada, por un lado come igual que un ave pero antes de comer se pone tan contenta que vuela en círculos 1 kilómetro

Ya sabemos que si heredamos de Ave nuestra definición de volar() no debe cambiar, debemos redefinir el método comer().

4.1 Self

Al comer, tenemos que hacer que vuele un kilómetro. Pero no queremos repetir el código de volar() dentro del método comer():

```
class Torcaza inherits Ave {

	override method comer(gramos) {
		energia = energia - (1 + 10) // copio el método volar, je
		super(gramos)
	}
}
```

Lo que podríamos hacer es enviar el mensaje volar(1), pero ¿a quién? Al mismo objeto receptor, usamos para eso la referencia *self*:

```
class Torcaza inherits Ave {

	override method comer(gramos) {
		self.volar(1)
		super(gramos)
	}

}
```

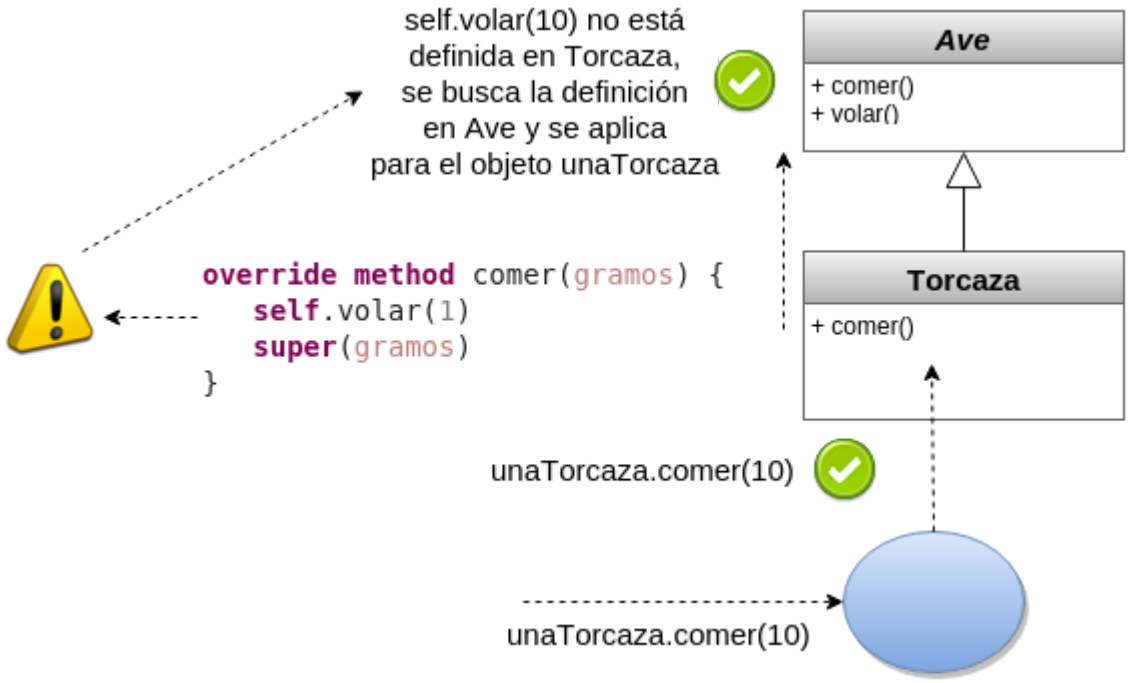
En general

- usamos *self* para enviar mensajes al objeto receptor
- y cuando no podemos usar *self*, porque entraríamos en loop infinito, usamos *super*

La diferencia está en que *self* no cambia el method lookup, que comienza por la clase del objeto receptor, mientras que *super* saltea el primer paso y comienza la búsqueda del método en la superclase de donde se invoca *super*. Pero el objeto receptor **nunca cambia, no hay otro objeto más que la torcaza**.

4.2 Method lookup de torcaza

Vemos el gráfico que muestra el lookup de la primera línea del método comer para torcaza:



El method lookup de la segunda línea es similar al que vimos en petrel:

- super(gramos) busca la definición del método comer(gramos) en Ave y la aplica para la torcaza

5 Más sobre herencia

5.1 Clases abstractas y concretas

Ave es una **clase abstracta**: tiene sentido como una forma de agrupar comportamiento y atributos para las subclases, no para generar instancias de Ave.

- En algunos lenguajes esta definición es explícita: no podés crear una instancia de Ave si dijiste que era una clase abstracta. Al hacer new Ave() el compilador chequea que Ave no sea abstracta y en ese caso se produce un error.

```
public abstract class Ave { ... // Definición explícita en Java
```

- En Wollok, esta definición ocurre naturalmente: si no hago new Ave(), esto implica que Ave termina siendo una clase abstracta, no hace falta decirlo.

Una pregunta frecuente que surge es: ¿las superclases deben ser abstractas? ¿puede haber superclases concretas? La respuesta es: las clases son concretas si tiene sentido instanciarlas, sean superclases, subclases o clases “sueltas” (que sólo hereden de *Object*). Y si una clase es abstracta, seguramente es porque tiene subclases que redefinen algún comportamiento. De otra manera, ¿para qué

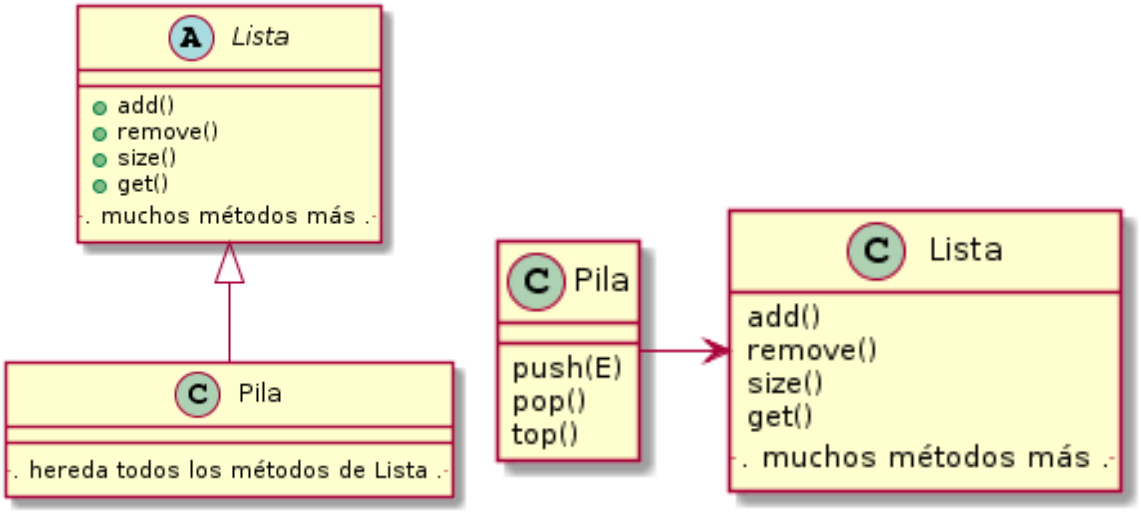
querríamos generar una clase abstracta sin subclasificarla luego? No podemos instanciarla y no sirve como agrupador de ninguna jerarquía, es difícil encontrarle sentido.

5.2 Herencia vs. composición

La herencia marca una relación entre clases (es estática), la superclase tiene características más generales mientras que la subclase toma comportamiento específico y cuando es necesario lo redefine. **En la composición** no hay una jerarquía de clases, sino **que intervienen dos instancias**: una conoce a la otra y le envía mensajes.

En los lenguajes donde tenemos herencia simple, la herencia es un mecanismo más limitado que la composición: la taxonomía de las clases tiene un único punto de vista. El ejemplo "de libro" es que al modelar una clase Perro, tengo que pensar si quiero que la jerarquía esté basada en animales domésticos y salvajes, o en vertebrados e invertebrados, o en mamíferos, ovíparos, ovovivíparos, etc. No puedo tener más de un punto de vista, y esto trae complicaciones por ejemplo al representar un String: ¿debería heredar de una clase Collection (porque es una colección ordenada de caracteres) o de una clase que sepa decirnos si es mayor o menor que otro String, algo así como un Ord de Haskell?

¿Y por qué comparar la herencia con la composición? Supongamos que tenemos que modelar una Pila. La pregunta que nos tenemos que hacer es: ¿una pila *es una* lista? ¿o una pila *tiene* una lista?



Pila es una Lista *Pila tiene una Lista*

Si Pila hereda de Lista, la ventaja es que toma todo el comportamiento de su superclase, y también allí radica su principal desventaja: quizás no sea necesario tener una interfaz tan grande, podríamos tener sólo tres métodos para la pila:

- push(): pone un elemento de la pila
- top(): muestra el elemento que está arriba de todo en la pila
- pop(): saca el elemento que está arriba de todo en la pila

Entonces lo que nos conviene es que una Pila tenga una Lista, pero su implementación quede encapsulada en la Pila. El que utiliza a la Pila no necesita saber cómo está construida, simplemente utiliza los tres mensajes que definimos para la Pila.

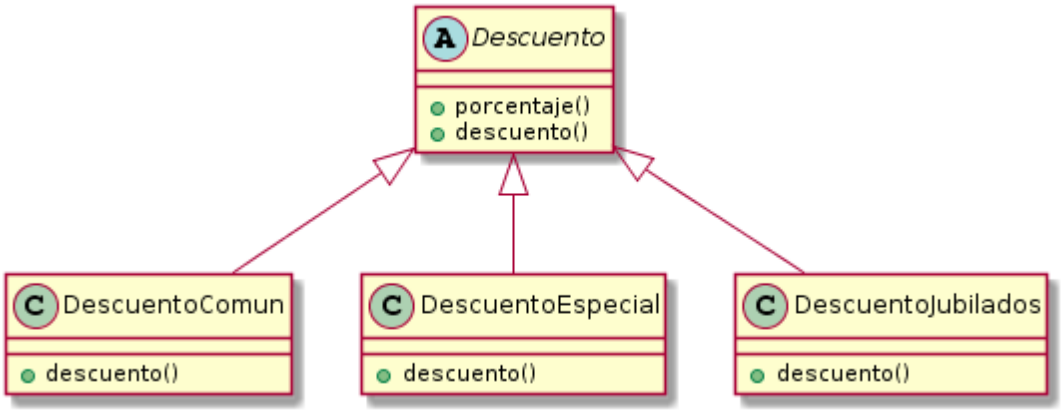
Mediante la composición, permitimos que a futuro la Pila herede de otra superclase que tenga más cosas en común que una Lista, sin gastar el *único tiro* que la herencia provee.

5.3 ¿Instancias o clases?

Dado el siguiente requerimiento

"A la hora de vender, tenemos tres tipos de descuento. El descuento común es del 5%, el especial de un 10% y para jubilados es un 20%".

A primera vista podríamos pensar en subclasificar Descuento en: DescuentoComun, DescuentoEspecial y DescuentoJubilados.



En la clase abstracta Descuento el método porcentaje se escribiría

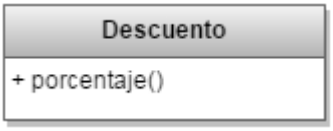
```
>>Descuento
method porcentaje() = self.descuento() / 100
method descuento()
```

El método descuento no tiene cuerpo, es un **método abstracto** (solo sirve para forzar a que las subclases implementen dicha interfaz), y provoca entonces que la clase Descuento sea abstracta.

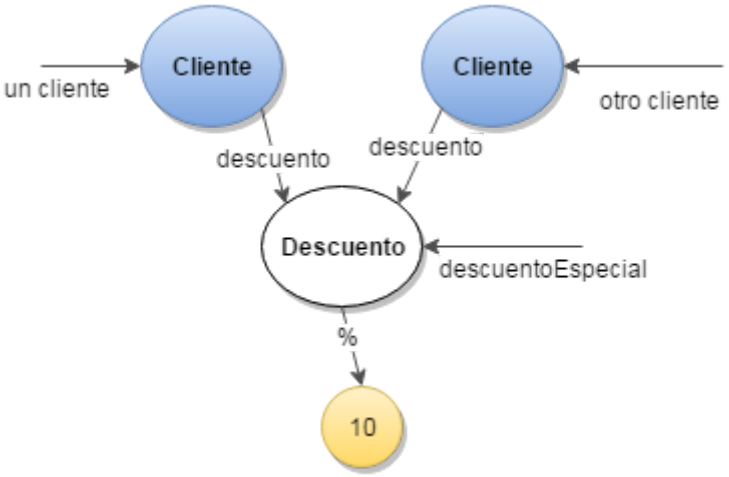
Pero ¿cómo implementaríamos el método descuento en cada subclase?

```
>>DescuentoComun
override method descuento() = 5

>>DescuentoEspecial
override method descuento() = 10

>>DescuentoJubilados
override method descuento() = 20
```

En realidad estamos repitiendo la misma idea, ya que lo único que difiere es el % de descuento. En ese caso podemos modelar una clase Descuento con tres instancias diferentes, una para cada tipo de descuento.




```
const descuentoEspecial = new Descuento(porcentaje = 20)
...
```



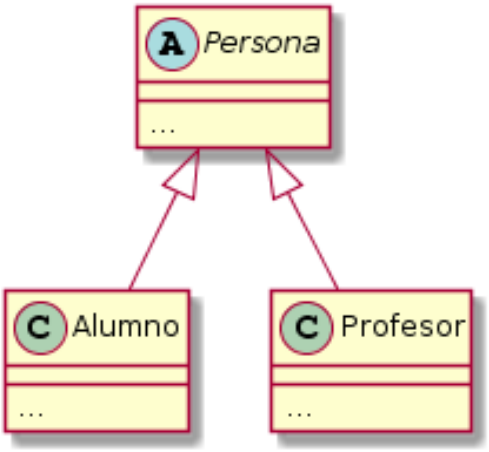
descuentoEspecial es una referencia a una instancia de Descuento

Otro ejemplo: “Modelar las piezas de ajedrez.” ¿tengo una clase Caballo, otra Alfil, otra Rey o me arreglo con 32 instancias de Pieza?

Al modelar con objetos: ¿cuándo usar instancias y cuándo usar clases? Uso clases cuando **hay comportamiento diferente (cuando el código es diferente)**.

5.4 Cambiar la clase

Si tenemos esta jerarquía de clases



Todo parece funcionar bien: el alumno y el profesor pueden en algún contexto ser polimórficos pero cada uno define su propio comportamiento. Ahora, cuando una persona finaliza su cursada, le ofrecen ser ayudante y pasa a ser profesor, entonces tenemos un problema: una instancia no puede pertenecer a dos clases, y tampoco es fácil cambiar una referencia a un alumno por el de un profesor.

De la misma manera jerarquías como joven / adulte, felices / aburridos / indignados, sin hijos / con hijos están basadas en características *temporales* de un objeto y no deberían elegirse como criterio de subclasificación. Por eso una buena práctica para elegir el punto de vista para armar la jerarquía de clases es buscar aquello que sea intrínseco al objeto, de forma tal que un **objeto pertenezca exactamente a una clase durante todo su ciclo de vida**.

6 Herencia de objetos autodefinidos *(wko)*

Volviendo al objeto pepita original:

```
object pepita {
      var energia = 100

      method energia() = energia
      method volar(kms) { energia = energia - (kms + 10) }
      method comer(gramos) { energia = energia + (4 * gramos) }
}
```

Incorporamos otra ave chichita, que

- vuela igual que pepita

- pero come distinto

Pero además, nuestro usuario nos avisa que hay infinidad de otras aves que vuelan y comen de la misma manera que pepita. Tanto *chichita* como *pepita* son dos objetos importantes en nuestra solución, no queremos perder la posibilidad de referenciarlas y mandarles mensajes. Pero no pasa eso con las otras aves: para ellas se pierde el sentido de la individualidad, entonces no necesito generar un *named object* para cada uno, me basta con que sean instancias de una clase Ave.

¿De qué manera podemos reutilizar el comportamiento de pepita, chichita y las demás aves? Wollok permite que un WKO (un objeto conocido) herede de cualquier clase. Entonces podemos mudar el comportamiento de pepita a una clase Ave, y definir el comportamiento específico para chichita:

```
class Ave {
	var energia = 100

	method energia() = energia
	method volar(kms) { energia = energia - (kms + 10) }
	method comer(gramos) { energia = energia + (4 * gramos) }
}

object pepita inherits Ave { ... }

object chichita inherits Ave {
	override method comer(gramos) {
		energia = energia * gramos
	}
}
```

Fíjense que:

- pepita toma el comportamiento de Ave (y debería agregar algún comportamiento individual)
- chichita también toma la definición de la clase Ave pero además pisa el método comer. Como hereda de Ave, tiene acceso a su estructura interna, en este caso la referencia `energia` se puede utilizar directamente.

¿Cómo vuela pepita? Igual que un ave
¿Cómo vuela chichita? Igual que un ave
¿Cómo come chichita? Como solo ella lo sabe hacer

6.1 Method lookup de un WKO con herencia

El method lookup de un objeto se resuelve de la siguiente manera: al enviar un mensaje a un objeto autodefinido

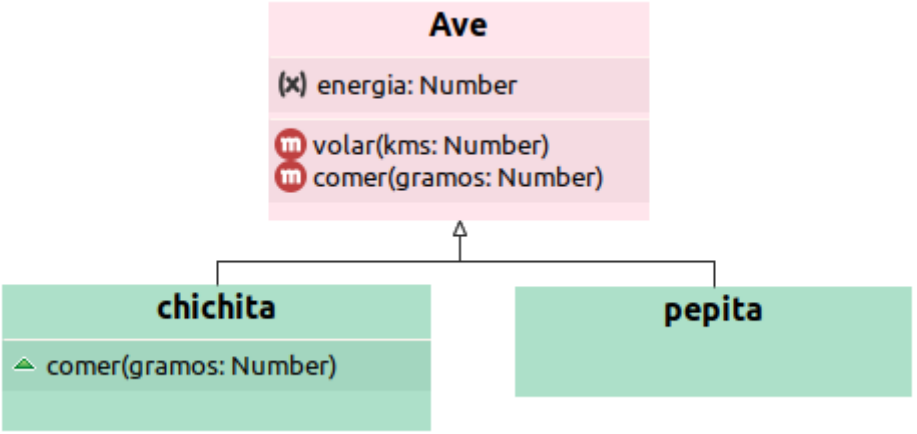
1. primero se busca la definición en el objeto (en el *wko*)
2. si no se encuentra un método y el objeto hereda de una clase, se busca en esta clase
3. ¿y si no está en dicha clase? Se busca en la superclase hasta encontrarla o bien llegar a Object y fallar.

6.2 Objetos y clases polimórficos

Con el ejemplo anterior, el lector habrá notado que tenemos un set de objetos polimórficos:

- pepita
- chichita
- y las instancias de la clase Ave

Todos entienden los mensajes energia(), volar(kms) y comer(gramos).



En un REPL podemos enviar mensajes sin tener que preocuparnos si son WKO o instancias de una clase

```
>>> pepita.volar(10)
>>> chichita.volar(5)
>>> const tweety = new Ave()
>>> tweety.volar(6)
>>> [pepita, new Ave(), tweety, chichita].forEach ({ ave =>
ave.volar(5) })
```

7 Resumen

Las clases pueden organizarse jerárquicamente de la más general a la más particular: el mecanismo de herencia permite reutilizar comportamiento y atributos para evitar duplicación de código, y redefinir en cada subclase lo que sea necesario en cada caso. El method lookup sigue la búsqueda desde la clase del objeto receptor por todas las superclases, hasta encontrar el método o fallar. Mientras que

self permite enviar un mensaje al propio objeto, *super* cambia el mecanismo de method lookup salteando la clase donde está definido el método y comenzando directamente en la superclase.

Por último, a la hora de modelar con objetos es conveniente tener en claro que para elegir la subclasificación debe haber comportamiento diferencial entre las subclases, y tener en cuenta el contrapunto entre herencia y composición como alternativas, cada una con sus puntos a favor y en contra.

Paradigma Orientado a Objetos

Módulo 14: Mutabilidad. Igualdad e identidad.

por Fernando Dodino

Versión 2.2

Noviembre 2018

Distribuido bajo licencia Creative Commons Share-a-like

Indice

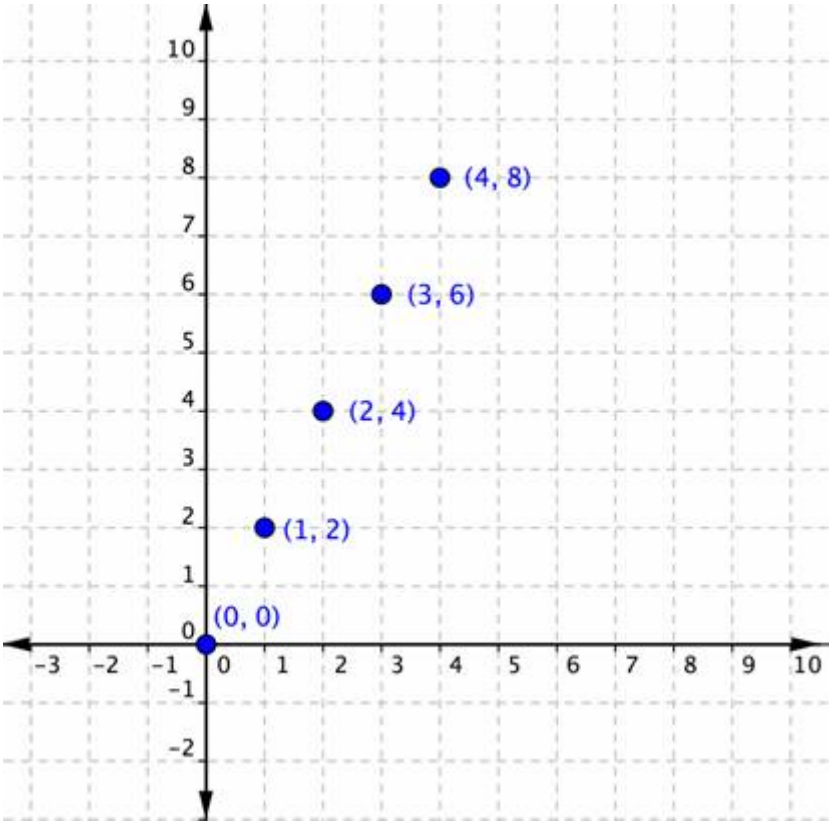
1 Mutabilidad / inmutabilidad
- 1.1 Value objects
- 1.2 Motivación

2 Igualdad e identidad
- 2.1 Identidad
- 2.2 Igualdad

3 Resumen

1 Mutabilidad / inmutabilidad

Si queremos modelar un punto en un eje de coordenadas:



Tenemos dos decisiones de diseño posible:

- hacer que el objeto sea **mutable**, definiendo setters para las propiedades *x* e *y*
- construir un objeto **inmutable**. El punto, una vez construido, no puede variar: representa una ubicación en el plano y no puede representar otro punto más que ése.

¿Qué pasa si queremos sumar dos puntos? Se termina construyendo un punto nuevo...

```
class Point {
      const property x
      const property y

      method +(otroPoint) =
            new Point(x = x + otroPoint.x(), y = y + otroPoint.y())
}
```

Esta misma estrategia podemos adoptar para

- los números: tenemos un objeto 2 y otro que representa al 3, si los sumo el 2 no “cambia” a 5, el 5 es un nuevo objeto
- los strings, que son inmutables: cuando quiero concatenar dos strings, genero uno nuevo
- los booleanos, ya que en realidad existe un solo true y un solo false

1.1 Value objects

En general, todos los objetos que describimos recién entran en la categoría de *Value Objects*: son objetos que representan un valor de nuestro dominio. Otros ejemplos posibles podrían ser: objetos que representan un color, como el rojo, un objeto que modela un mail, un objeto que representa una figura bidimensional (sería un value object construido como una lista de puntos), una fecha, etc.

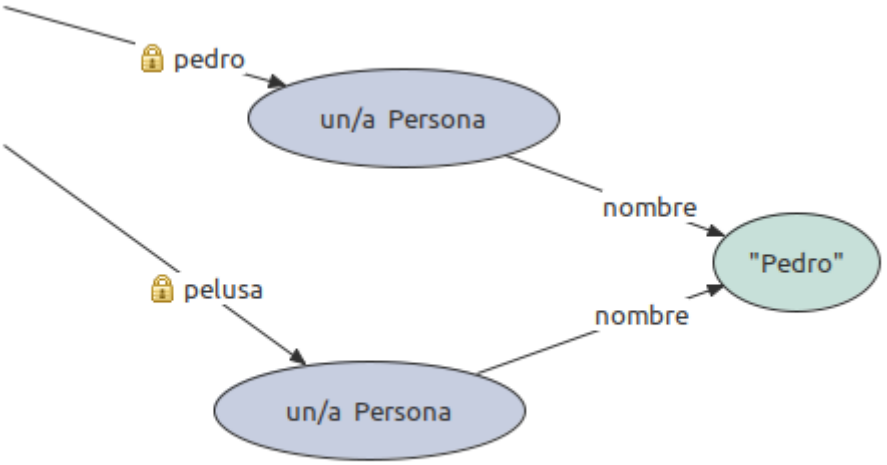
1.2 Motivación

¿Por qué queremos tener objetos inmutables? Porque son objetos que no tienen efecto colateral: más allá de que el paradigma lo soporte, yo elijo no trabajar con este concepto, reforzando la idea de que el paradigma está en quienes desarrollan.

Al no tener efecto colateral

- el testing se simplifica, porque entran en juego una menor cantidad de situaciones y contextos
- es más fácil compartir los objetos en forma concurrente, porque sabemos que nadie puede hacer modificaciones a ese objeto

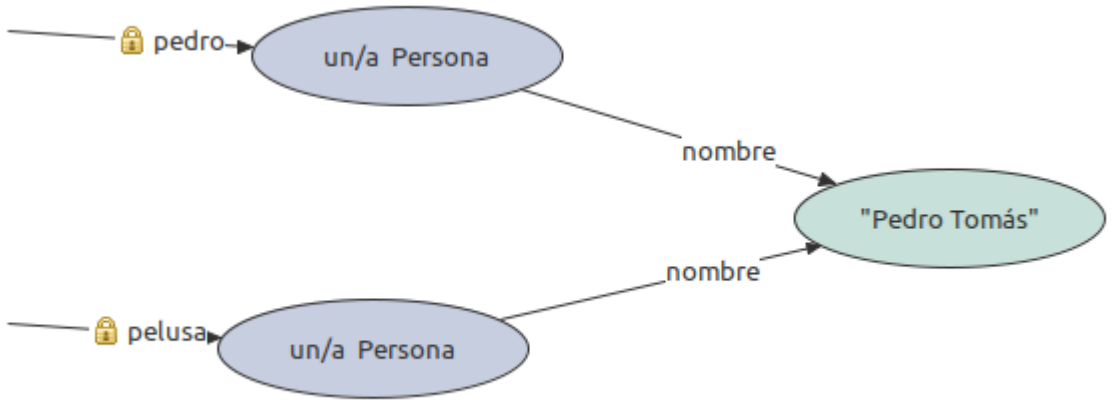
Consideremos por ejemplo dos personas que tienen el mismo nombre: Pedro.



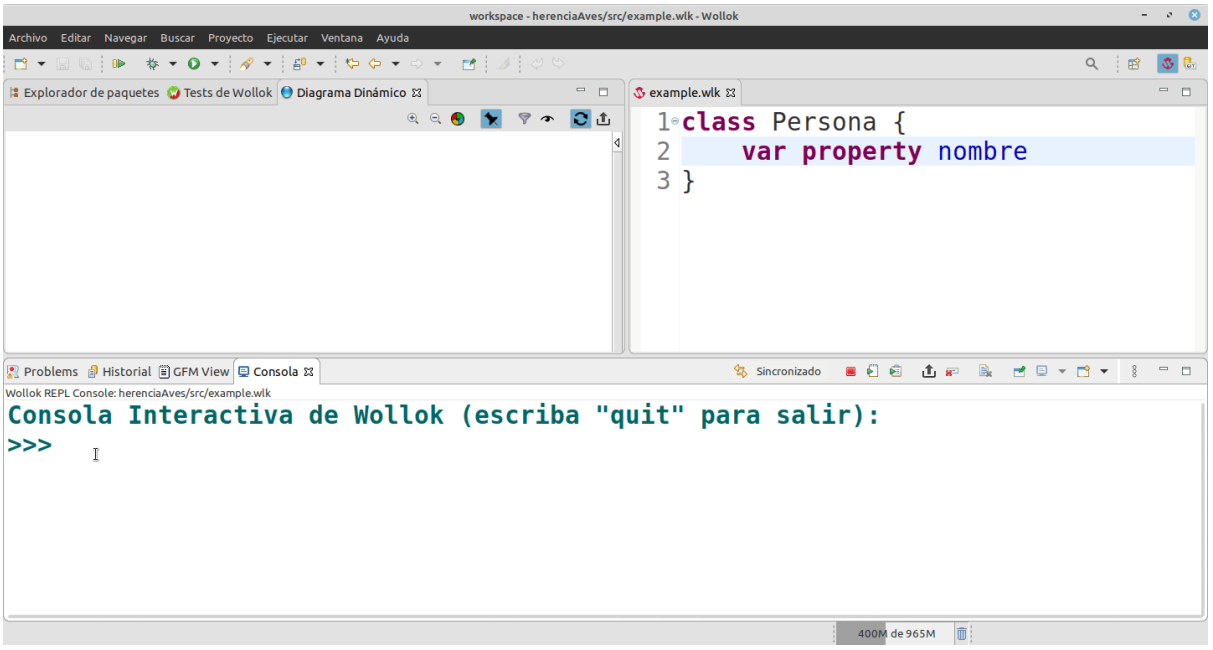
Si los Strings fueran mutables, al enviar el mensaje

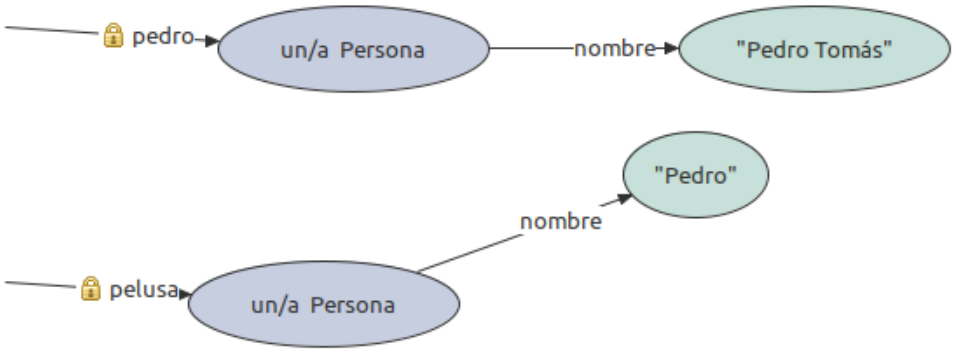
```
pedro.nombre("Pedro Tomás")
```

¡estaríamos cambiando la referencia nombre de `pelusa`!



Pero eso no es lo que sucede:



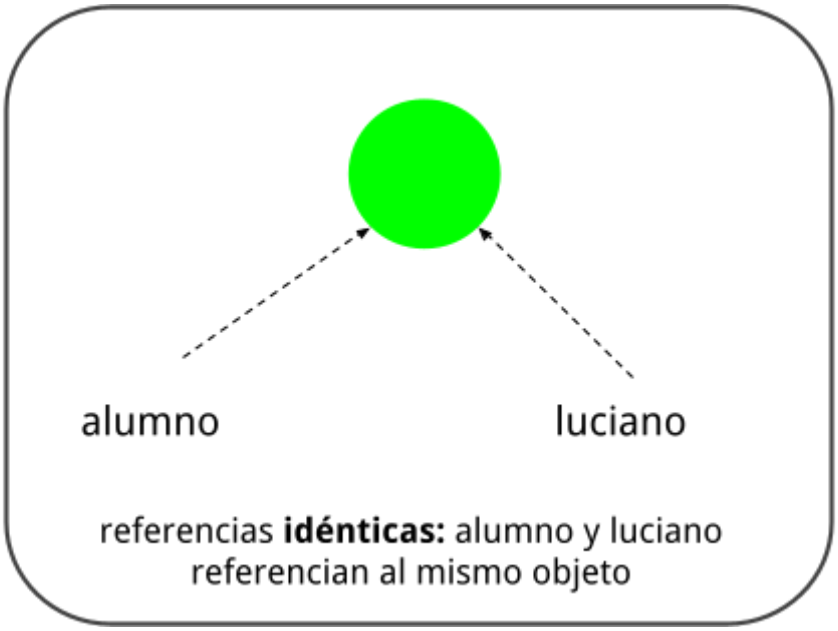


2 Igualdad e identidad

Otro concepto importante en el manejo de referencias es diferenciar la igualdad vs. la identidad.

2.1 Identidad

Si tenemos dos referencias idénticas, esto significa que están apuntando al mismo objeto.



Esto en Wollok se representa mediante el mensaje ===

Consideremos la clase Persona...[1]

```
class Persona {
	const property nombre
	const property domicilio
}
```

Nuestro primer test es sencillo: utilizando la asignación, ambas referencias apuntan al mismo objeto.

```
import personas.*

test "si asigno una referencia a otra ambas apuntan al mismo objeto"
{
```

[1] El ejemplo puede descargarse en https://github.com/wollok/igualdad-identidad-domicilios

```
    const alumno = new Persona(nombre = "luciano", domicilio =
"Centenera 83")
    const luciano = alumno
    assert.that(alumno === luciano)
}
```

2.2 Igualdad

En la mayoría de los casos estaremos bien con esta definición. Pero a veces tendremos que cambiar la estrategia para determinar si dos referencias están *representando* al mismo objeto, aun cuando no se trate exactamente del mismo objeto. Este concepto se llama **igualdad**.

Por defecto, dos referencias son iguales si son el mismo objeto. Pero esa definición está sujeta a cambios, si redefinimos el método *equals* o *==*. Por ejemplo, dos personas podrían ser iguales si tienen el mismo nombre.

Ahora bien

- ¿por qué tendría en el ambiente dos alumnos con el mismo legajo?
- ¿o dos personas con el mismo DNI?
- ¿o dos materias que se llamen igual?

Dado que trabajamos los objetos en un único ambiente, no parece una idea razonable: después de todo es probable que encontremos varias personas que se llamen igual pero no por eso representarán la misma persona. Codificamos entonces la clase Persona **sin redefinir el equals**.[2]

Ahora bien, una persona vive en un domicilio. ¿Cómo representamos esa abstracción? Mediante un String. Entonces tenemos a Chiara y Melina que viven en 'Nazca 3143'. Pero como los Strings los genera la VM, no puedo asegurar que los domicilios de ambas personas referencien exactamente a la **misma** instancia. Entonces cuando pregunto si Chiara y Melina viven juntas, no está bueno confiar en el ===, es preferible preguntar por equals.

Si definimos el siguiente comportamiento en Persona:

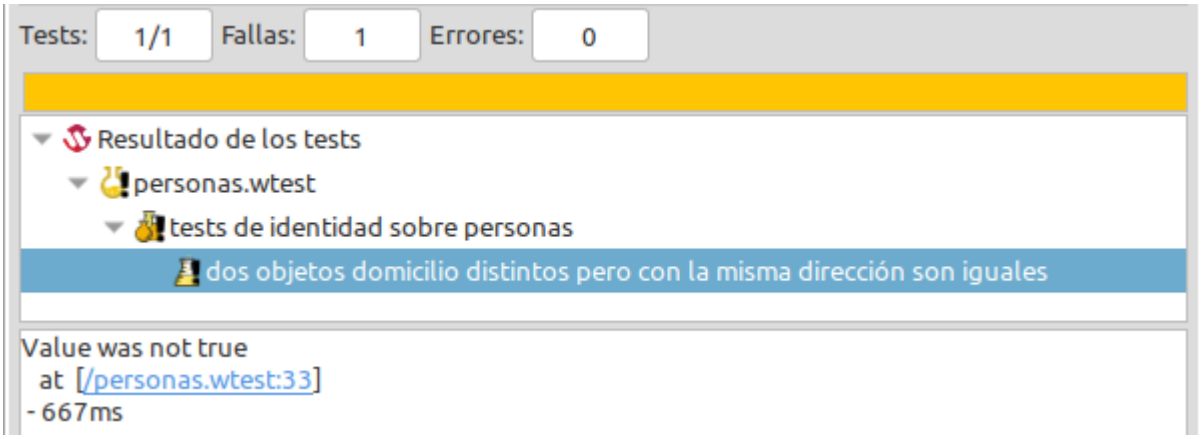
```
method vivenJuntosCon(_otraPersona) =
    domicilio === _otraPersona.domicilio()
```

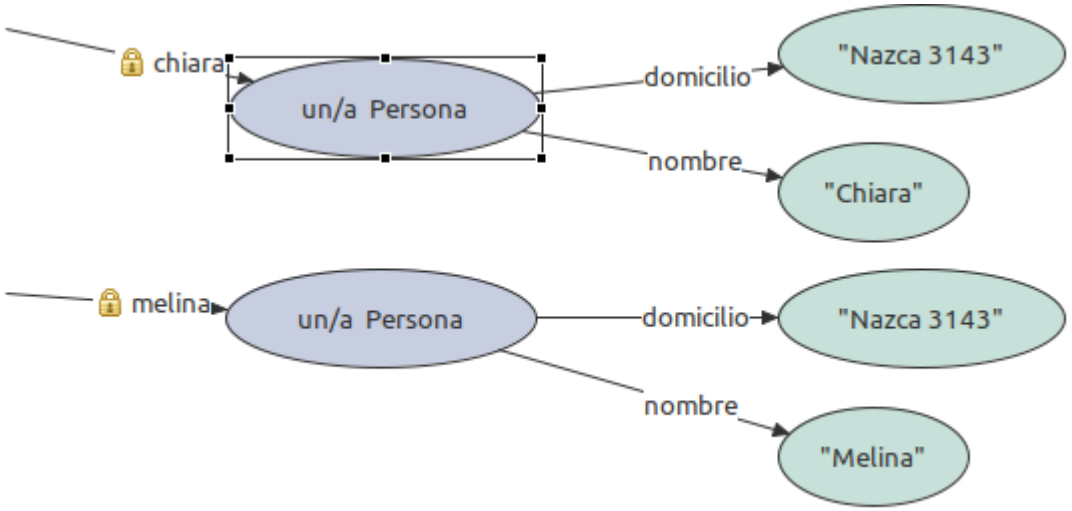
El siguiente test falla:

[2] Más adelante, cuando tenemos varios ambientes y necesitemos sincronizarlos entre sí, es posible que tengamos que cambiar nuestra perspectiva. Por el momento mantengamos la inocencia y la simpleza de nuestra solución.

```
test "dos objetos domicilio distintos pero con la misma dirección
son iguales" {
	const chiara = new Persona(nombre = "Chiara", direccion =
"Nazca 3143")
	const melina = new Persona(nombre = "Melina", direccion =
"Nazca 3143")
	assert.that(chiara.vivenJuntosCon(melina))
}
```

Gráficamente





Eso puede verse enviando este mensaje en el test anterior:

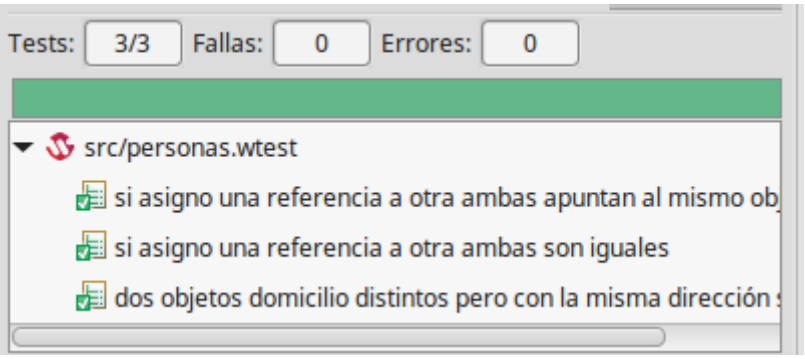
```
console.println(chiara.domicilio().identity()) // 2095573052
console.println(melina.domicilio().identity()) // 847018986
```

El mensaje identity() devuelve un número que podemos usar para diferenciar un objeto de otro en el ambiente.

Ahora bien, si reemplazamos el uso de === en la definición por ==, ya no dependemos de que exista un único String que represente al domicilio:

```
method vivenJuntosCon(_otraPersona) =
      domicilio == _otraPersona.domicilio()
```

Y el mismo test que antes fallaba, ahora pasa perfectamente...



En general, cuando estamos hablando de objetos de nuestro dominio (los que nosotros diseñamos) es lógico pensar que cada objeto va a representar un concepto dentro de mi ambiente: no habrá dos objetos representando al mismo alumn@, ni dos objetos para el mismo cliente, siempre y cuando yo tenga una aplicación con un único ambiente. En cambio, cuando tengamos números, strings, fechas, los *value objects* en general, tenemos que estar atentos a no esperar que se cumpla la identidad sino la igualdad. Entonces dos personas habrán nacido el mismo día si coinciden sus fechas de nacimiento, y dos empleados ganarán lo mismo si sus sueldos son iguales.

3 Resumen

Hemos visto que los objetos pueden ser mutables, en cuyo caso podemos cambiar las referencias, o ser inmutables (una vez construidos no pueden cambiar sus referencias, carecen así de efecto colateral). Esto nos permite compartirlos sin necesidad de estar cuidando las modificaciones en forma concurrente, y también facilita su testeo. Los objetos que modelan valores (como los strings, números, fechas, etc.) son buenos ejemplos de objetos que nos conviene que sean inmutables.

Cada objeto tiene una identidad que lo diferencia de los demás objetos del ambiente. En general preguntamos por referencias idénticas cuando estamos hablando de objetos de nuestro dominio, pero cuando tenemos strings, números, domicilios, puntos, booleanos no es conveniente preguntar si dos referencias apuntan exactamente al mismo *value object*, sino que hacemos la pregunta por igualdad (==, *equals*).

Paradigma Orientado a Objetos

Módulo 15: Elementos de Diseño.

por Fernando Dodino

Versión 2.1

Octubre 2017

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Elementos de Diseño
- 1.1 Componentes
- 1.2 Cohesión
- 1.3 Acoplamiento

2 Resumen

1 Elementos de Diseño

A lo largo de estos capítulos hemos incorporado algunas técnicas para el diseño en el paradigma de objetos. Repasaremos a continuación algunos de esos conceptos aprendidos. Primero que todo, señalemos que diseñar es

- encontrar componentes
- sus responsabilidades
- y la relación que cada componente tiene con los demás componentes para formar un sistema

1.1 Componentes

¿De qué hablamos cuando hablamos de componente?
Existen varios puntos de vista:

- un componente puede ser un objeto, o una clase
- o también puede ser un conjunto de objetos/clases que cumplen un objetivo. Ej: un alumno con sus notas modela la carrera académica de un alumno.
- a más alto nivel, un componente puede ser un módulo o un sistema que se comunica con otro módulo o sistema. Ej: el módulo de seguimiento de carrera académica y el módulo de inscripciones a los cursos.

Cuando decimos que un componente tiene responsabilidades, ¿cómo se marcan esas responsabilidades? En un objeto, a partir de los métodos que implementa.

1.2 Cohesión

- Una clase es cohesiva si podemos definirle un objetivo claro y puntual.
- Un método es cohesivo si tiene un único objetivo.

Emitir una factura y calcular el total de facturación está bueno que estén en diferentes métodos. En general, **separar los métodos con efecto colateral** (emitir factura, realizar un descuento, firmar una libreta de un alumno, cambiar el sueldo básico a un empleado) **y los métodos que no tengan efecto colateral** (conocer el sueldo de un empleado, saber el promedio de notas de un alumno en finales, conocer el total de facturación de un mes para un cliente, etc.) **es una buena práctica**. También es bueno abstraer ideas que se repiten en la misma clase dándole un nombre y dejándolo en un método aparte. Así por un lado evitamos duplicar código y por otro aumenta la cohesión de un método: se concentra en hacer sólo una cosa por vez.

Veamos este ejemplo que resuelve cuándo un alumno estudia para un parcial:

```
class Alumno {
      const notas = []
      var tipoAlumno = "E"
      /*
```

```
       * E = Estudioso, estudia siempre
       * V = Vago, estudia solo cuando le fue mal en el último
examen,
       * H = Hijo del rigor, estudia si el examen va a ser difícil
       * (o sea, si el parcial tiene más de 5 preguntas)
       */

      method rendir(nota) {
            notas.add(nota)
      }

      method estudiaPara(parcial) {
            if (tipoAlumno == "E") {
                  // Estudioso
                  return true
            }
            if (tipoAlumno == "V") {
                  // Vago
                  return notas.last() < 6
            }
            if (tipoAlumno == "H") {
                  // Hijo del rigor
                  return parcial.cantidadPreguntas() > 5
            }
            return false
      }
}

class Parcial {
      const property cantidadPreguntas
}
```

Consideremos en particular este método:

```
      method estudiaPara(_parcial) {
            if (tipoAlumno.equals("E")) {
                  // Estudioso
                  return true
            }
            if (tipoAlumno.equals("V")) {
                  // Vago
                  return notas.last() < 6
            }
            // es Hijo del rigor
            return _parcial.cantidadPreguntas() > 5
      }
```

Cada uno de los colores representa una responsabilidad que está resolviendo el alumno:

- sabe cuándo un alumno estudioso estudia
- sabe cuándo un alumno vago estudia
    - y sabe que el alumno tiene una lista de notas a la cual le puede pedir el último elemento
- sabe cuándo un alumno hijo del rigor estudia
    - y además sabe cuándo un parcial es difícil

Son en total 5 responsabilidades... ¿bastante, no? ¿Qué pasaría si pudiéramos trabajar con objetos polimórficos, que modelen el criterio de cuándo estudia un alumno? Pero además, si pensamos en mantener los objetos altamente cohesivos...

- quién sabe decirnos si un parcial es difícil: el parcial
- quién sabe decirnos la nota del último examen: un alumno
- quién sabe decirnos si un alumno estudiará para un parcial: el alumno... pero delegando al criterio que tiene ese alumno para estudiar

Construimos entonces la solución alternativa, en la que cada objeto tiene una responsabilidad clara y definida[1]:

```
class Estudioso {
	method estudiaPara(parcial, alumno) = true
}

class Vago {
	method estudiaPara(parcial, alumno) =
		alumno.notaUltimoExamen() < 6
}

class HijoDelRigor {
	method estudiaPara(parcial, alumno) = parcial.esDificil()
}

class Alumno {
	const notas = []
	var property criterioParaEstudiar = new Estudioso()

	method rendir(nota) {
		notas.add(nota)
	}

	method estudiaPara(parcial) =
		criterioParaEstudiar.estudiaPara(parcial, self)
```

[1] El ejemplo completo puede descargarse en https://github.com/wollok/cohesion-alumnos

```
    method notaUltimoExamen() = notas.last()
}

class Parcial {
    const property cantidadPreguntas

    method esDificil() = cantidadPreguntas > 5
}
```

El parcial se construye como un *value object* inmutable.

1.3 Acoplamiento

Es el grado en que los componentes de un sistema se conocen. En el ejemplo anterior, el criterio de estudio de un alumno vago podría haberse escrito de la siguiente manera[2]:

```
class Vago {
    method estudiaPara(parcial, alumno) =
        alumno.notas().last() < 6
}
```

Pero esto implica que el criterio Vago sabe que el alumno guarda una colección ordenada de notas. Si el Alumno cambia el tipo de colección de notas de List a Set la clase Vago se ve afectada porque **deja de funcionar** (Set no entiende el mensaje last() porque sus elementos no tienen orden).

Por eso, delegando bajamos el acoplamiento indeseado entre Alumno y Vago. Queremos que se conozcan, no buscamos acoplamiento cero, porque es necesario que ambas clases trabajen juntas para resolver el objetivo. Pero también queremos que ese acoplamiento sea bajo, mantenido a partir del envío del mensaje aproboUltimoExamen(), que es responsabilidad del alumno:

```
class Vago {
    method estudiaPara(parcial, alumno) =
        !alumno.aproboUltimoExamen()
}
```

De la misma manera, el alumno que estudia como hijo del rigor no necesita saber que el parcial tiene una cierta cantidad de preguntas que determina si es difícil o no, simplemente debe delegar la pregunta al parcial:

```
class HijoDelRigor {
    method estudiaPara(parcial, alumno) = parcial.esDificil()
```

[2] Asumiendo la existencia del getter de notas.

```
}
```

Entonces

- si la lógica que determina cuándo un parcial es difícil cambia, eso implica modificar la definición **en un solo lugar**.
- la dificultad de un parcial queda **reificado en un método,** eso facilita encontrarlo cuando lo necesite usar más adelante. Si hubiéramos optado por escribir el método de esta manera:

```
class HijoDelRigor {
     method estudiaPara(parcial, alumno) =
          parcial.cantidadPreguntas() > 5
}
```

¿De qué manera podemos asociar la pregunta del hijo del rigor al concepto “el parcial es difícil”? No queda claro en ningún momento y es muy difícil que podamos reutilizar esa idea cuando lo necesitemos.

2 Resumen

En este capítulo utilizamos el ejemplo para mostrar cómo el polimorfismo y la delegación nos ayudan a mantener alta la cohesión de los objetos y con un nivel de acoplamiento lo suficientemente bajo para ayudar a que nuestro sistema sea mantenible a largo plazo.

Paradigma
Orientado a Objetos

Módulo 16:
El proceso de diseño.

por Fernando Dodino
Versión 3.1
Noviembre 2020

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Enunciado

2 Objetivo de la clase

3 Qué es diseñar

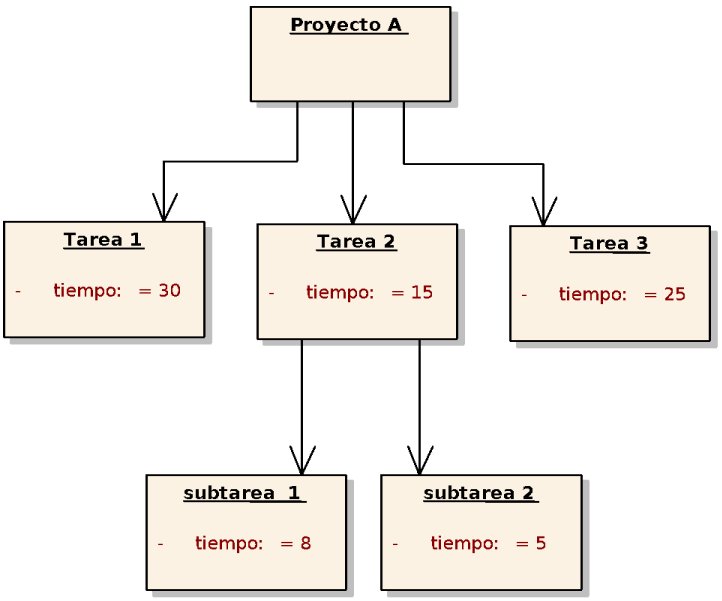
4 Algunas cuestiones sobre el diseño
- 4.1 El diseño en otros paradigmas
- 4.2 Cuánto diseñar
- 4.3 Diseño y diagramas

5 Manejo de proyectos
- 5.1 Por dónde arrancar
- 5.2 Objetos candidatos
- 5.3 Diagrama estático: primeros intentos
- 5.4 Identificando los requerimientos a resolver
- 5.5 Costo de una tarea
- 5.6 Agregando los impuestos
- 5.7 Costo total de un proyecto
- 5.8 Complejidad de una tarea
- 5.9 Resolviendo el costo de la complejidad

6 Design Patterns que aparecieron en la solución
- 6.1 Strategy
- 6.2 Composite

1 Enunciado

La compañía Kendari Corporation INC. desea desarrollar para sus sucursales internacionales una herramienta de manejo de proyectos.



Los proyectos se componen de tareas y las tareas de subtareas (las subtareas pueden tener a su vez subtareas). Cada una de estas tareas y subtareas tiene un costo en tiempo y en dinero según se muestra en la figura siguiente:

El costo de las tareas se calcula de acuerdo a la complejidad de la misma, a saber:

Complejidad mínima: tiempo * $25
Complejidad media: (tiempo * $25) + 5% de lo anterior
Complejidad máxima:

- si tiempo es menor o igual a 10 días --> tiempo * $25 + 7%
- si tiempo es mayor a 10 días --> (tiempo * $25) + 7% + $10 por cada día después del décimo

Nota: Para calcular el costo de una tarea sólo se utiliza el tiempo específico de la misma. Por ejemplo para calcular el costo de la tarea 2, el tiempo es a utilizar es 15.

A su vez las tareas que tengan más de 3 subtareas asociadas tienen un costo extra por overhead del 4%.

Por último hay un costo impositivo extra que dependerá de la reglamentación fiscal del país y algunos otros factores, lo importante es que existe el impuesto A (3% del valor) y el impuesto B (5% del valor). Hay tareas en donde aplican los dos impuestos, algunas que aplica sólo uno de ellos y otras que no se aplica ninguno.

Se debe poder determinar el porcentaje de completitud de cada tarea. Las tareas que no posean subtareas sólo pueden estar completas o no (al 0% o al 100% taxativamente). En cambio, en las tareas que posean subtareas se calculará en base al promedio ponderado de las mismas.

Nota: Los impuestos sólo se aplican sobre la tarea que los tiene, no sobre las subtareas.

Otro punto importante para el sistema es poder calcular el atraso posible en una tarea. Este dato depende exclusivamente de la complejidad y el tiempo de la misma y se calcula de la siguiente forma:

Tareas de complejidad mínima: 5 días.
Tareas de complejidad media: 10% del tiempo de la tarea.
Tareas de complejidad máxima: 20% del tiempo de la tarea + 8 días.

Aplicar el diseño necesario que permita:

- Obtener el costo de una tarea
- Obtener el costo total de un proyecto
- Obtener los días máximos de atraso de un proyecto
- Saber o asignar el porcentaje de completitud de una tarea

2 Objetivo de la clase

Esta clase tiene varios objetivos: la primera y principal es plantear algunas decisiones que no son triviales, **y son decisiones que tienen que ver específicamente con nuestro diseño**. En esta clase no nos importará tanto el resultado final de ese diseño, sino cómo es el proceso que me lleva a elegir alternativas.

De aquí se desprenden otros objetivos periféricos

- encontrar formas objetivas de comparar soluciones posibles y decidir en consecuencia
- repasar cómo la composición es una herramienta más flexible que la herencia
- definir cuándo elijo modelar con clases y cuándo con instancias
- encontrar una metodología iterativa que permita que me equivoque

entre otros que iremos descubriendo a lo largo del apunte.

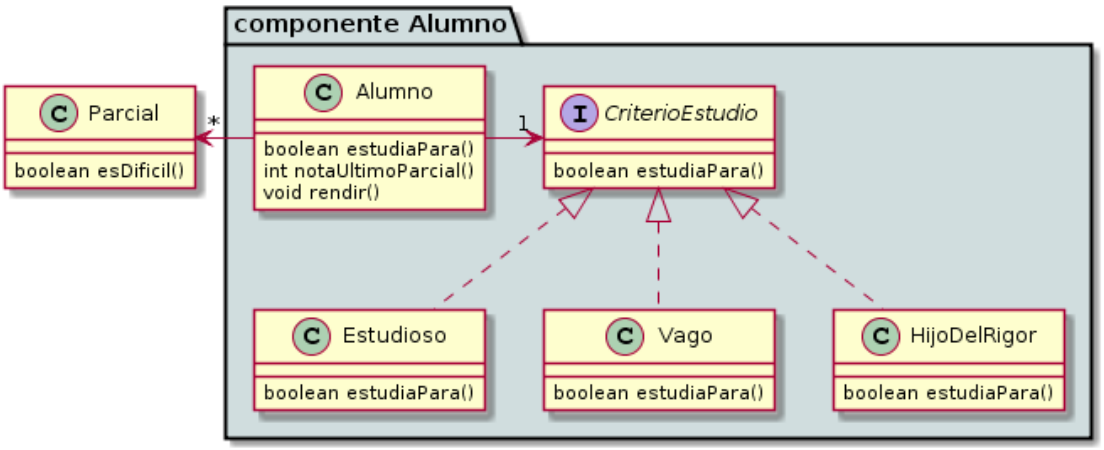
3 Qué es diseñar

Preguntamos a la clase qué es diseñar, salen respuestas que sin duda son valiosas. Nosotros vamos a quedarnos con algunas definiciones que nos sirven para disparar preguntas, antes que para estudiar de memoria para un examen:

Diseñar es

- encontrar los componentes
- qué hacen esos componentes
- cómo se relacionan

En particular, dentro del paradigma orientado a objetos, los componentes pueden ser objetos, o bien un conjunto de ellos que brindan servicios para resolver los requerimientos del usuario. Por ejemplo: cuando en el módulo anterior hablamos de un alumno y el criterio de estudio, podemos pensar en que ambos forman un único componente que nos sabe decir si un alumno va a estudiar para un parcial:



4 Algunas cuestiones sobre el diseño

4.1 El diseño en otros paradigmas

Vale recordar que tanto en lógico como en funcional hemos estado diseñando, solo que los componentes se construyen de diferente manera. Aquellos que deseen profundizar al respecto pueden leer el apunte Introducción al Diseño de Sistemas.

4.2 Cuánto diseñar

Algunas metodologías proponen pensar **todo** el diseño antes de programar, nosotros preferimos concentrarnos en algunas decisiones, y postergar otras. ¿Por qué postergo algunas decisiones?

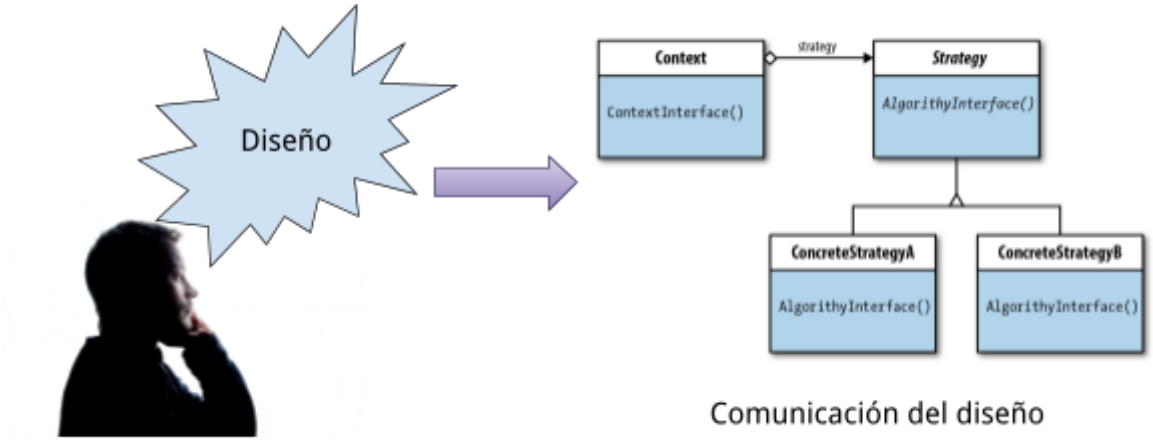
- No siempre tengo toda la información
- El usuario y el desarrollador van aprendiendo sobre el dominio mientras se construye la aplicación: evito así tomar decisiones prematuramente
- Los requerimientos cambian

Entonces cuando tengo alguna idea, la valido con algo de código. De esa manera puedo darme cuenta de si es implementable la idea o no. No tiene que ser todo el código, ni todas las alternativas posibles, pero sí algo que me permita saber si estoy yendo bien o no. Al no diseñar todo, es más fácil aceptar el error y volver a pensar una solución. El error es parte de la naturaleza humana, deberíamos adoptar una metodología que los tolere y no que los esconda.

4.3 Diseño y diagramas

Queremos dejar en claro que el diseño ocurre en la mente de las personas

cuando generan abstracciones, por lo tanto diseñar no es saber UML, en todo caso saber UML me permite comunicar el diseño. Puedo o no tener documentación formal, eso no excluye que el **diseño ya exista**.



5 Manejo de proyectos

Les damos 10/15 minutos para leer el enunciado.

5.1 Por dónde arrancar

Leer una serie de requerimientos puede ser un poco abrumador, pero a esta altura contamos con muchas herramientas que nos van a ayudar a resolver lo que nos piden.

Primero que nada, no hay que perder de vista los requerimientos (a veces en forma de caso de uso[1], a veces con el formato de *user story*), porque son los que deben guiar el proceso. ¿Qué necesitamos resolver?

- el costo de una tarea
- el costo de un proyecto
- la cantidad máxima de días de atraso de un proyecto
- asignar o conocer la completitud de un proyecto

Es importante concentrarse en lo que tenemos que resolver para no hacer cosas de más.

Ahora bien, para poder calcular el costo de una tarea deben intervenir muchos componentes, entonces para poderlos encontrar tenemos a mano varias opciones:

- Diagrama de clases (o estático)

[1] Para más información se puede leer el apunte Las entradas del diseño

- Diagrama de objetos
- Buscar objetos **candidatos** (no todos van a quedar) y responsabilidades que pueden cumplir
- Pensar en cómo lo pruebo: escribir un test primero (técnica que se conoce como TDD)
- O bien podría ser que imagine una clase o un objeto tarea y empiece a tirar código, que es una posibilidad más para iniciar nuestro diseño. Lo importante es saber frenar en algún momento y levantar la cabeza para que no se transforme en una actividad “code & fix”.

Tener varias opciones nos permite comenzar por cualquiera de las variantes que mejor nos parezca, o con la que nos sintamos más cómodos.

Cuando me trabo... cambio la perspectiva del problema:

1. si estoy haciendo un diagrama estático, podría tirar código (en otra situación podría ser tratar de escribir un test o pensar un diagrama de objetos)
2. pregunto al que entiende del negocio, o
3. no tomo este camino y postergo la decisión.

De esa manera, mis herramientas para diseñar y la metodología que adopto permiten que cuando me trabe pueda seguir avanzando con mi solución.

Nosotros vamos a escribir una lista de objetos candidatos.

5.2 Objetos candidatos

Armamos una lista tentativa, no necesariamente todos van a quedar:

- Proyecto
- Tarea
- Subtarea
- Complejidad
- Impuesto

5.3 Diagrama estático: primeros intentos

Para resolver el costo de una tarea vemos que eso depende

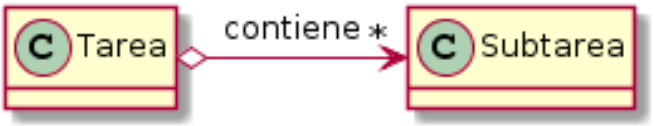
- de si la tarea tiene subtareas
- de la complejidad de la tarea (mínima, media o simple)
- de los impuestos que tenga una tarea

A su vez, veamos lo que sucede con el porcentaje de completitud:

- las tareas que no tienen subtareas se pueden marcar como cumplidas, las tareas que tienen subtareas no
- el % de completitud de una tarea se calcula distinto para una tarea que tiene subtareas y para las que no tienen subtareas

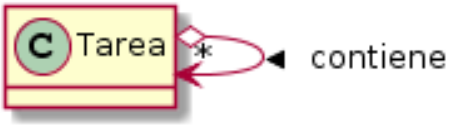
Entonces, una decisión importante a resolver es ¿cómo represento a las tareas y las subtareas?

Opción 1



Mmm... no me sirve porque no permite que una subtarea tenga subtareas.

Opción 2: una tarea tiene una colección de objetos tarea (referencia recursiva). Si la colección está vacía, la tarea no tiene subtareas.

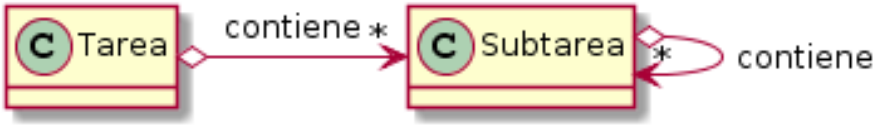


Si bien es una solución flexible, nos obliga a preguntar si la tarea tiene subtareas en tres lugares

1. para determinar el % de completitud,
2. para asignar el % de completitud
3. e incluso el % de costo extra depende de que la tarea tenga subtareas

Entonces la tarea es un objeto que tiene **baja cohesión**: representa a las tareas que tienen y las que no tienen subtareas.

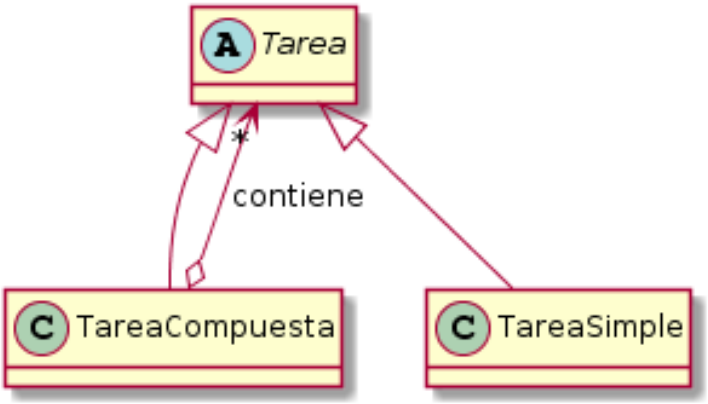
Opción 3



Esta solución tiene dos desventajas: es la más compleja de las tres y además no permite reutilizar conceptos que a priori parecen similares como la tarea y la subtarea. Tampoco discrimina las tareas que tienen y las que no tienen subtareas.

En realidad, lo que pasa es que el enunciado tiene una trampa de redacción: lo importante no es diferenciar la tarea de la subtarea, sino discriminar las tareas que tienen subtareas vs. las tareas que no tienen subtareas (como estuvimos escribiendo en las dos páginas anteriores, vuelvan a releer el apunte si no nos creen)

Opción 4



Al subclasificar la tarea le puse un nombre mejor (encontré una abstracción).

Pero no olvidemos que para calcular el costo y el tiempo máximo de atraso de una tarea hay comportamiento diferencial para las tareas de complejidad mínima, media y máxima. Entonces si pensaba subclasificar la tarea en TareaComplejidadMinima, TareaComplejidadMedia y TareaComplejidadMaxima tengo un conflicto, porque la herencia solo permite armar taxonomías (clasificaciones) por un único punto de vista.

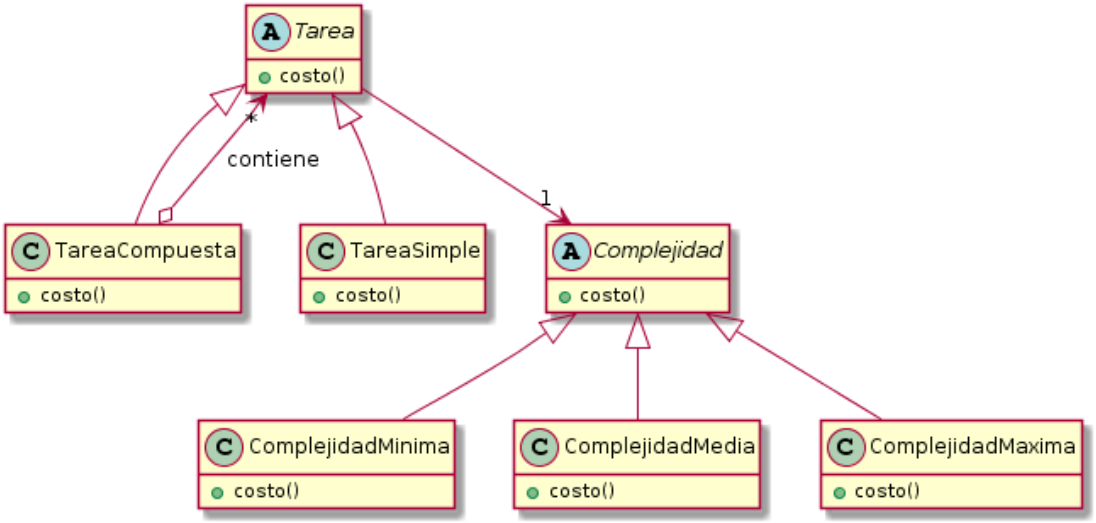
Separamos dos tipos de clasificación posibles para tareas:

- Tarea simple y compuesta
- Tarea de complejidad mínima, media y máxima

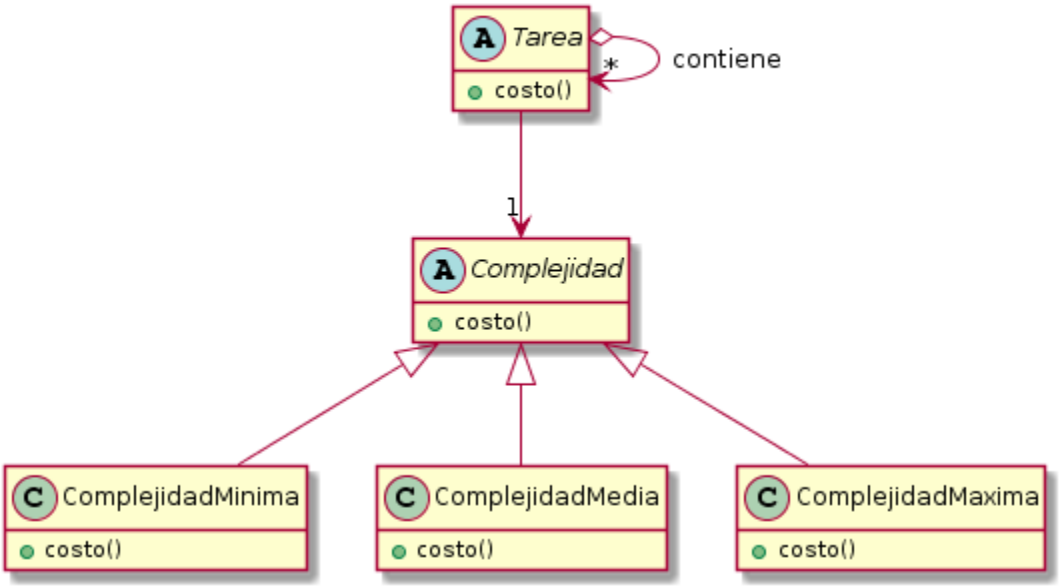
Si ambas jerarquías no pueden convivir juntas, recordemos que una alternativa a la herencia es la **composición**, entonces tenemos diferentes posibilidades:

Opción 1

Según esta solución nos interesa separar la tarea de su complejidad. La consecuencia es que es costoso hacer que una tarea simple devenga en compuesta, mientras que cambiar la complejidad de una tarea es simplemente intercambiar el objeto que representa la complejidad.



Opción 2



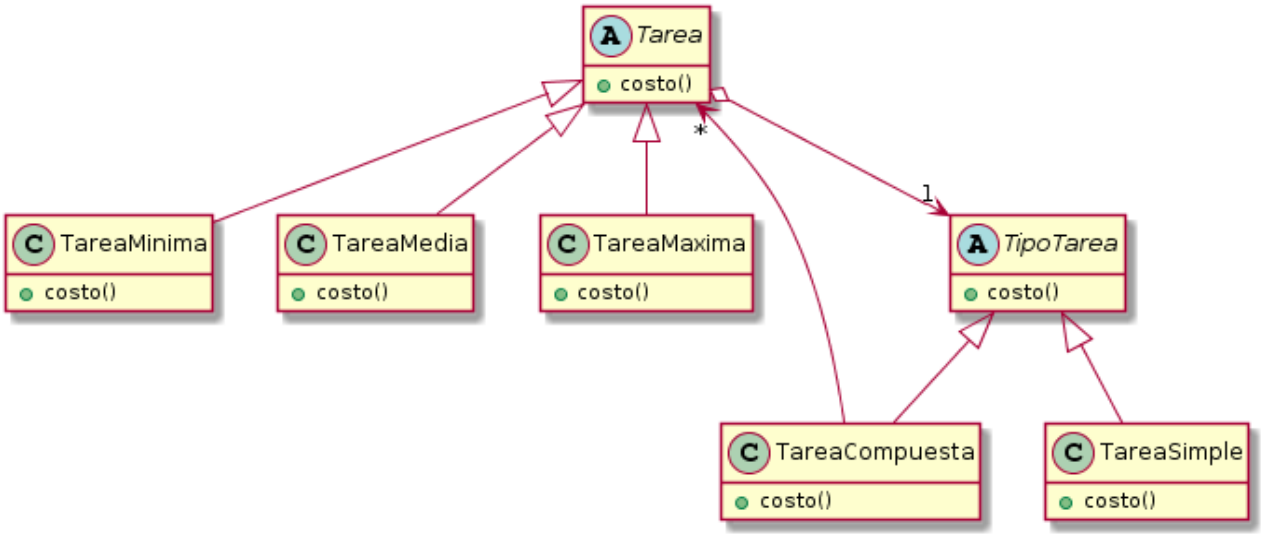
Según esta solución es posible que una tarea de complejidad mínima pase a ser de complejidad media. Por otra parte si en muchos lugares se discrimina el comportamiento entre tareas simples y compuestas, habrá muchos métodos que pregunten con un if si la tarea es simple o compuesta. Podemos reificar la decisión en un método esSimple():

```
if (self.esSimple())
     <-- se abstrae en un método el concepto simple
     ... comportamiento de tarea simple...
} else {
     ... comportamiento de tarea compuesta...
```

}

Aun así la Tarea debe contemplar ambos casos y pierde cohesión (sabe hacer las cosas de la tarea simple y de la compuesta también).

Opción 3



Con esta solución es más fácil pasar una tarea simple a compuesta que cambiar la complejidad de una tarea.

¿Qué hacemos entonces?

No puedo decidir yo, pero **el proceso de diseño me ayuda a hacer una pregunta valiosa al usuario/analista**. El usuario dice: “las tareas podrían cambiar su complejidad, es decir, una tarea fácil podría pasar a ser compleja, en cambio una tarea simple no se transforma en compuesta ni viceversa”.

En base a este último requerimiento, es más fácil subclasificar la tarea en simple y compuesta y delegar en otro objeto la complejidad, nos decidimos por la primera opción. A esta altura desaparece la idea de subtarea (queda como objeto candidato nomás).

¿Qué falta?

1. calcular el costo (de una tarea y de un proyecto), para lo cual debemos
    a. resolver el costo extra por cantidad de subtareas
    b. resolver el costo impositivo
2. atacar la completitud del proyecto

Toda esta lista de tareas pendientes las anotamos en el pizarrón, en un papel,

en un archivo de texto plano, abrimos un issue en Github o una tarjeta en Trello, como quieran. Lo importante es saber qué cosas tenemos que encarar.

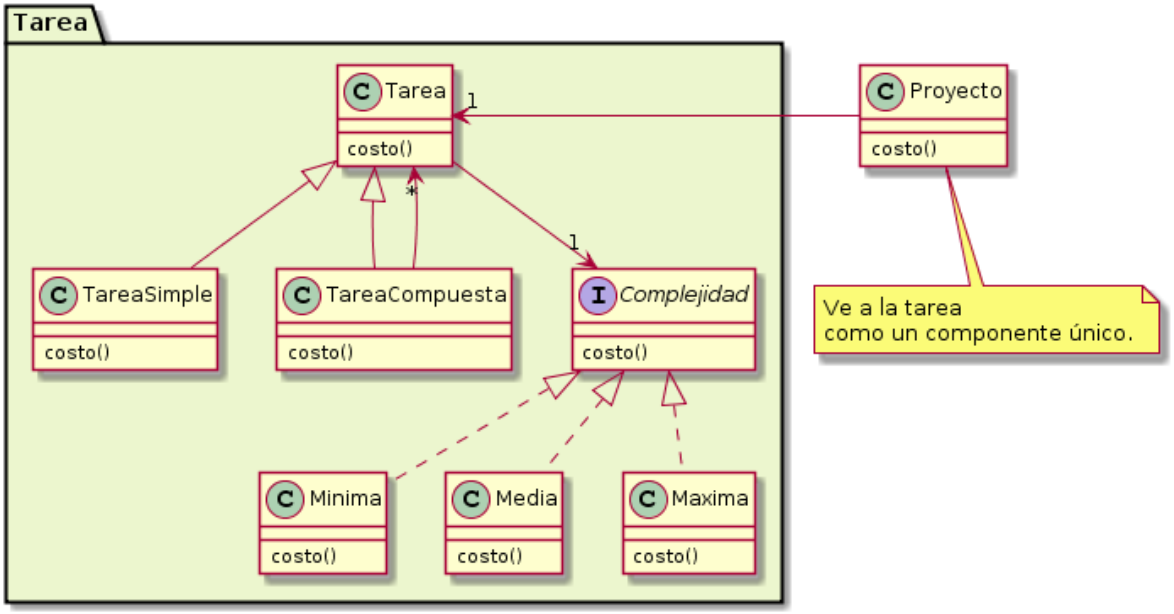
5.4 Identificando los requerimientos a resolver

Cada uno se resuelve enviando un mensaje a un objeto. Ok, a qué objeto le mando qué mensaje

- obtener el costo de una tarea ⇨ a una tarea.costo()
- obtener el costo total de un proyecto ⇨ a un proyecto.costoTotal()
- asignar % de completitud a tarea simple ⇨ a una tareaSimple.porcentajeCompletitud(x)
- obtener % de completitud de un proyecto ⇨ a un proyecto.porcentajeCompletitud()
- obtener el tiempo total de un proyecto ⇨ a un proyecto.tiempoTotal()
- obtener los días máximos de atraso de un proyecto ⇨ a un proyecto.diasMaximoDeAtraso()

En este ejercicio todo parece fluir normalmente... hasta que aparece el caso típico del siguiente requerimiento: “Quiero saber el saldo de un cliente”. ¿A quién le pregunto eso? Podría parecer "poco natural" preguntárselo al cliente, ya que en la realidad yo no le pregunto al cliente. El tema es que objetos trabaja con un modelo que es una simplificación de la realidad. El cliente que yo represento como objeto no es igual al cliente físico, es una abstracción para mi sistema, y está bien que en esa abstracción el saldo sea un comportamiento de mi cliente que vive en un ambiente de objetos, y no tiene nada que ver con el cliente físico[2].

[2] Para más información ver el apunte Guías para comunicar un diseño



Tenemos componentes claramente delimitados por el rectángulo verde. No hablamos de componente a nivel objeto, sino que un componente agrupa a varios objetos que cumplen una funcionalidad (todos los objetos incluidos en el rectángulo me ayudan a saber el costo de una tarea).

¿Para qué sirve pensar en componentes? Si hacemos cambios sin cambiar la interfaz de Tarea, el Proyecto no debería verse afectado. Y acá hablamos de interfaz como la parte visible / los servicios / lo que me puede contestar un objeto.

Ahora sí vamos al código para calcular el costo.

5.5 Costo de una tarea

```
class Tarea {
  method costo() = ...
}
```

Si recordamos que una tarea depende

- de la complejidad (el costo base)
- del costo por overhead (para subtareas)
- de los impuestos

Podemos pedir que el costo tenga esas tres abstracciones:

```
method costo() = self.costoBase() + self.costoPorOverhead() +
      self.costoImpositivo()
```

El costoPorOverhead

- se puede definir como abstracto en tarea y redefinirlo para tarea simple o compuesta. Esto tiene como consecuencia que nuevas subclases de tarea están forzados a definir su comportamiento y no toman un valor por defecto, lo cual puede ser una ventaja o una desventaja.
- o bien se puede definir por defecto 0 en tarea y redefinirlo en tarea compuesta. Esto permitiría que Tarea pueda instanciarse, algo que deberíamos preguntar si es lo que queremos.

Decidimos que Tarea sea una clase abstracta y que el método costoPorOverhead() sea abstracto en Tarea:

```
method costoPorOverhead()
```

En TareaSimple se redefine para que devuelva 0

```
class TareaSimple inherits Tarea {
      override method costoPorOverhead() = 0
}
```

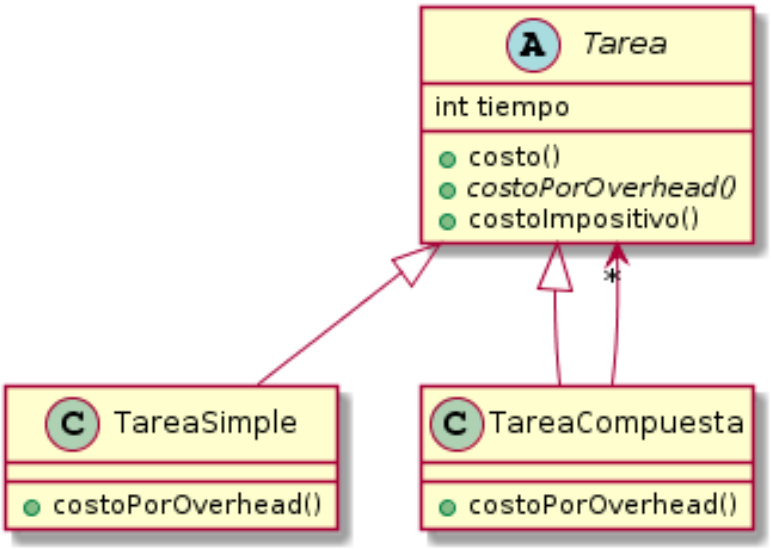
y en Compuesta preguntando si las tareas son más de 3. La desventaja de esta solución es que tenemos que calcular dos veces el costo base (podemos almacenar ese valor y pasarlo como parámetro).

```
class TareaCompuesta inherits Tarea {
      override method costoPorOverhead() =
            if (self.tieneMuchasSubtareas())
                  self.costoBase() * 0.04
            else
                  0
}
```

Fíjense que elegimos delegar en otro método la decisión de “tener muchas subtareas” lo que permite que lo utilicemos en otro contexto:

```
class TareaCompuesta inherits Tarea {
      const subtareas = []

      method tieneMuchasSubtareas() = subtareas.size() > 3

      ...
```

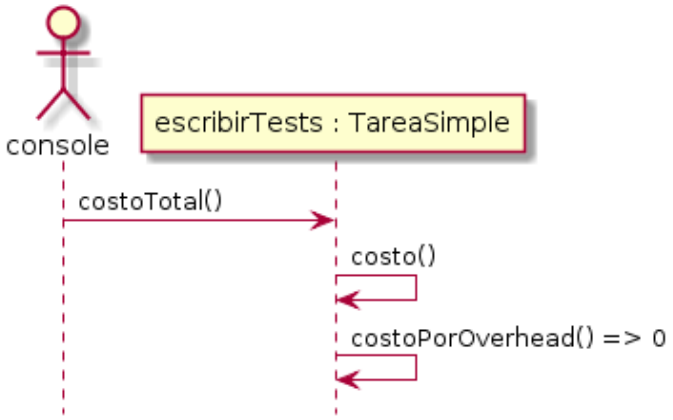
Y aun cuando todavía no tenemos implementado el costo, queremos contar una idea interesante que acabamos de implementar, el patrón **Template Method,** que consiste en definir una responsabilidad (el costo) en una clase abstracta que delega parte de la resolución en cada una de las subclases (costo por overhead).

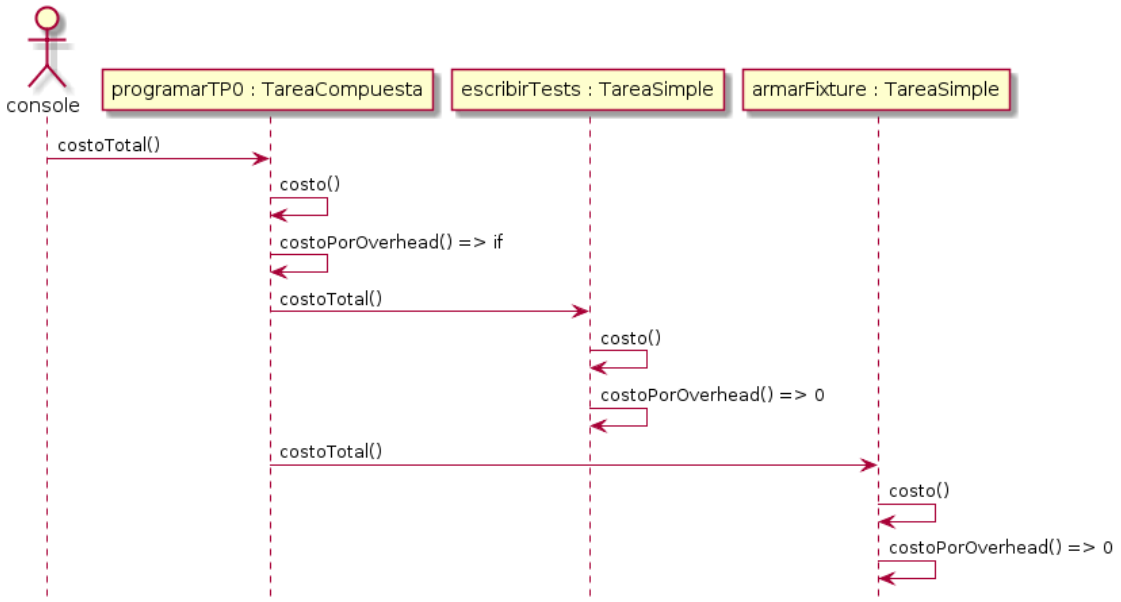


Si tenemos una tarea simple y una compuesta:

```
>>> const escribirTests = new TareaSimple()
>>> escribirTests.costoTotal()
>>> const programarTP0 = new TareaCompuesta()
>>> const armarFixture = new TareaSimple()
>>> programarTP0.agregarTarea(armarFixture)
>>> programarTP0.agregarTarea(escribirTests)
>>> programarTP0.costoTotal()
```

Podemos ver cómo se resuelve el método costoPorOverhead para *escribirTests* y para *programarTP0*:

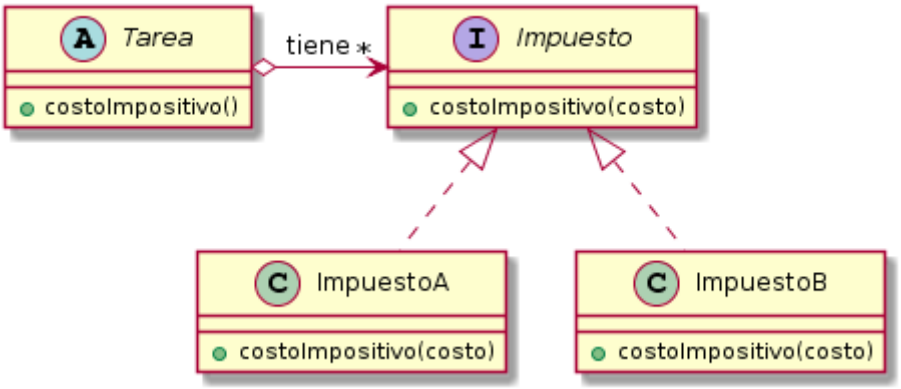




5.6 Agregando los impuestos

Tenemos distintos impuestos que pueden aplicar sobre una tarea, podemos considerar que una solución posible sería:

Opción 1: *los impuestos como clases*

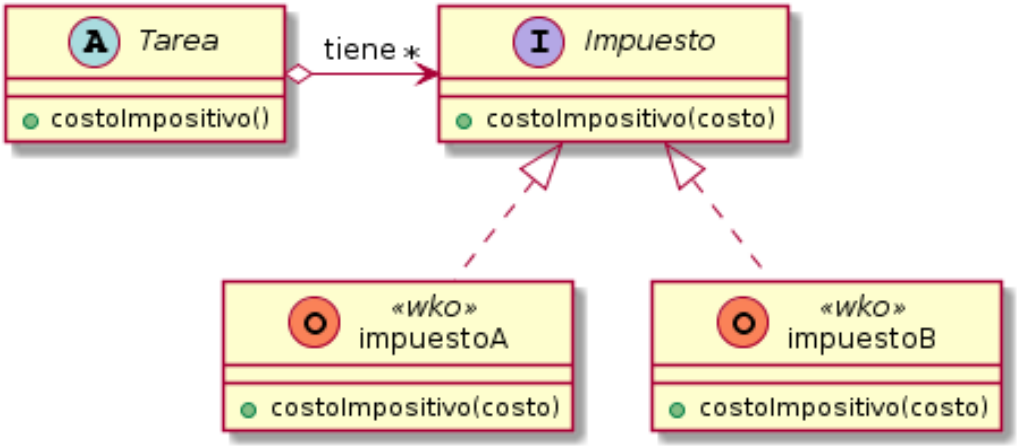


Ahora, ¿en qué se diferencia cada impuesto en particular?
Y… en el impuesto A es un 3% y el impuesto B es un 5%. Es decir, si codificamos el impuesto A y el B, solo se diferencian en el porcentaje a descontar.

```
class ImpuestoA {
      method costo(costo) = costo * 0.03
}

class ImpuestoB {
      method costo(costo) = costo * 0.05
}
```

En esta solución no existe la necesidad de modelar una superclase Impuesto, por lo que ImpuestoA e ImpuestoB comparten una interfaz común, que en Wollok no se codifica.

Opción 2: los impuestos como wko



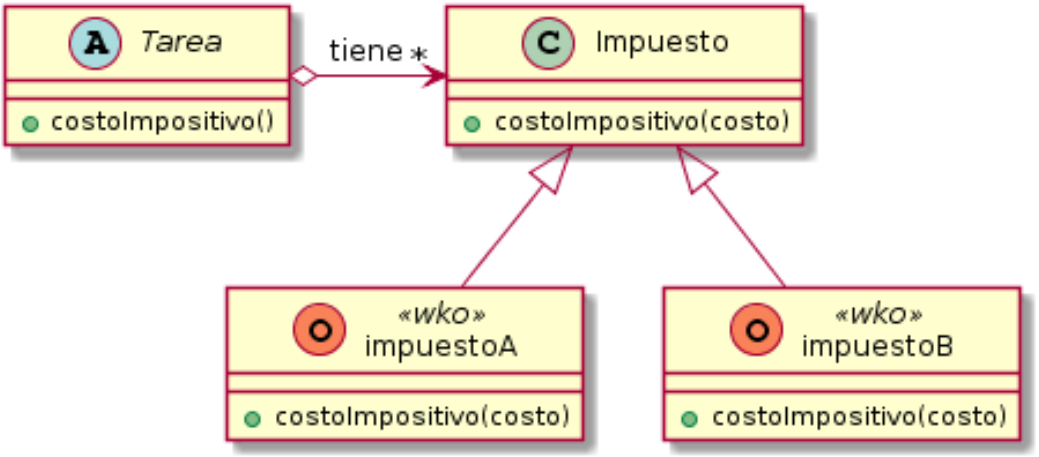
Una segunda opción en Wollok podría ser modelar el impuesto A y B como wko. En cuanto a código, no es mucho el cambio, simplemente los class pasan a objects:

```
object impuestoA {
      method costo(costo) = costo * 0.03
}

object impuestoB {
      method costo(costo) = costo * 0.05
}
```

La diferencia está en el uso:

- el impuesto A y B se pueden acceder globalmente
- también se pueden compartir entre varias tareas, ya que no tienen estado (no tienen referencias específicas de una tarea). Como consecuencia de esto, puedo tener varias tareas apuntando a dos impuestos A y B.
- aun así, estoy modelando dos *named objects* que internamente comparten un comportamiento similar.

Opción 3: los impuestos como *named objects (wko)* heredando de una superclase común



Si queremos evitar la repetición de la idea, podríamos decir en un solo lugar cómo es el cálculo, y tener dos wko específicos para el impuesto A y el B.

```
class Impuesto {
      const porcentaje

      method costo(costo) = costo * porcentaje / 100
}

object impuestoA inherits Impuesto(porcentaje = 3) {}
object impuestoB inherits Impuesto(porcentaje = 5) {}
```

Aquí tenemos algunas ventajas:

- el comportamiento del impuesto se define una única vez, y es posible redefinirlo en cada impuesto
- los impuestos A y B se acceden globalmente, como en el ejemplo anterior

Una variante mejor es pensar en cada impuesto como una constante:

```
const impuestoA = new Impuesto(porcentaje = 3)
const impuestoB = new Impuesto(porcentaje = 5)
```

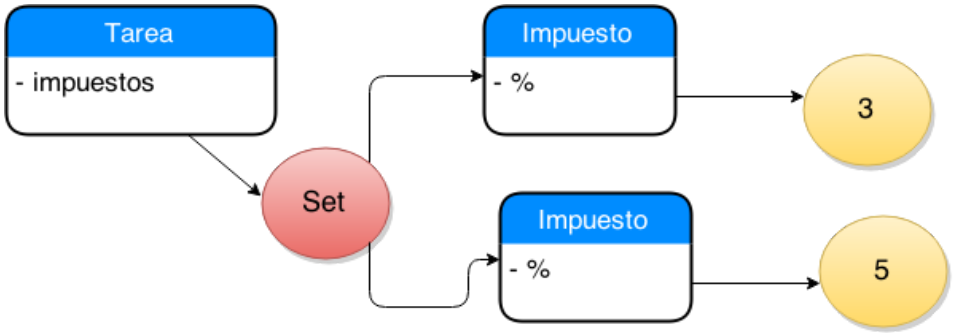
lo que permite conservar las referencias globales para impuestoA / impuestoB **pero comunicando que no hay comportamiento diferencial entre los impuestos**, ya que son simplemente instancias diferentes de Impuesto.

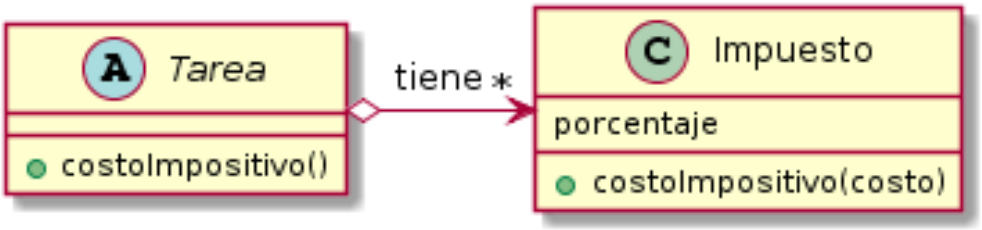
Ya sea porque vayamos por la opción de los objetos autodefinidos (wko) o las constantes, esta solución no escala cuando la cantidad de impuestos empieza a crecer, si lo único que varía es el % que se aplica sobre el costo base.

Opción 4: Tener una clase impuesto y manejarla con diferentes instancias

Si no hay comportamiento diferencial, podemos manejarnos con instancias a secas. Hay dos objetos: un objeto Impuesto que representa al impuesto A y otro objeto Impuesto que representa al impuesto B.

Entonces, cada tarea puede tener 0, 1 ó 2 objetos que van a pertenecer a la misma clase:






```
class Impuesto {
      const porcentaje

      method costo(costo) = costo * porcentaje / 100
}
```

La ventaja de esta solución: puedo tener una gran cantidad de impuestos y solo necesito instanciar cada uno de ellos. La desventaja: podría perder la abstracción “impuesto A” e “impuesto B” que dentro de la solución podía ser importante.

Ya sabemos al menos que tenemos una pregunta para disparar al usuario: ¿cuántos impuestos surgieron en los últimos años? ¿hubo impuestos basados en otro cálculo? ¿siempre hubo dos impuestos A y B? Esa tarea de relevamiento permitirá que estemos más seguros de nuestra decisión, mientras tanto nos inclinamos por la cuarta opción, ya que es un modelo simple y no nos interesa por el momento identificar los impuestos A o B (no se pretende en el apunte

abordar la carga de los impuestos de una tarea).

Ahora sí, para calcular el costo impositivo de una tarea hacemos:

```
class Tarea {
	const impuestos = #{}

	method costoImpositivo() = impuestos.sum { impuesto =>
		impuesto.costo(self.costoBase())
	}
```

Claro, eso fuerza a que por cada impuesto estamos calculando el costo de la complejidad. Podemos evitar eso mediante una variable local:

```
method costoImpositivo() {
	const costoBase = self.costoBase()
	return impuestos.sum { impuesto =>
		impuesto.costo(costoBase)
	}
}
```

5.7 Costo total de un proyecto

El costo total de un proyecto es la sumatoria de los costos de todas las tareas asociadas.

```
class Proyecto {
	const tareas = []

	method costoTotal() = tareas.sum {
		tarea => tarea.costoTotal()
	}
}
```

En la tarea simple el costo total es el costo a secas:

```
>>TareaSimple
method costoTotal() = self.costo()
```

Y en las tareas compuestas es el costo de dicha tarea más la sumatoria de los costos de cada subtarea:

```
>>TareaCompuesta
```

```
method costoTotal() = self.costo() + subtareas.sum {
      tarea => tarea.costoTotal()
}
```

Vemos que la codificación del método para el proyecto y la tarea compuesta es el mismo. Ojo con cambiar **todo** el diseño porque hay en un lugar código que es "parecido" pero no "igual" (porque representan distintos conceptos).

Podríamos tener una tarea raíz en proyecto y que para calcular el costo total se delegue directamente a la tarea raíz:

```
class Proyecto {
       var tareaRaiz

       method costoTotal() = tareaRaiz.costoTotal()
}
```

Cada vez que planteamos un modelo, existe el riesgo de caer en el **sobrediseño**, hacer cosas de más que luego nunca se utilizan (el problema es que esto tiene un costo, además la aplicación se vuelve más compleja y por lo tanto más difícil de mantener). La idea de trabajar en forma **iterativa**, donde vamos explorando ideas de diseño y tirando código permite comenzar con un problema de baja complejidad e incrementarla a medida que uno gana confianza.

5.8 Complejidad de una tarea

La idea de separar a la tarea de su complejidad en otro objeto resulta interesante para remarcar algunas cosas:

- en la mayoría de los lenguajes OO la herencia ofrece una sola chance para categorizar, y ya decidimos que era más importante para nosotros distinguir a las tareas simples de las compuestas
- si elijo tener un `int` (0 para tareas de mínima complejidad, 1 para tareas de complejidad media y 2 para tareas de complejidad máxima), voy a tener un if cuando calcule el costo base de una tarea:

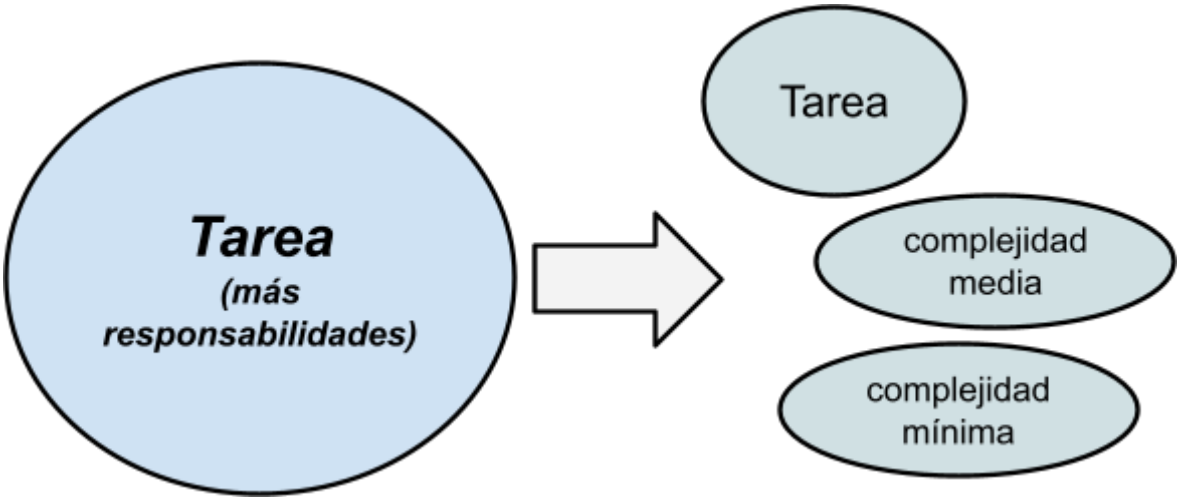
```
method costoBase() {
      if (complejidad == 1) { // minima
            ...
      }
      if (complejidad == 2) { // media
            ...
      }
```

```
        if (complejidad == 3) { // maxima
            ...
        }
}
```

Cuando necesite conocer los días máximos de atraso de un proyecto, voy a enviar el mensaje diasMaximosDeAtraso() al proyecto, que a su vez va a sumarizar los días máximos de atraso de una tarea. Una vez más, tengo que poner un if múltiple:

```
method diasMaximosDeAtraso() {
        if (complejidad == 1) { // minima
            ...
        }
        if (complejidad == 2) { // media
            ...
        }
        if (complejidad == 3) { // maxima
            ...
        }
}
```

Lo más criticable de esta solución es que tengo dos lugares donde estoy tomando la misma decisión. No solo molesta si aparece la complejidad crítica y hay que agregar un if más y recompilar, molesta sobre todo que estoy escribiendo 1, 2 y 3 en dos lugares distintos que tienen que estar sincronizados. Y además tengo que cuidarme de no exponer la complejidad de la tarea hacia afuera, para que esos números mágicos 1, 2 y 3 no estén diseminados por toda la aplicación. Entonces esa es la otra ventaja de separar tarea y complejidad:



```
>>Tarea
method costoBase() = complejidad.costo(self)
```

Discusiones de diseño que nos interesa tener: la complejidad, ¿debería ser clase abstracta o interface? ¿De qué depende?

- Si hay comportamiento en común (como es el caso, se repite la idea tiempo * $ 25), la clase abstracta nos permite no repetir código.
- Cuando no hay comportamiento común, una interfaz que defina los mensajes que deben entender las clases que la implementen alcanza. La interfaz es una abstracción de diseño, en el caso de Wollok no termina implementándose ninguna construcción.

El método costo() en cada subclase de Complejidad, le debe pasar ¿una tarea o sólo el tiempo? Hay argumentos a favor de uno y otro:

```
method costoBase() = complejidad.costo(self)
```

al pasar la tarea, la complejidad puede enviar más mensajes que tarea.tiempo(). A priori esto parece generar un acoplamiento mayor. Por otra parte hoy el costo de la complejidad depende solo del tiempo de la tarea: si los requerimientos cambian y necesitamos alguna información extra, eso no implicará un cambio en la interfaz del método costo()

```
method costoBase() = complejidad.costo(tiempo)
```

al pasar la referencia *tiempo* a la complejidad, la Tarea sabe que el costo de la complejidad se determina en base a su tiempo, de manera que hay un grado de acoplamiento o conocimiento de la Tarea respecto a lo que hace la complejidad. Además si los requerimientos cambian y para calcular el costo se necesita más información de la tarea, eso requerirá cambiar la interfaz del método costo(tiempo). Por otra parte, la complejidad no recibe una referencia a la tarea, esto evita mandarle mensajes adicionales (en alguna medida baja el acoplamiento).

5.9 Resolviendo el costo de la complejidad

- El costo de la complejidad mínima se calcula como tiempo * 25 .
- El costo de la complejidad media se calcula como tiempo * 25 + 5%.
- El costo de la complejidad máxima se calcula como tiempo * 25 + 7% + $ 10 a partir del décimo día.

En todos los casos se repite la misma idea: tiempo * 25, entonces podemos modelar eso en un único lugar: una superclase Complejidad. Por otra parte si

pensamos que en la complejidad mínima
`tiempo * 25` es igual a
`tiempo * 25 + 0`, podemos nuevamente tener un **Template method**. La superclase Complejidad define el comportamiento base, y cada subclase implementa el valor adicional que hay que sumarle al costo final, que depende en algunos casos del costo base y en otros del tiempo:

```
class Complejidad {

  method costo(tarea) {
    const costoBase = tarea.tiempo() * 25
    return costoBase + self.costoExtra(tarea.tiempo(),costoBase)
  }

  method costoExtra(tiempo, costoBase)
}
```

En este caso tenemos:

- *costo(tarea)* como **template method**, el método definido en la superclase
- *costoExtra(tiempo, costoBase)*: el **método hook** o **primitiva** que están obligados a definir las subclases de Complejidad

La complejidad mínima define un método *hook* de compromiso (que nuevamente, podría haber definido la superclase Complejidad):

```
class ComplejidadMinima inherits Complejidad {
      override method costoExtra(tiempo, costoBase) = 0
}
```

La complejidad media considera el 5% del costo base:

```
class ComplejidadMedia inherits Complejidad {
      override method costoExtra(tiempo, costoBase) = 0.05 *
costoBase
}
```

Y el cálculo de la complejidad máxima es el más molesto, se puede resolver considerando 7% del costo base, y 10 $ a partir del décimo día, lo que es lo mismo decir 10 $ * (el máximo entre 0 y la cantidad de días que pasaron a partir del décimo día):

```
class ComplejidadMaxima inherits Complejidad {
      override method costoExtra(tiempo, costoBase) =
            costoBase * 0.07 +
            self.costoAdicionalPorRetraso(tiempo)
```

```
    method costoAdicionalPorRetraso(tiempo) =
        10 * 0.max(tiempo - 10)
}
```

Otra forma de resolverlo:

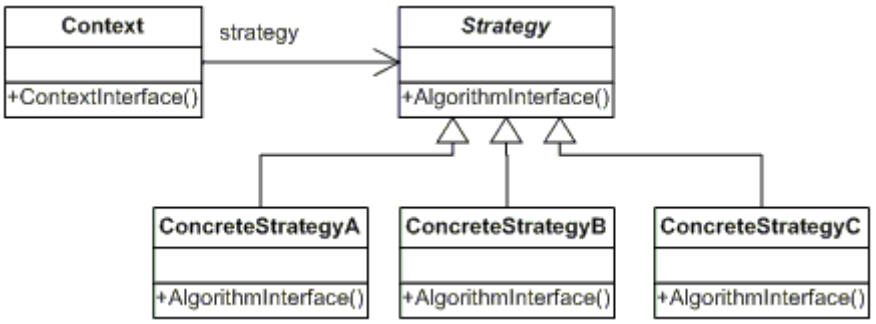
```
class ComplejidadMaxima inherits Complejidad {
    method costoExtra(tiempo, costoBase) =
        costoBase * 0.07 +
        self.costoAdicionalPorRetraso(tiempo)

    method costoAdicionalPorRetraso(tiempo) =
        if (tiempo <= 10)
            0
        else
            10 * (tiempo - 10)
}
```

6 Design Patterns que aparecieron en la solución

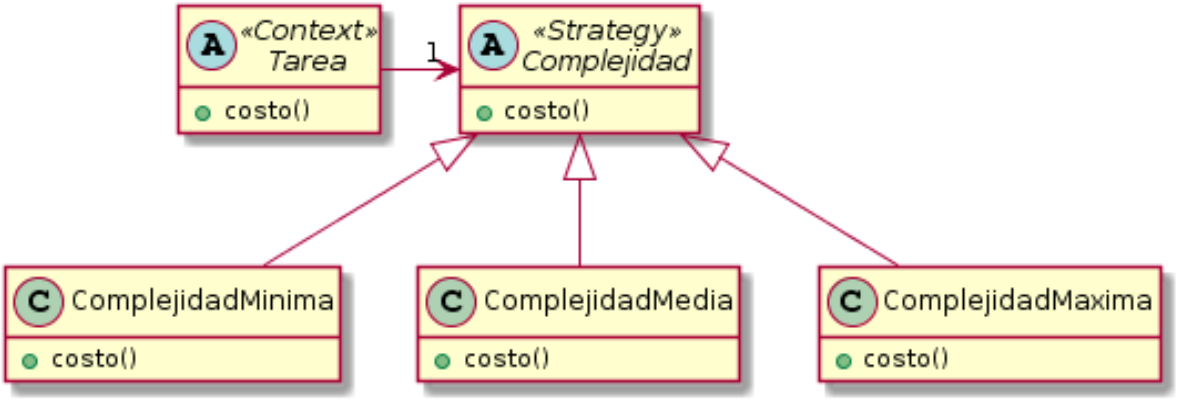
Además del patrón Template Method, explicamos otras ideas de diseño que surgieron en el ejercicio:

6.1 Strategy

Para separar la complejidad de las tareas: cada algoritmo que calcula la complejidad se encapsula en un objeto.



(patrón Strategy del libro *Design Patterns*, de Erich Gamma et al.)



Ventajas

- *Respecto a tener el comportamiento en Tarea y distinguir con un if la complejidad*: encapsulo el algoritmo que calcula el costo y los días máximos de atraso en varios objetos polimórficos. Así hay mayor división de responsabilidades entre la tarea y la complejidad.
- *Respecto a subclasificar la complejidad de la Tarea:*
    - El Strategy permite encontrar abstracciones nuevas (la complejidad) que antes estaban incorporadas dentro de una abstracción mayor (la tarea).
    - mientras que la herencia es estática (me obliga a cambiar de objeto), el cambio de comportamiento que ofrece el Strategy es dinámico (la tarea sigue siendo la misma, conserva la identidad).

Consecuencias

- Hay más objetos (antes había una tarea, ahora hay tarea + complejidad).
- La tarea debe conocer a la complejidad, la complejidad puede o no conocer a la tarea (el que implementa el Strategy debe definir si la complejidad tendrá una variable tarea, si la recibirá como parámetro cuando lo necesite o si sólo recibirá el tiempo).

Desventajas con respecto a la subclasificación/manejo con ifs de la complejidad

- La construcción de una tarea es más compleja
- Se ve distribuido el comportamiento del objeto en otro objeto y no dentro de él mismo, esto parece a simple vista algo no natural.

Ejemplo asociado: una colección que

- almacena los elementos de una determinada manera (Set, List, Map)
- y puede ordenarse con diferentes algoritmos de ordenamiento: Burbujeo,

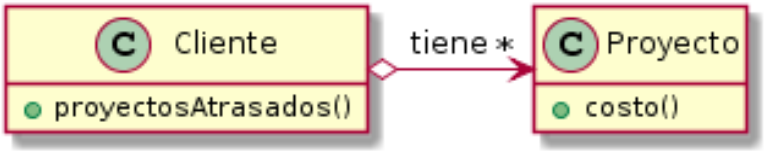
QuickSort, MergeSort, etc. Cada algoritmo representa un *strategy*.

6.2 Composite

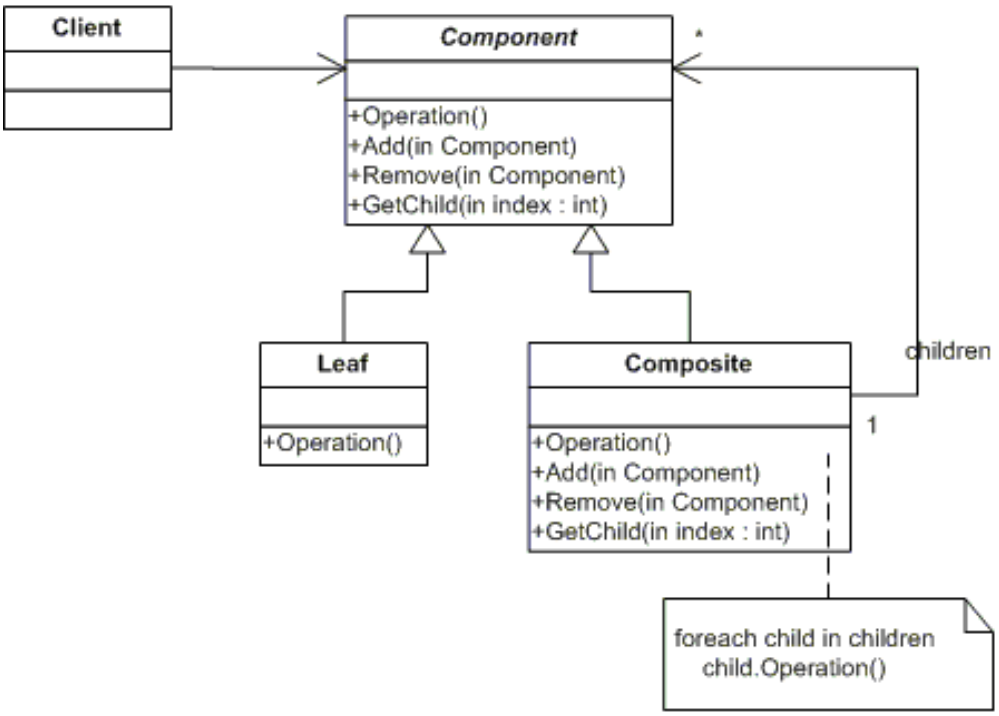
Para trabajar una estructura jerárquica en donde ramas y hojas operan en forma polimórfica. *Ejemplo asociado:* los directorios y los archivos en el Explorador del Sistema operativo:

- yo los quiero trabajar en forma polimórfica (con ambos puedo hacer botón derecho y abrir, copiar o eliminar)
- un archivo no se transforma en directorio y un directorio no se transforma en archivo

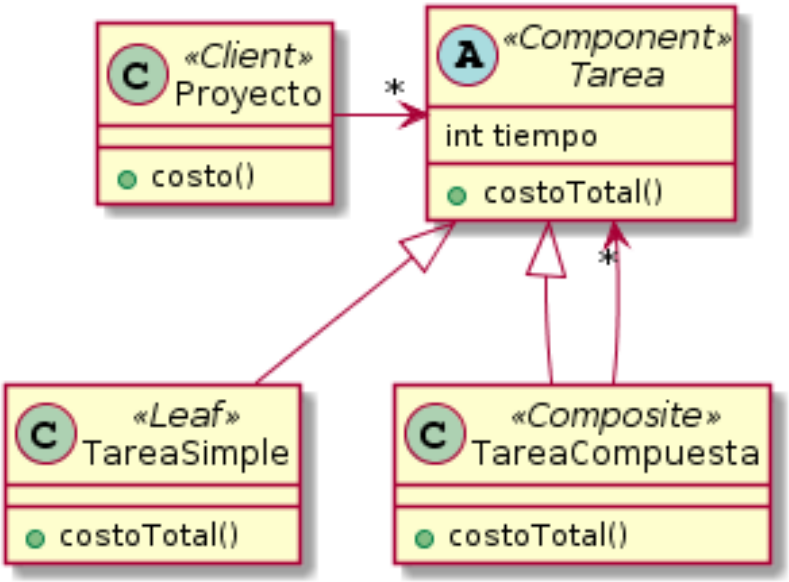
Ojo, composite pattern y composición son dos términos que describen situaciones diferentes. Un cliente tiene proyectos, una tarea tiene recursos, etc. En ese caso estamos hablando de composición de objetos (cuando un objeto tiene una relación de asociación con otro/s):



En el Composite pattern lo que nos interesa remarcar es que **hay un cliente opera con componentes polimórficos de una jerarquía** (al cliente no le interesa distinguir si son simples o compuestos).



Para ser polimórficos, los componentes simples y compuestos comparten una interfaz común. Aplicado al ejercicio:



Cuando la TareaCompuesta tiene una referencia a una colección de tareas de tipo Tarea, **eso permite anidar n niveles de tareas simples y compuestas.**

7 La solución

Se encuentra en el repositorio https://github.com/wollok/manejoProyectos

8 Resumen

A lo largo del apunte hemos estudiado a partir de un ejemplo decisiones no triviales de diseño, en la que la composición surge como una forma de generar jerarquías de objetos polimórficos para implementar comportamientos específicos. La búsqueda de alternativas requiere un análisis objetivo de las ventajas y debilidades de cada solución; también nos sirve para reforzar y retroalimentar el análisis disparando más preguntas al usuario. Por último, este proceso de iteraciones de pensar y bajar a tierra nuestro modelo permite que nuestras decisiones de diseño puedan cambiar y escalar.

Diagrama estático

Manual de uso

por Fernando Dodino

Versión 1.0

Septiembre 2017

Distribuido bajo licencia Creative Commons Share-a-like

Índice

1 Introducción

2 Activación

3 Generando nuestro primer diagrama
- 3.1 Integración con el editor
- 3.2 El diagrama se actualiza automáticamente
- 3.3 Por qué es un diagrama estático y no “de clases”
- 3.4 Exportar el diagrama
- 3.5 Asociación entre wlk y diagrama

4 Configuraciones básicas
- 4.1 Mostrar variables
- 4.2 Recordar posiciones y tamaños de los elementos
- 4.3 Eliminar configuración de tamaños y posiciones de elementos

5 Nuestro valor agregado
- 5.1 Ocultando métodos
- 5.2 Ocultar una clase
- 5.3 Ocultar una variable
- 5.4 Volver a mostrar un elemento
- 5.5 Volver a mostrar atributos y métodos ocultos
- 5.6 Volver a mostrar atributos y métodos de un elemento particular
- 5.7 Panel de Zoom

6 Relaciones
- 6.1 Relaciones de asociación
- 6.2 Relaciones de asociación autorreferenciales
- 6.3 La relación de asociación queda activada
- 6.4 Eliminar una relación de asociación
- 6.5 Eliminar todas las relaciones de asociación y dependencia
- 6.6 Relaciones de dependencia

7 Componentes externos
- 7.1 Agregar componentes externos
- 7.2 Eliminar elementos externos

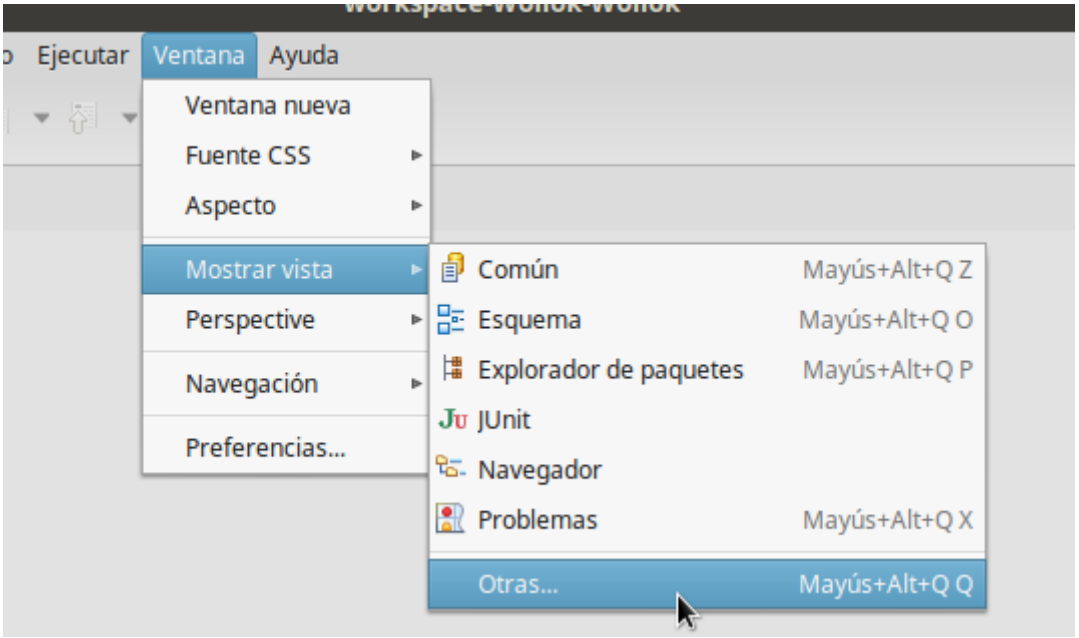
8 Resumen

1 Introducción

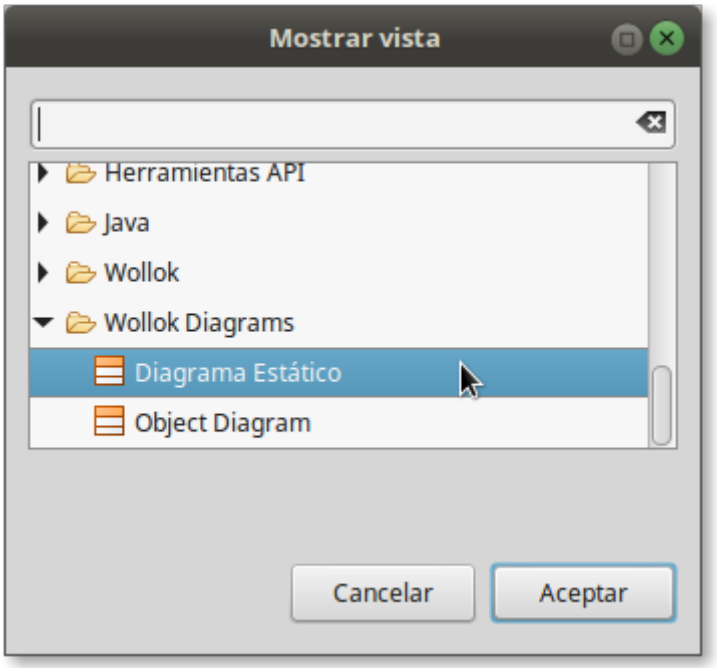
En el presente capítulo estudiaremos el diagrama estático que trae la herramienta Wollok para documentar el diseño de un trabajo general. Asumiremos que el idioma del entorno es español, por lo que algunos nombres y mensajes pueden diferir basado en el idioma que tenga instalado.

2 Activación

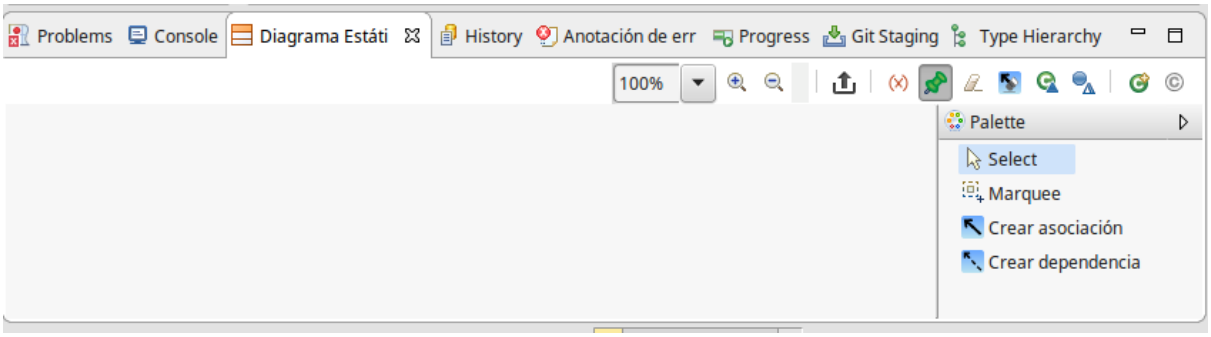
En el caso que el diagrama se encuentre oculto, se puede activar desde el menú Ventana: Ventana | Mostrar vista | Otras...



Y luego dentro del conjunto “Wollok diagrams” está el Diagrama Estático



Al dar Aceptar se visualizará el Diagrama dentro del entorno de trabajo:



Inicialmente no muestra nada, ya que trabaja siempre asociado a un archivo.

3 Generando nuestro primer diagrama

El diagrama estático

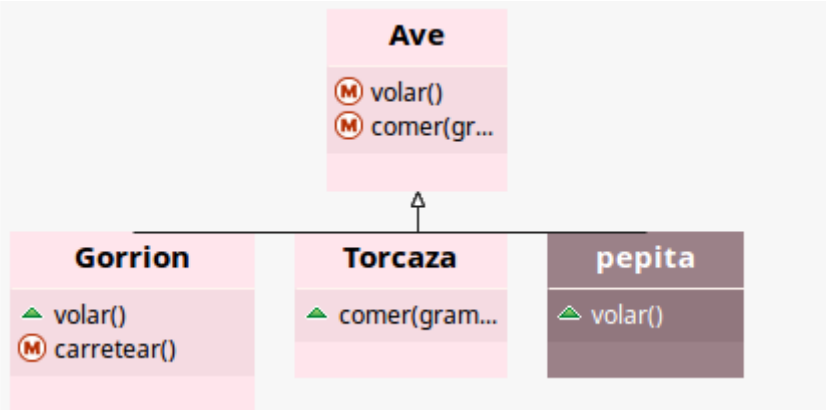
- está pensado para mostrar los elementos estáticos de un archivo wlk
- no está pensado para tests o programas

Podemos abrir un archivo wlk, y esto muestra todas las definiciones de ese archivo (objetos, clases y mixins). Supongamos que tenemos este archivo:

```
class Ave {
	var energia = 0
	method volar() { energia -= 15 }
	method comer(gramos) { energia += gramos }
	method energia() = energia
	method energia(_energia) { energia = _energia }
}
```

```
class Gorrion inherits Ave {
      var color
      var carreteos = 0
      method color() = color
      method color(_color) { color = _color }
      override method volar() {
            super()
            color = "gris"
      }
      method carretear() { carreteos = carreteos + 1 }
}

class Torcaza inherits Ave {
      override method comer(gramos) {  energia += 50 }
}

object pepita inherits Ave {
      override method volar() { energia -= 10     }
}
```

Esto produce por defecto el siguiente diagrama estático:

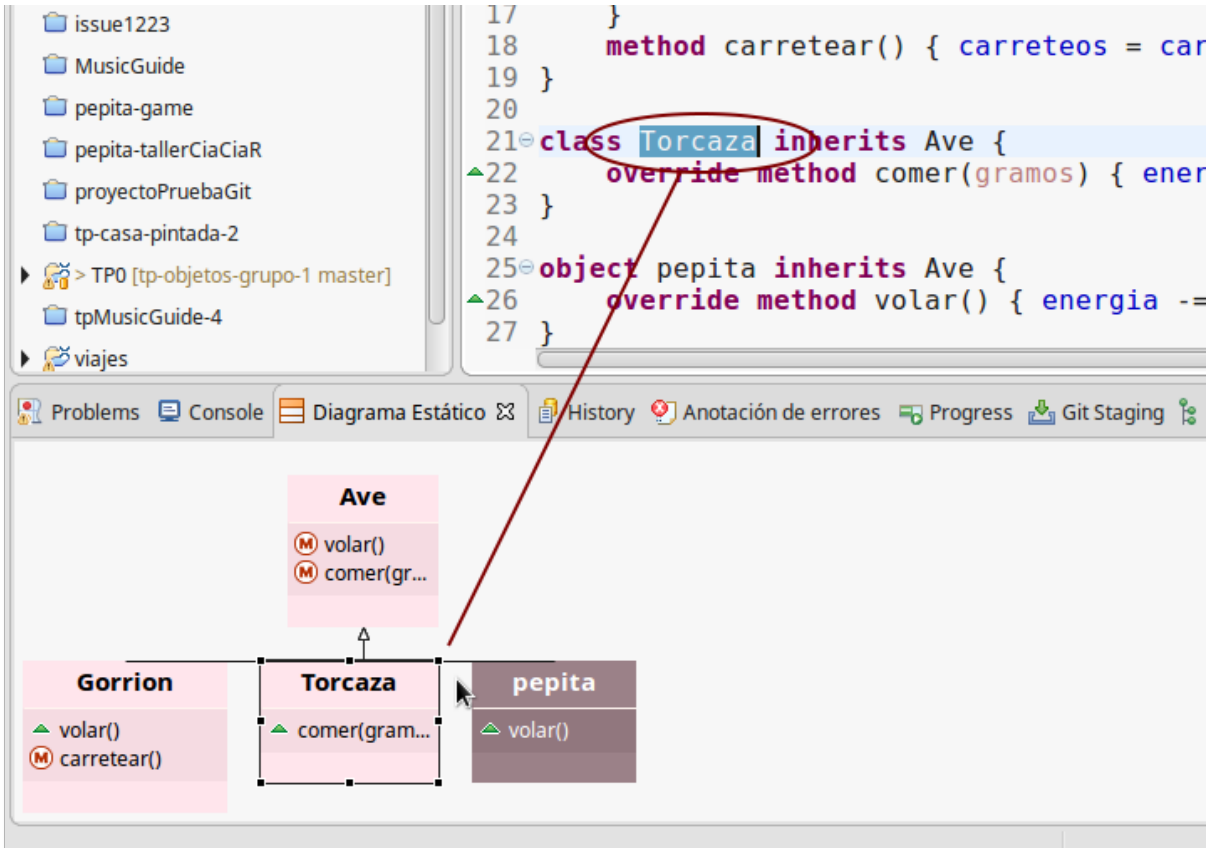


Algunas observaciones:

- las subclases se ordenan alfabéticamente
- no aparecen las variables
- los accessors (getters y setters) se ocultan, siguiendo la convención de que deben tener el mismo nombre que la referencia
- los métodos redefinidos aparecen con una flecha verde
- el diagrama se ordena de arriba hacia abajo y de izquierda a derecha

3.1 Integración con el editor

El diagrama se integra con el editor, de manera que al hacer click sobre un método, va directamente a la definición de dicho método (lo mismo con la clase, o las variables cuando necesitemos verlas).

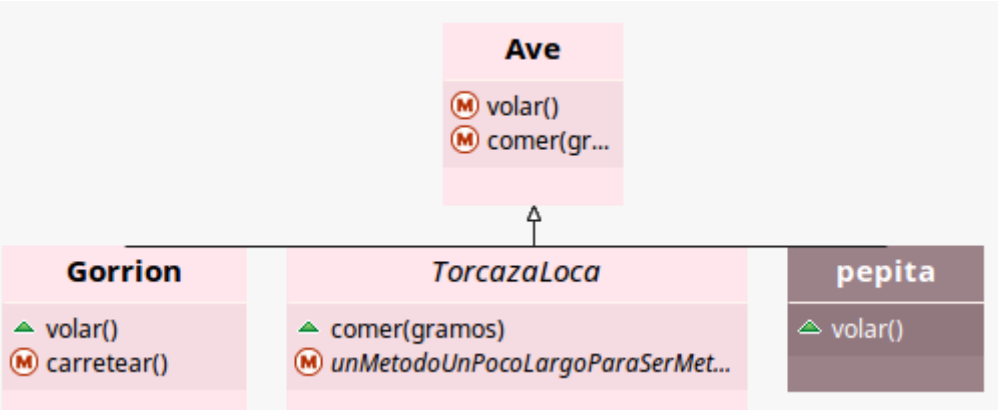


3.2 El diagrama se actualiza automáticamente

Una vez asociado a un archivo, el diagrama se actualiza con cualquier modificación de código: si cambiamos el nombre de la clase, de un método o una referencia, el diagrama de clases se acomoda para que se muestren los elementos correctamente: si escribimos

```
class TorcazaLoca inherits Ave {
      override method comer(gramos) {  energia += 50 }
      method unMetodoUnPocoLargoParaSerMetodo
}
```

Eso se ve reflejado de la siguiente manera en el diagrama:



Como vemos

- TorcazaLoca no define aún “unMetodoUnPocoLargoParaSerMetodo”, por lo tanto se considera clase abstracta y se marca en cursiva.
- También marca “unMetodoUnPocoLargoParaSerMetodo” como método abstracto, ya que no tiene implementación

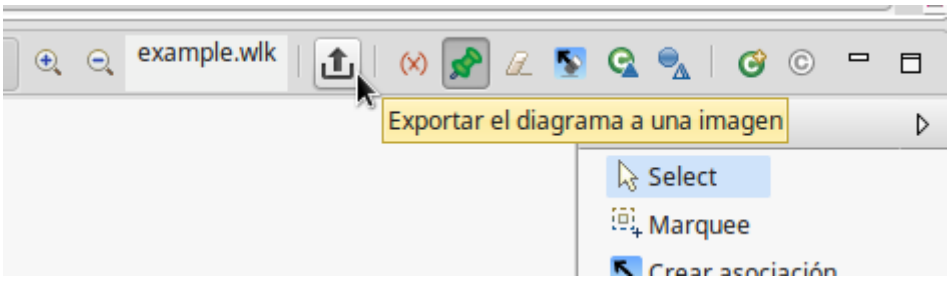
3.3 Por qué es un diagrama estático y no “de clases”

Como vimos anteriormente, pepita es un wko y aparece en el diagrama estático, que ya no es exclusivamente de clases. También los mixins[1] escapan de la definición inicial de clase.

Sí es estático ya que su intención es mostrar la estructura general de la solución, sin detenerse a pensar en casos concretos para todos los elementos que participan del diagrama.

3.4 Exportar el diagrama

En cualquier momento que necesites podés exportar el diagrama en formato png, mediante el botón “Exportar el diagrama a una imagen” que está en la barra de herramientas del diagrama:



[1] no te preocupes si no viste este concepto, es una idea avanzada para reutilizar comportamiento

Esto permite guardar el png en un directorio.

3.5 Asociación entre wlk y diagrama

Una de las preguntas iniciales que uno se hace es: ¿puedo pasar el diagrama a un compañero? Sí, podés, ya que hay un paralelismo entre el archivo wlk y su diagrama, y se almacena dentro del proyecto.

```
+ proyectoTP
      + src
            - example.wlk
      + .diagrams
            - example.wsdi
```

Cuando subas los cambios, no ignores la carpeta .diagrams para que tus compañeros al bajar los cambios del repositorio puedan visualizar el diagrama exactamente igual que vos.

Para esto ambos deben usar exactamente la misma versión de Wollok. O de lo contrario será necesario que cada uno mantenga su propio diagrama

- ignorando el directorio .diagrams en el control del código que hace el repositorio
- y borrando los archivos .wsdi que estén en dicha carpeta

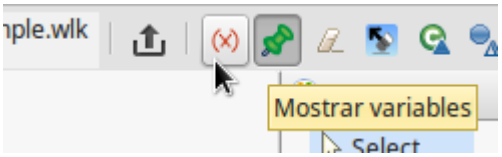
ya que de lo contrario te va a aparecer un mensaje de error.

4 Configuraciones básicas

La barra de herramientas del diagrama estático tiene muchas variantes, vamos a explicar a continuación cada una de ellas.

4.1 Mostrar variables

Por defecto, las variables no se muestran en el diagrama. Para activar la visualización de dichas referencias, se debe dar un click en el botón “Mostrar variables” que tiene un ícono rojo con una equis entre paréntesis:

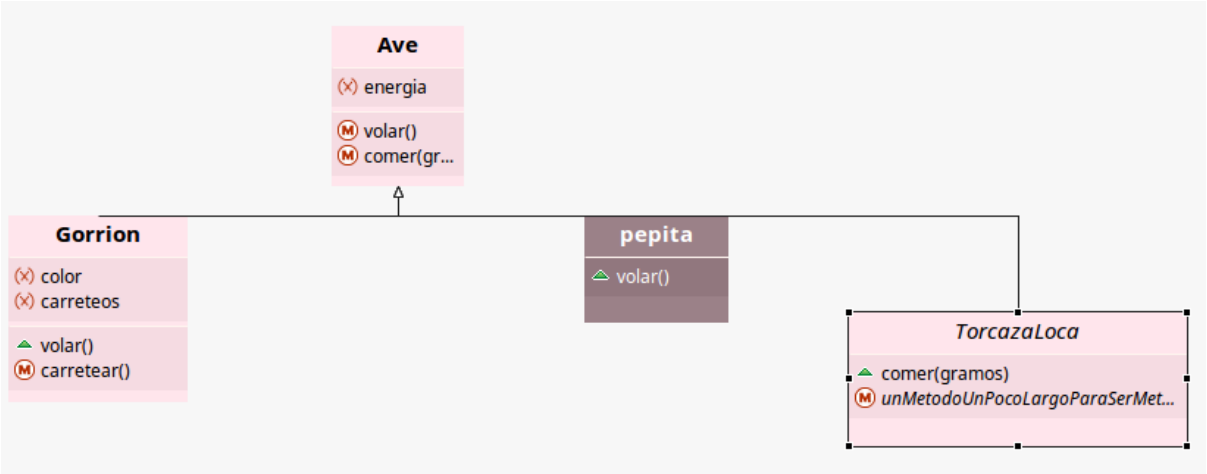


El botón es *toggle*, esto significa que deja activa la visualización de variables. El diagrama ahora muestra:

Y como vemos el botón “Mostrar variables” queda presionado. Para desactivar esta opción volvemos a presionar el mismo botón.

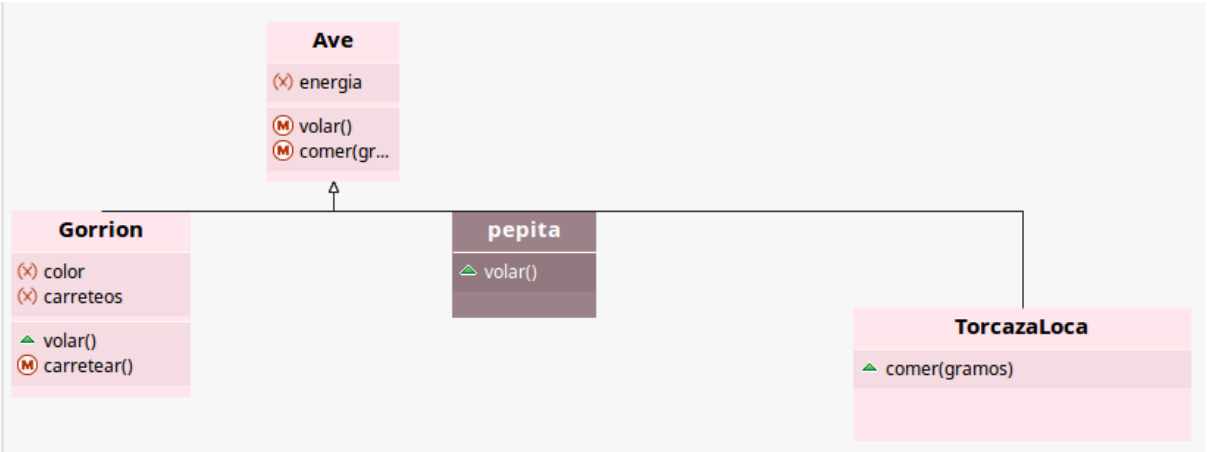
4.2 Recordar posiciones y tamaños de los elementos

Por defecto está activada esta opción. Esto implica que al mover un elemento de lugar, o agrandarlo/achicarlo, tanto la posición como el tamaño se almacenan dentro de la configuración. Entonces cuando el código cambia, todos los elementos se redibujan a excepción del componente que previamente movimos. Por ejemplo, llevemos *TorcazaLoca* a la derecha de *pepita*:

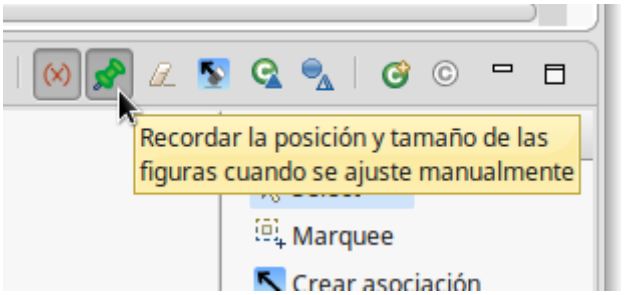


Esto se hace al seleccionar TorcazaLoca mediante un click izquierdo del mouse, luego mantenemos apretado y arrastramos hacia la derecha el mouse, soltando en el lugar donde nosotros queremos ubicar a dicha clase.

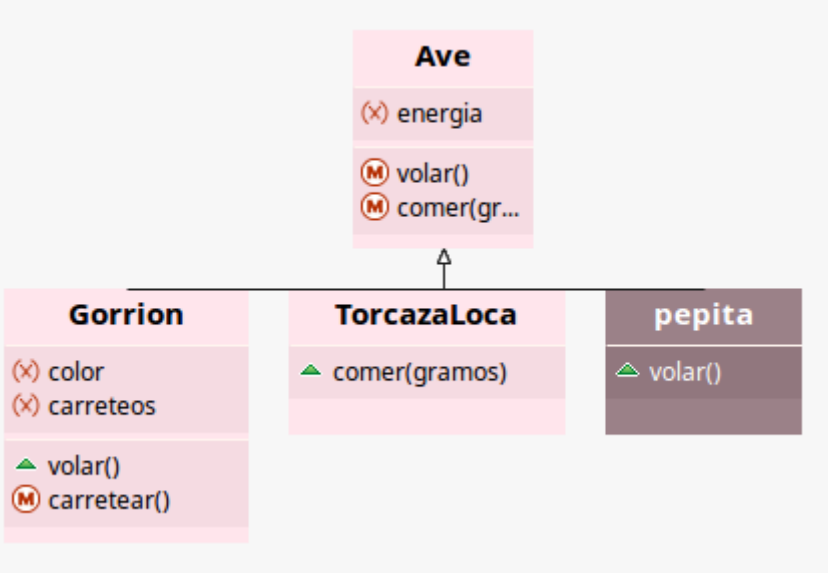
Si eliminamos el método con extensión larga, vemos que TorcazaLoca queda con un ancho grande:



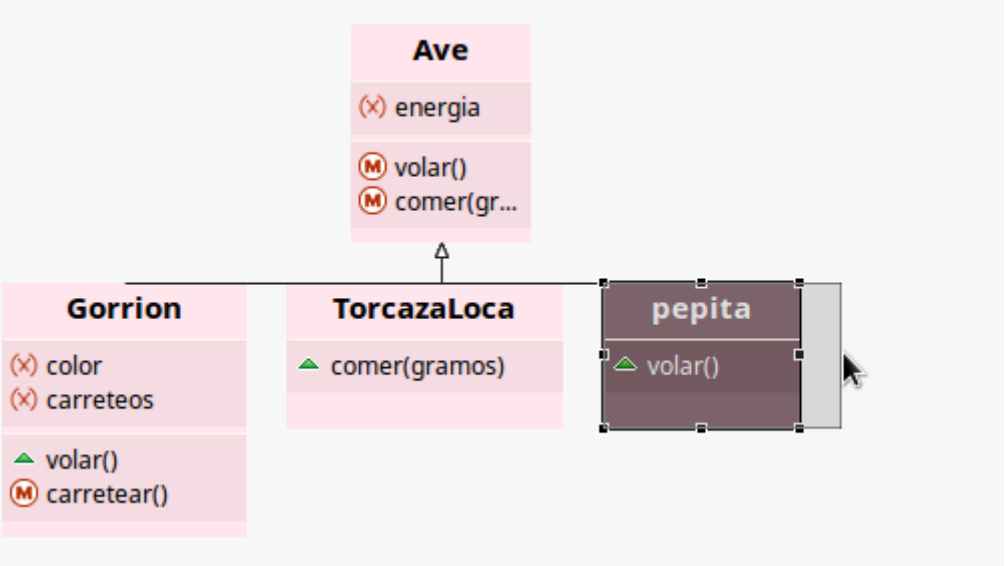
Esto es debido a que el toggle button que recuerda tamaños y posiciones de los elementos está activo:



Si desactivamos ese botón haciendo click, el diagrama se vuelve a dibujar desde cero cada vez:



Entonces no importa si movemos un elemento hacia la izquierda, o derecha, o lo agrandamos (posicionándonos sobre uno de los extremos, haciendo click y arrastrando hasta el tamaño deseado):

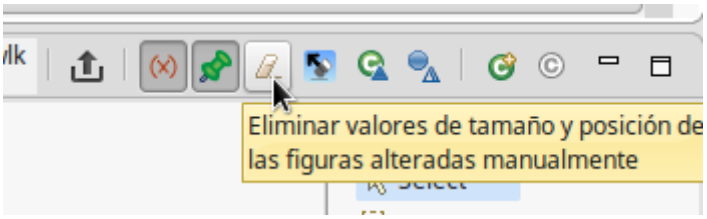


Al escribir nuevamente un espacio en el editor, el diagrama volverá a dibujarse desde cero, perdiendo la configuración. Por este motivo es importante dejar activo el botón que recuerda tamaños y posiciones, sabiendo que

- **si se agregan nuevos métodos, o los métodos son más largos, debemos manualmente ajustar el diagrama**. Por este motivo es conveniente hacer ajustes previo a una entrega de trabajo práctico, o cuando sabemos que las definiciones de dicho archivo son estables.
- **si modificamos el nombre de la clase, la configuración que tenía se pierde**, ya que la referencia es por el nombre completo de la clase: paquete + clase, lo que se conoce como FQN o *full qualified name*. Por ejemplo si tenemos un paquete model y el nombre de la clase es Ave, el FQN es model.Ave

4.3 Eliminar configuración de tamaños y posiciones de elementos

Supongamos que modificamos muchas posiciones o tamaños de los elementos, ¿cómo podemos eliminar esa configuración sin tener que presionar dos veces el botón “Recordar posición y tamaño de los elementos”? Mediante el botón que está al lado, que es una goma de borrar:



De esa manera el diagrama vuelve a redibujarse pero seguimos teniendo activada la opción de recordar los cambios manuales en las posiciones o tamaños de los componentes.

5 Nuestro valor agregado

Sabemos que las herramientas “automágicas” necesitan cierta inteligencia adicional, para que el diagrama estático muestre información importante y no sea un simple vuelco de la misma información que está en el código.

Entonces el valor que le damos nosotros al diagrama es que podemos decirle

- qué elementos son importantes comunicar
- cuáles son las responsabilidades más importantes que cumplen los componentes (los métodos que queremos comunicar)
- y eventualmente si queremos mostrar las referencias, cuáles son las variables que resaltan en nuestra solución

Esto lo podemos hacer ocultando

- clases, objetos o mixins
- referencias
- métodos

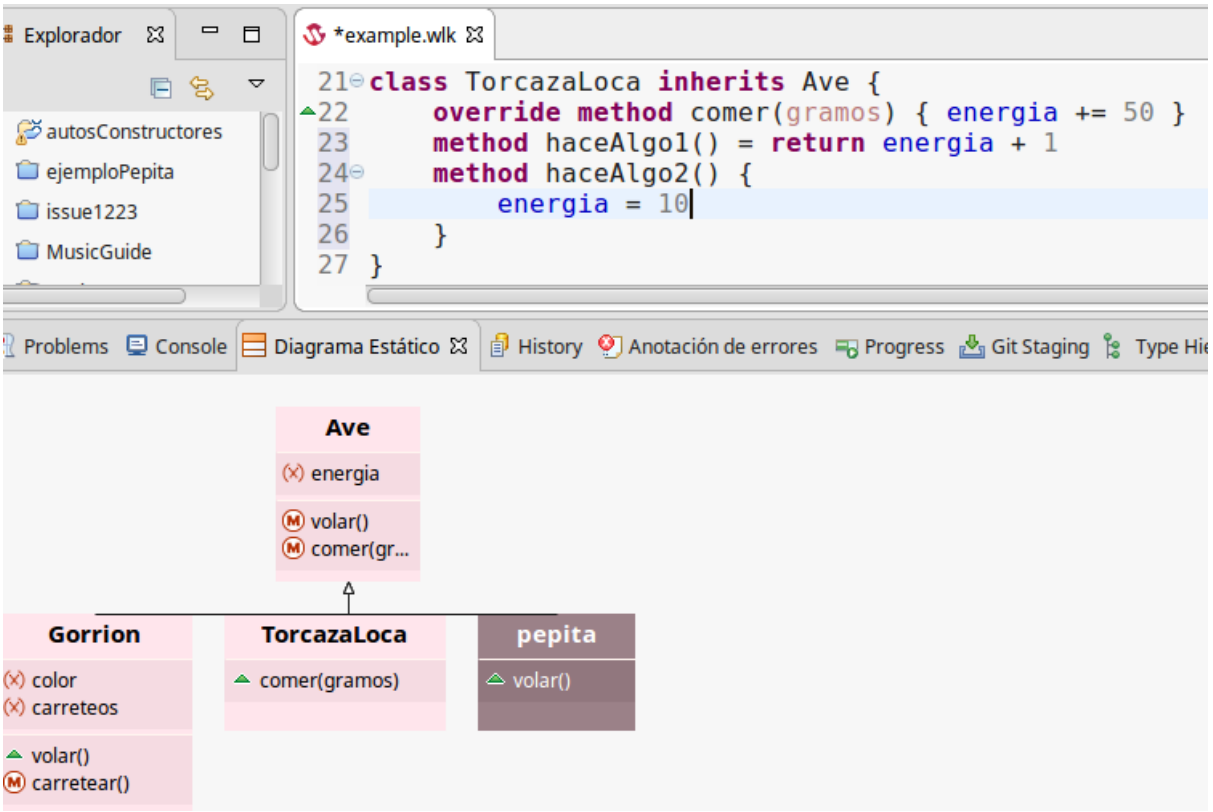
5.1 Ocultando métodos

Por ejemplo, definimos dos métodos adicionales en TorcazaLoca:

```
class TorcazaLoca inherits Ave {
      override method comer(gramos) {  energia += 50 }
      method haceAlgo1() = return energia + 1
      method haceAlgo2() = return energia + 2
}
```

Pero son métodos internos, o bien que podrían formar parte de la implementación de otro requerimiento más general. Entonces vamos a ocultar ambos métodos. Seleccionamos el método haceAlgo1(), luego botón derecho y Eliminar del diagrama:

Hacemos lo mismo con el método haceAlgo2() y vemos que a partir de aquí no se visualiza más esos métodos:

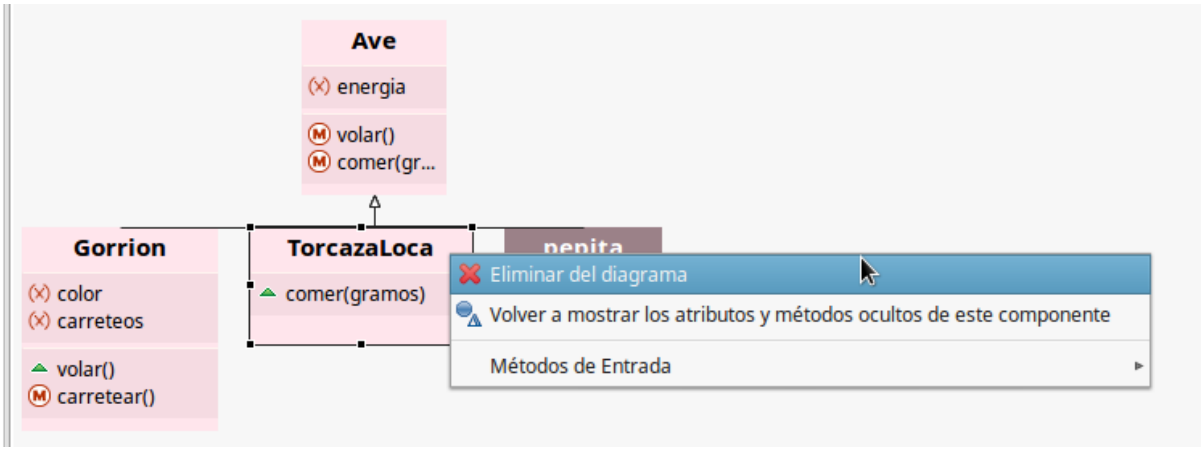


Importante: dado que la configuración almacena los nombres de los métodos a ocultar

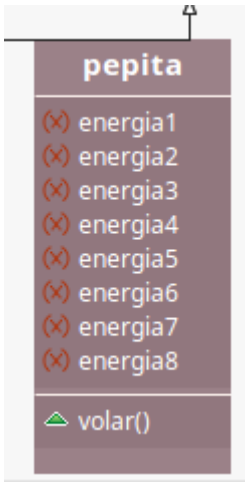
- si renombran algún método oculto, éste volverá a verse en el diagrama
- si además algún método nuevo tiene el nombre de algún método oculto que ha sido renombrado, éste no se verá en el diagrama. Por ejemplo: si ocultamos hacerAlgo1(), lo renombramos a hacer() y luego escribimos un método hacerAlgo1(), en el diagrama se visualizará hacer() y no hacerAlgo1()

5.2 Ocultar una clase

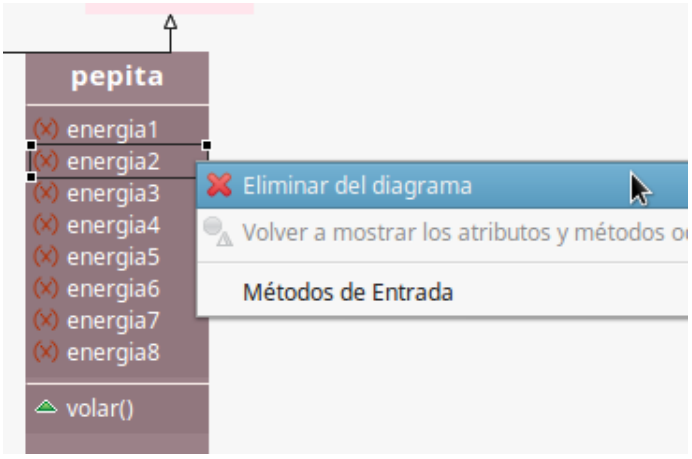
Con un botón derecho sobre la clase, se la puede eliminar del diagrama



5.3 Ocultar una variable



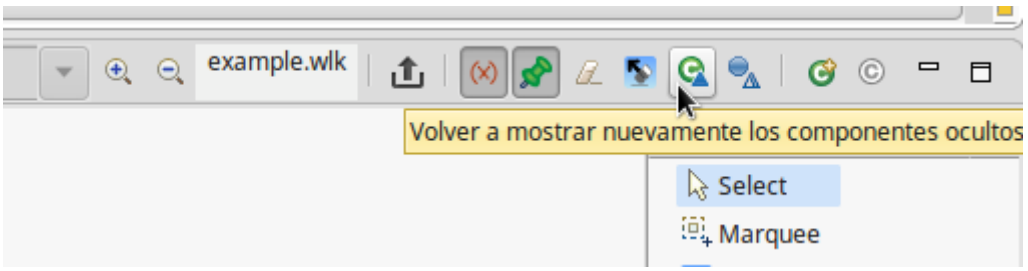
Si estamos mostrando variables de un elemento, es un problema cuando un componente tiene muchas referencias.
En ese caso podemos hacer click sobre las variables que no queremos mostrar y eliminarlas del diagrama.



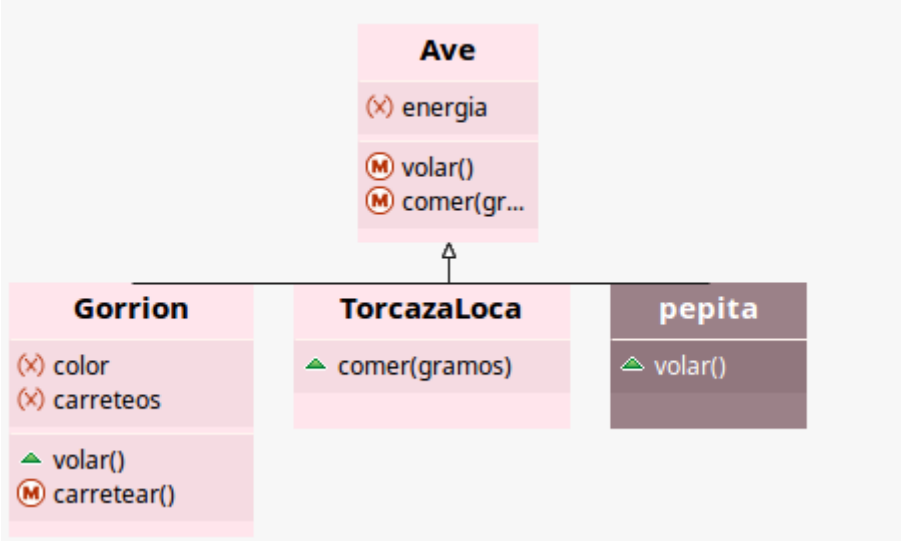
Las variables a ocultar se guardan en la configuración en base al nombre, entonces hay que tener en cuenta en caso de renombrar que volverán a aparecer y desaparecerá cualquier variable nueva que tenga el nombre de alguna variable oculta.

5.4 Volver a mostrar un elemento

Para visualizar nuevamente un elemento oculto, tenemos una opción en la barra de herramientas, representado por el ícono de clase (una C enmarcada en un círculo verde) con un triángulo azul en la diagonal derecha de abajo:

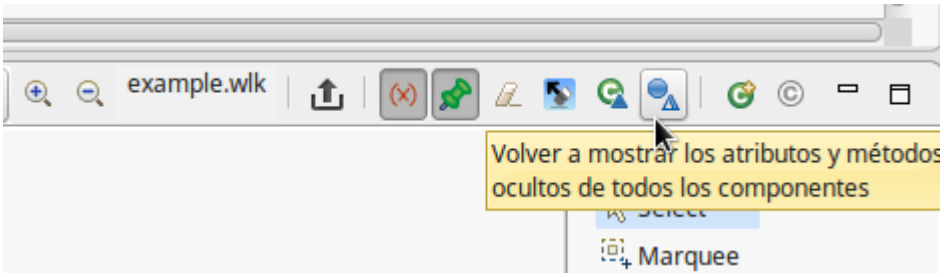


Esto vuelve a mostrar TorcazaLoca en el diagrama:

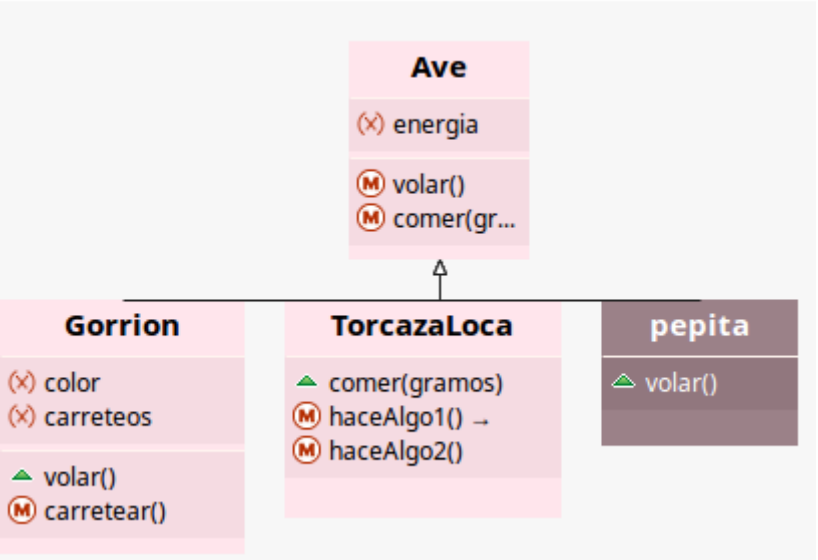


5.5 Volver a mostrar atributos y métodos ocultos

En TorcazaLoca todavía no aparecen dos métodos ocultos hacerAlgo1() y hacerAlgo2(). Para volverlos a visualizar en el diagrama, tenemos la opción en la barra de herramientas, representado por un círculo azul con un triángulo azul en la diagonal derecha de abajo:

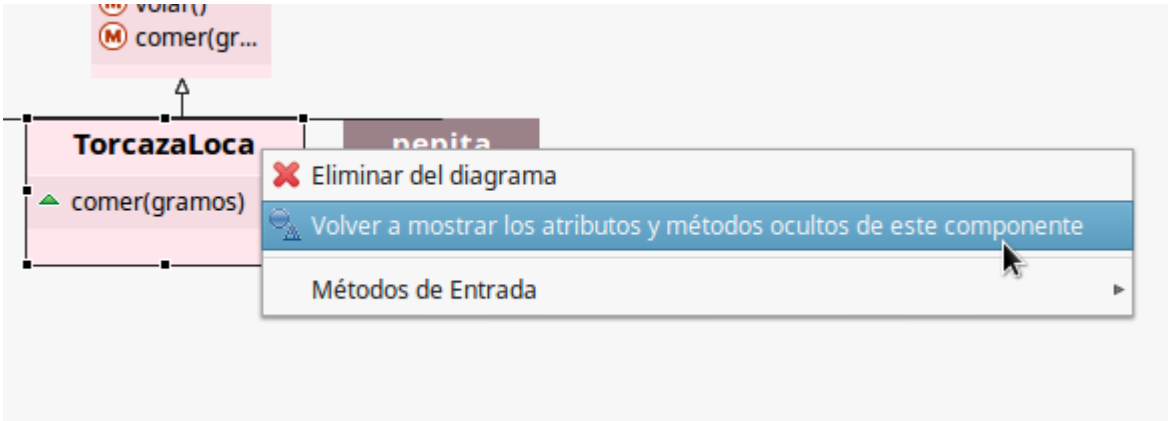


Ahora sí, los dos métodos de TorcazaLoca que estaban ocultos se visualizan nuevamente:

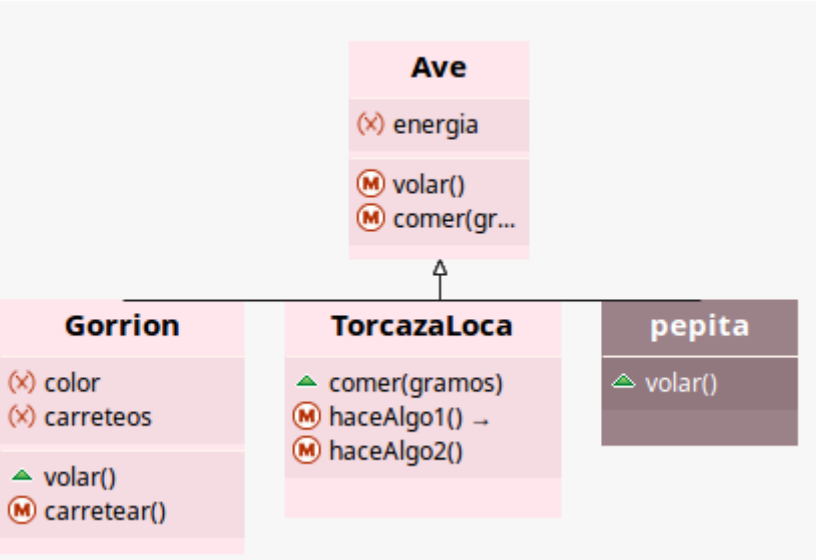


5.6 Volver a mostrar atributos y métodos de un elemento particular

La barra de herramientas hace aparecer todos los atributos y métodos ocultos de todos los componentes. Esto puede ser demasiado abarcativo cuando nos equivocamos y ocultamos involuntariamente un atributo o método de un componente. Entonces podemos seleccionar la clase, dar un click derecho y seleccionar la opción “Volver a mostrar los atributos y métodos ocultos de este componente”:

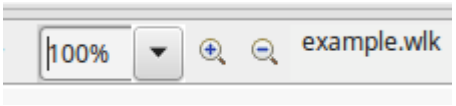


De esa manera, solo los atributos y métodos ocultos de TorcazaLoca se volverán a visualizar, dejando los atributos y métodos de los otros elementos ocultos:

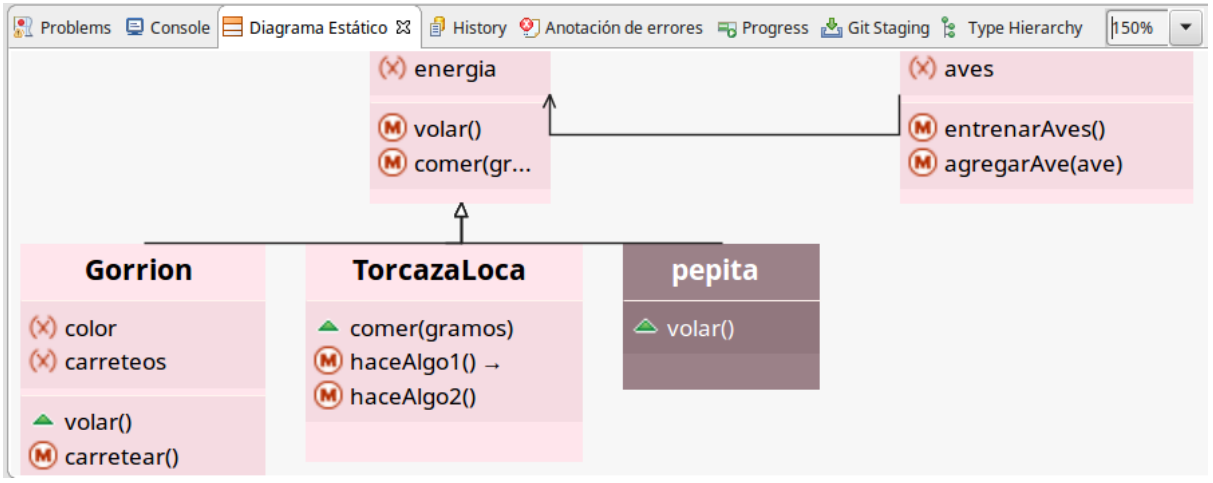


5.7 Panel de Zoom

El diagrama estático cuenta con un panel de zoom, para aumentar o disminuir el tamaño de las clases visualizadas. Esto se puede lograr mediante el cuadro de selección (que permite ingresar un número) o bien con los botones + / - de la barra de herramientas:



Al presionar el botón + podremos ver el efecto que tiene:



6 Relaciones

Las relaciones de herencia son inmediatamente detectadas por el diagrama estático. Por el momento hay otras dos relaciones que debemos incorporar manualmente:

- relaciones de asociación
- relaciones de dependencia

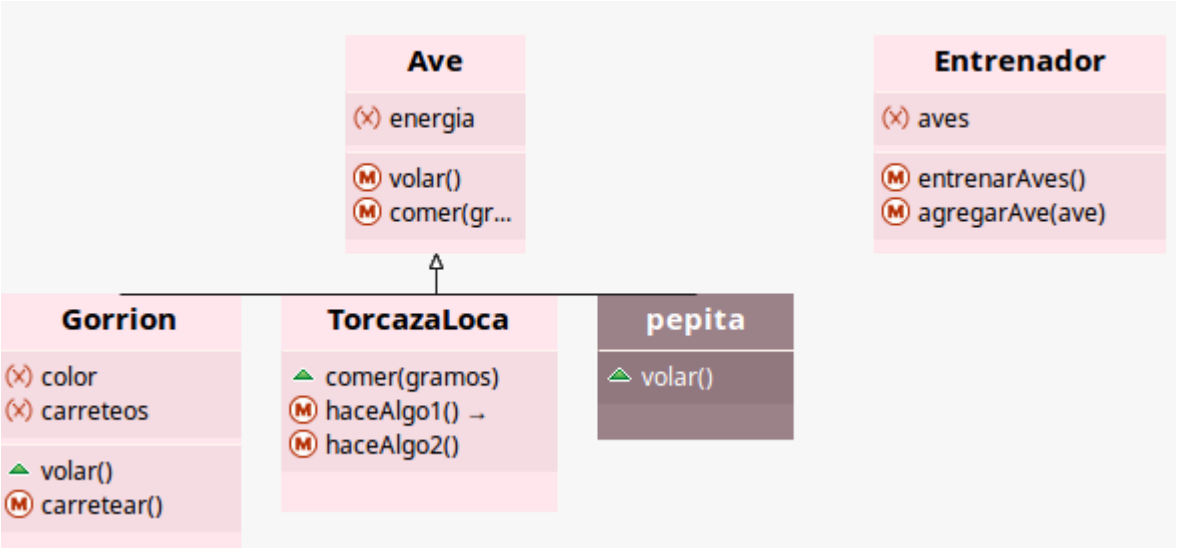
En caso de dudas puede verse el apunte que explica cómo comunicar las decisiones de diseño.

6.1 Relaciones de asociación

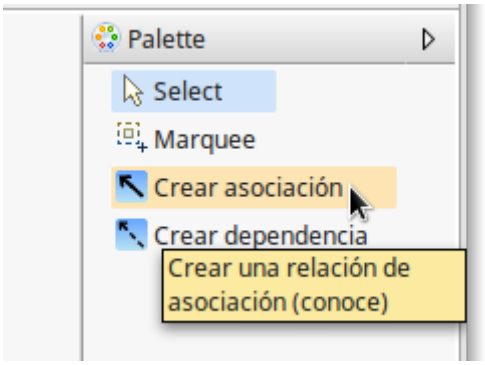
Una relación de asociación implica que un objeto A tiene una referencia a un objeto B. En nuestro ejemplo podemos incorporar un entrenador, que conoce a una lista de aves:

```
class Entrenador {
      const aves = []
      method entrenarAves() {
            aves.forEach { ave => ave.volar() }

      }
      method agregarAve(ave) { aves.add(ave) }
}
```

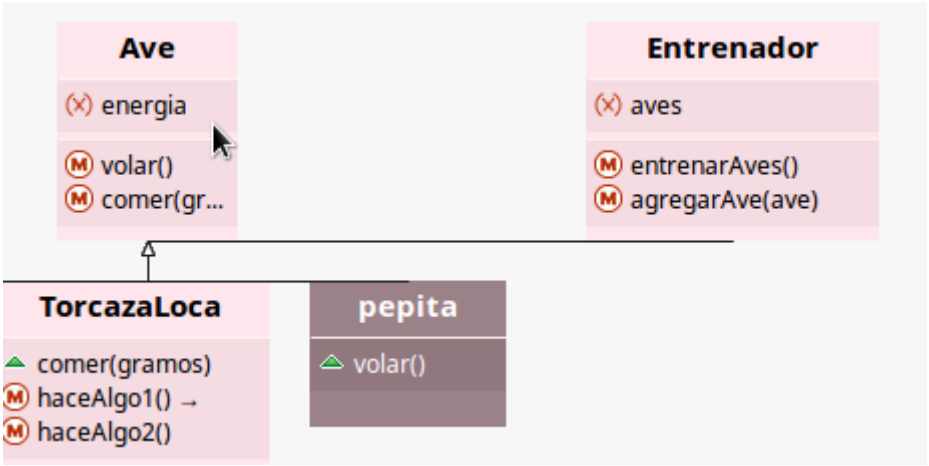
Al agregar esta clase, el diagrama se ve de la siguiente manera:



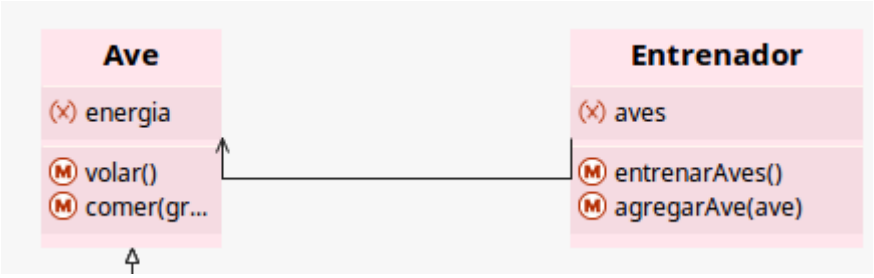
Aquí nos está faltando marcar la relación que tiene Entrenador con Ave, asumiendo que aves será una lista de especializaciones de Ave. Para ello, vamos a la paleta (*Palette*) que está ubicada a la derecha del diagrama y seleccionamos la herramienta "Crear asociación"



A continuación debemos hacer un click en el primer elemento, y otro click en el segundo: esto marcará el origen y destino de la asociación

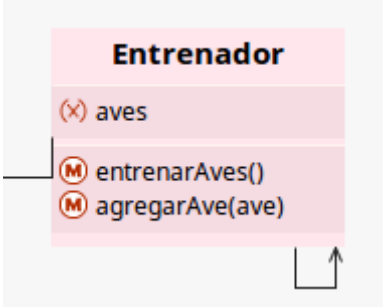


Con el segundo click se genera la relación de asociación en nuestro diagrama:



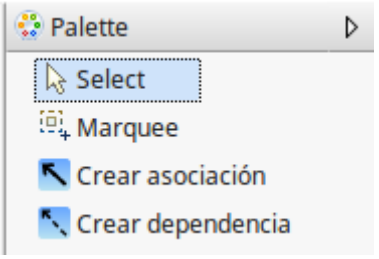
6.2 Relaciones de asociación autorreferenciales

Es posible generar una relación de asociación que vaya de un elemento a sí mismo, basta con seleccionar la herramienta "Crear asociación", elegir un componente y luego hacer dos clicks:



6.3 La relación de asociación queda activada

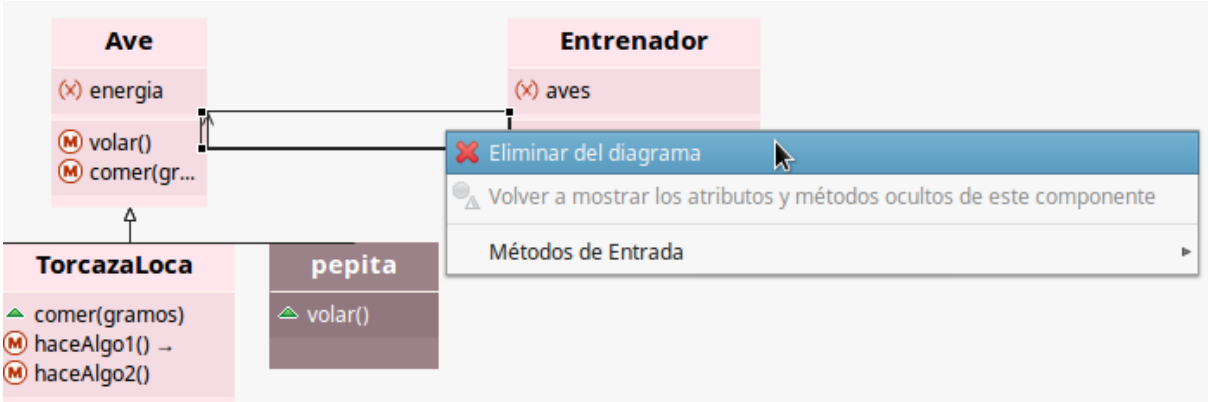
Esto implica que al seleccionar la herramienta “Crear asociación” para volver a mover elementos o seleccionarlos, debemos volver a presionar “Select”:



El sentido que tiene esto es que permite configurar todas las relaciones de asociación juntas.

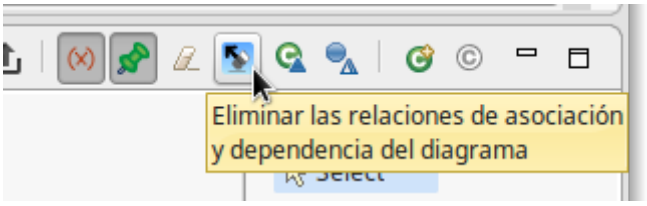
6.4 Eliminar una relación de asociación

Para eliminar una relación de asociación, primero hay que elegir la opción “Select” en la paleta (“*Palette*”) y luego hacer click sobre la relación



6.5 Eliminar todas las relaciones de asociación y dependencia

Para no tener que eliminar las relaciones de asociación y dependencia una por una, puede presionar el botón que está en la barra de herramientas, representado por una goma de borrar sobre una flecha que enmarca una relación:



Esto permite rápidamente recomenzar la tarea de definir las relaciones de asociación en el diagrama.

6.6 Relaciones de dependencia

Aunque menos frecuente, una relación de dependencia marca un conocimiento más débil de un objeto hacia otro, que se da

- porque un objeto A tiene una variable local de B
- o bien porque A instancia un objeto B

sin almacenarlo en una variable de instancia.

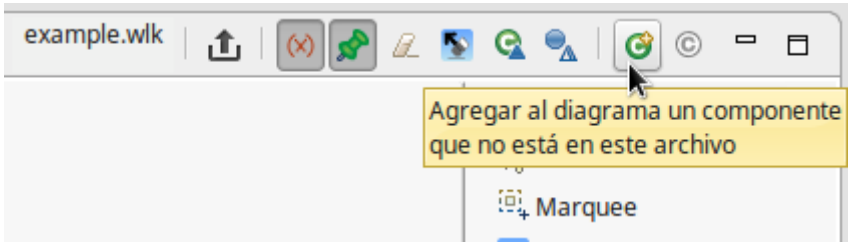
La operatoria es similar a la creación y eliminación de relaciones de asociación, agregando estas restricciones:

- si definieron que Entrenador conoce a Ave, no tiene sentido que Entrenador use Ave, es decir que una relación de asociación hace innecesario que se defina una relación de dependencia adicional. En el diagrama estático eso se valida, por lo que no van a poder hacerlo.
- no es posible marcar una relación de asociación de un objeto hacia sí mismo.

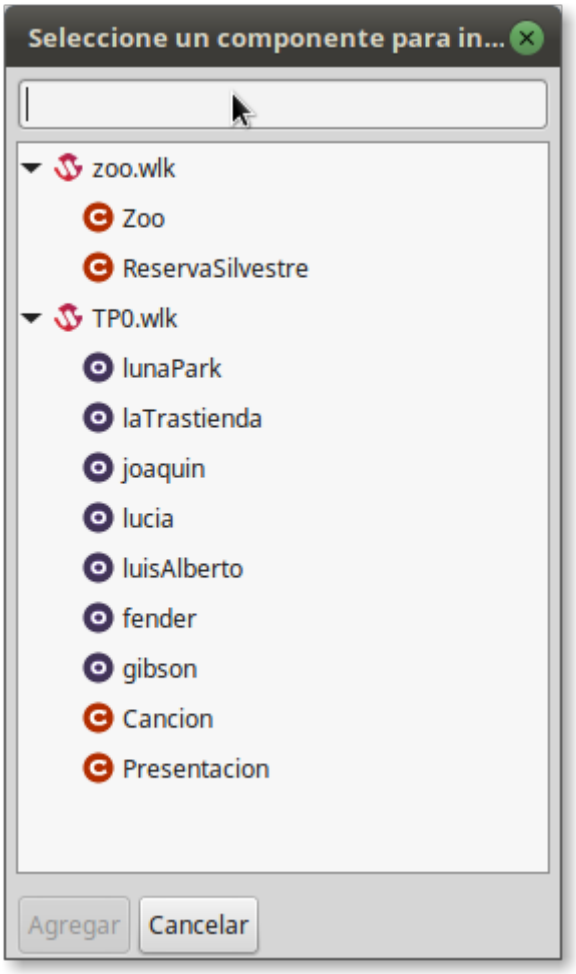
7 Componentes externos

7.1 Agregar componentes externos

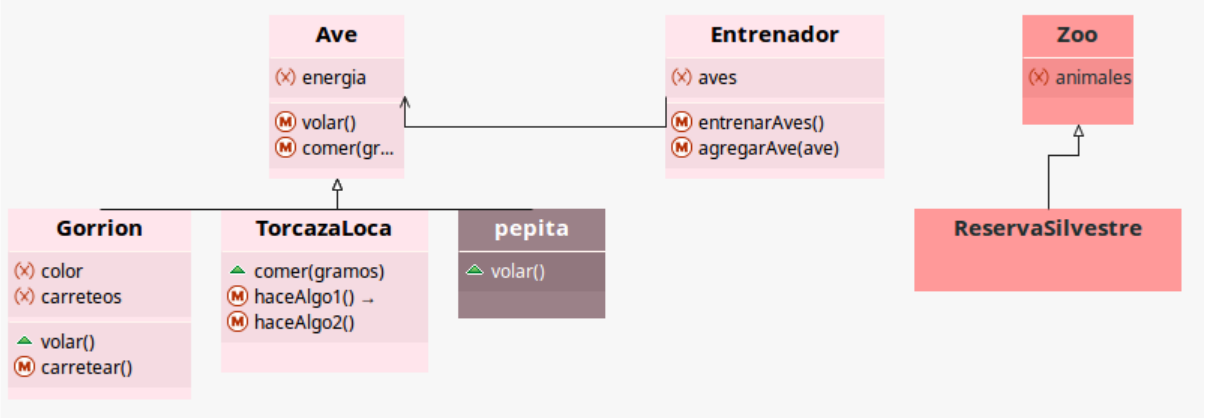
Lo común es que hagamos definiciones en diferentes archivos wlk. Para tener un diagrama en donde podamos dar un vistazo general a nuestra solución, el diagrama estático permite agregar un elemento externo:



Esto habilita un menú contextual, donde podemos elegir uno o varios elementos a incorporar:

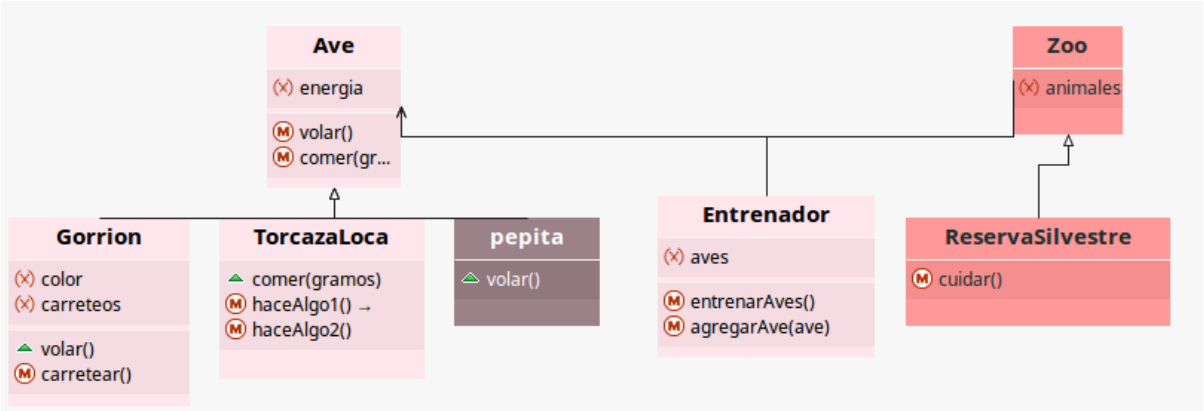


Aquí se muestran los elementos de cada archivo wlk que esté en el proyecto actual. Podemos seleccionar ReservaSilvestre, por ejemplo y el efecto que tiene es traer tanto ReservaSilvestre como Zoo, que es su superclase:



Cada vez que actualicemos el código en el archivo zoo.wlk, y volvamos luego al archivo example.wlk, estos cambios se visualizarán en el diagrama. Podemos

- ocultar atributos, métodos o componentes de archivos externos
- también definir relaciones de asociación o dependencia

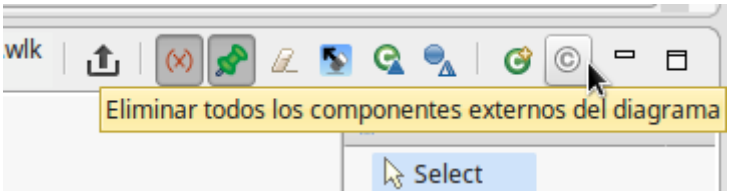


Aquí vemos que tanto Entrenador como Zoo conocen a Ave.

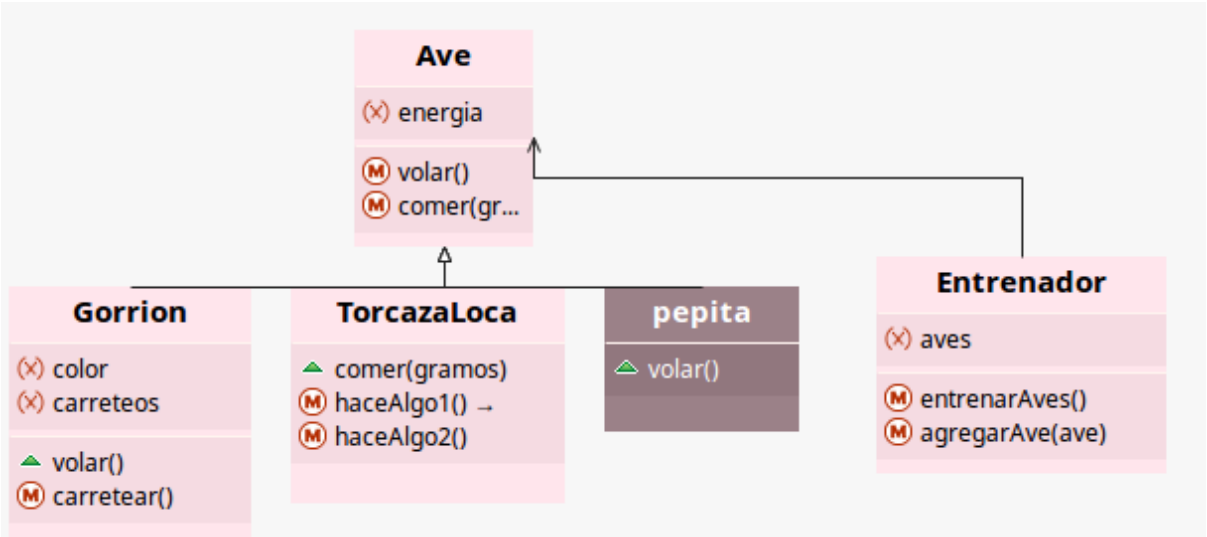
Para diferenciar los elementos que están en el archivo de los que son externos, aparecen en el diagrama estático con otro color.

7.2 Eliminar elementos externos

El diagrama presenta una opción en la barra de herramientas, para eliminar componentes situados en archivos externos:



El diagrama estático vuelve a tener los elementos del archivo original:



8 Resumen

En el presente apunte hemos repasado al diagrama estático como una herramienta que ayuda a la comunicación del diseño, recordando que la parte más importante es el valor agregado irreemplazable que tiene la persona que modela una solución.

Tutorial Git en Wollok
Versión 2.0
Agosto 2017

por Tobías Sandler,
Alfredo Sanzo
Versión 2.0
Agosto 2017

Índice

¿Qué es Git?

Instalar Plugin de Git en Wollok

Clonar un proyecto usando Eclipse
- ¿Dónde está lo que quiero clonar?
- Ahora sí, a clonar

Hacer pull usando Eclipse

Hacer commit y push usando Eclipse
- ¿Cada cuánto es conveniente hacer un commit?
- Subiendo archivos al repositorio remoto

Deshacer mis cambios para hacer pull

Flujo de trabajo simple

Resolución de conflictos usando Eclipse

Flujo de trabajo completo

Etiquetar un commit (tag)

Significado de los símbolos

Extra 1: Terminología completa de Git

Extra 2: Equivalencias con la línea de comandos

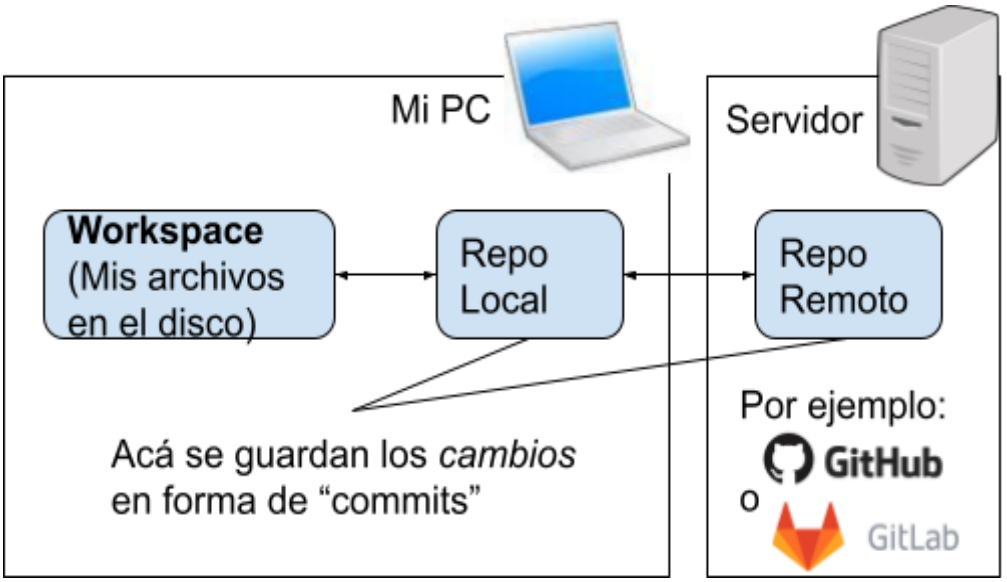
Extra 3: ¿Cómo subo a Github cualquier proyecto mío?

1. ¿Qué es Git?

Git es un *software de control de versiones* que facilita mucho el trabajo de programar en un equipo de personas. En particular resuelve varios problemas: los que tienen que ver con **compartir el código** para que varios desarrollen, y permitir tener **control sobre los cambios**. ¡Es como tener un Ctrl+Z con esteroides!

Entonces lo que hace git es *agregar dos lugares más donde va a estar tu código*, además del típico lugar en disco: esos lugares se llaman **repositorios**. Git básicamente trabaja con dos repositorios:

- **Repo Local**: almacena las modificaciones que fui haciendo dentro de mi máquina.
- **Repo Remoto**: almacena una versión estable del sistema que se está desarrollando. ¡Es donde va a estar el código accesible para todos!



Estructura del sistema de versionado "Git"

Una vez que los cambios realizados de forma local están funcionando correctamente, entonces se los sube al servidor remoto para que puedan ser compartidos, permitiendo que otras personas dentro del mismo grupo de desarrollo puedan acceder a esas modificaciones.

Cada usuario va trabajando dentro de su repositorio local (modificando, agregando o eliminando archivos), y una vez que se llega a un punto estable (no rompí nada de lo que había y lo que agregué funciona bien), entonces es un buen momento para subirlo al repositorio remoto. Para esto hay que seguir los siguientes pasos:

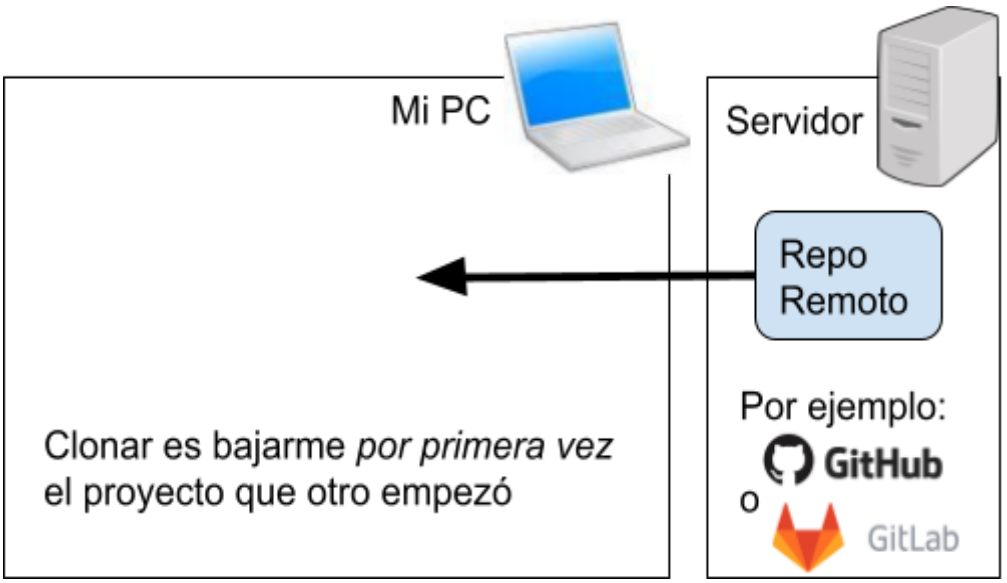
- Elegir qué cambios de los que hice a mis archivos quiero guardar en el repo local.
- Hacer un commit: esto guarda los cambios en el repo local.
- Hacer un push: esto pasa lo que hay guardado en el repo local al repo remoto. Este es el único paso que requiere conexión a internet ya que envía información, mientras que los 2 primeros pasos son locales.

Para usar git existen varios softwares, tanto de interfaz gráfica como por línea de comando. A continuación vamos a ver cómo compartir un proyecto simple en github. Para esto necesitan tener una cuenta creada en https://github.com/ y pasarle el usuario al docente a cargo del curso para que les pueda dar permisos para acceder al repositorio.

2. Instalar Plugin de Git en Wollok

A partir de la versión 1.6 Wollok trae el plugin eGit instalado por defecto, así que no es necesario ningún paso de instalación. Ahora vamos a traernos el proyecto remoto con el que queramos trabajar.

3. Clonar un proyecto usando Eclipse

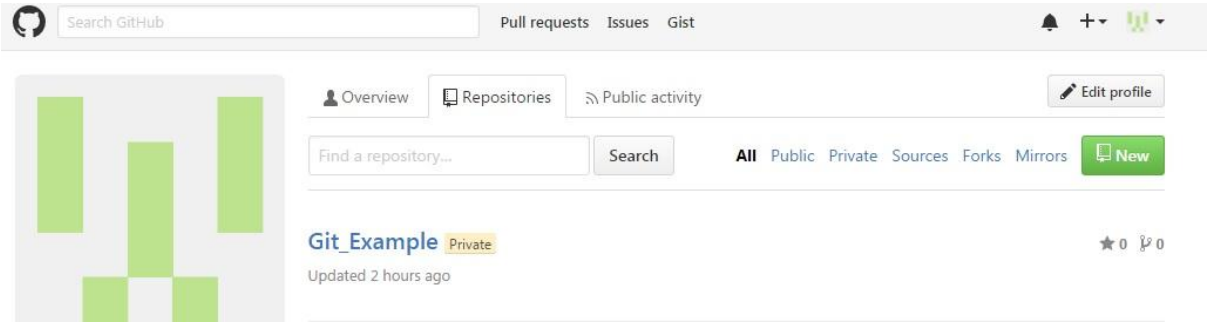


Clonarse un proyecto

¿Dónde está lo que quiero clonar?

Para poder clonar un repositorio primero hay que obtener el link asociado al mismo, para esto hay que hacer lo siguiente:

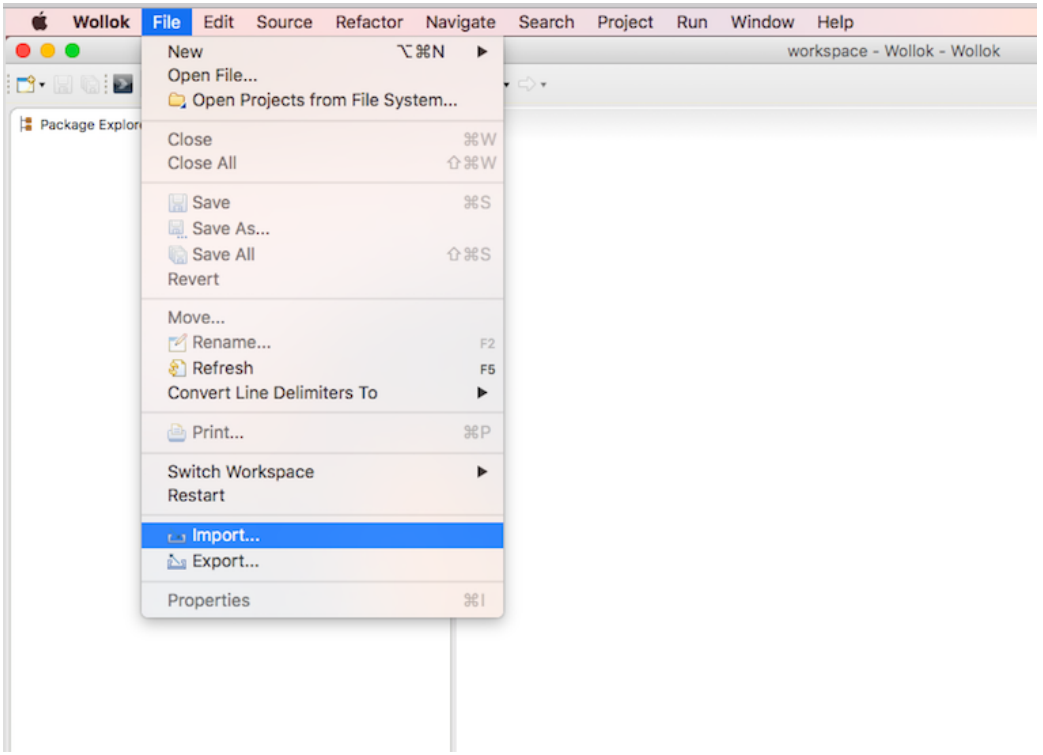
- Entrar a la página de github, ir al perfil de usuario y seleccionar el repositorio que se desee clonar.



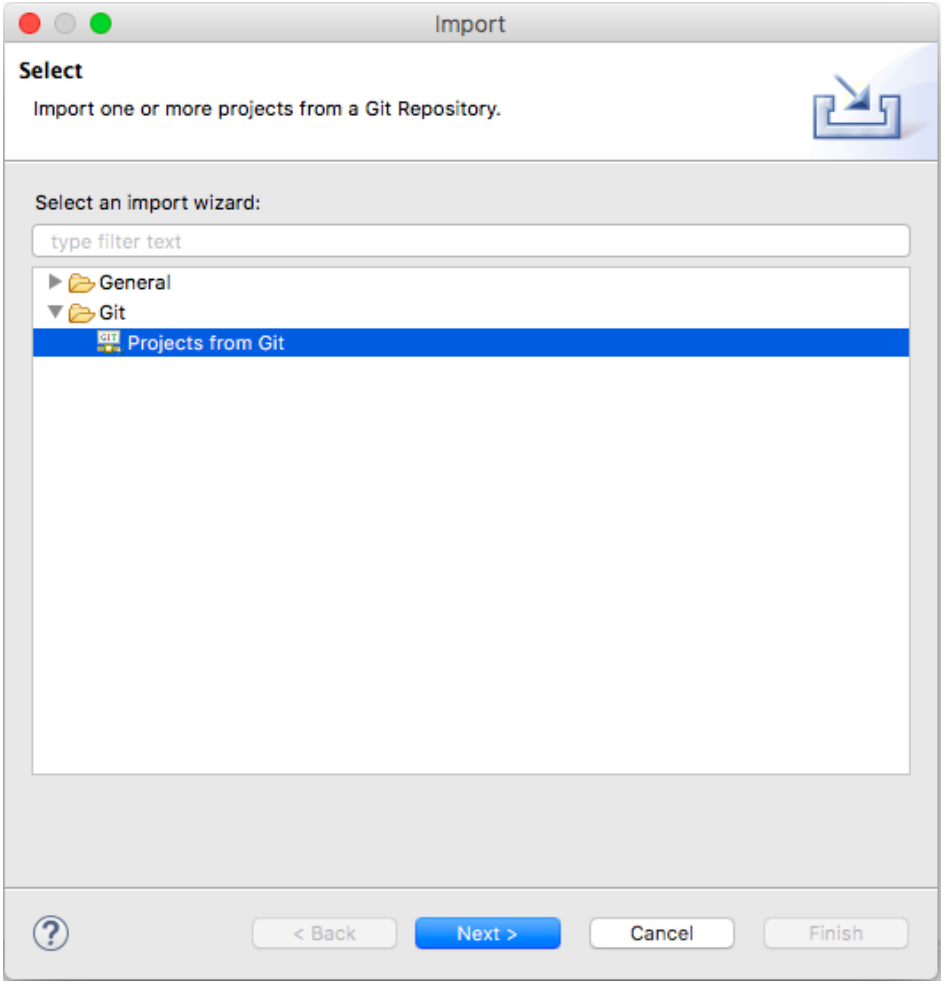
- Una vez dentro, seleccionamos “Clone or Download” y copiamos la ruta que nos aparece.

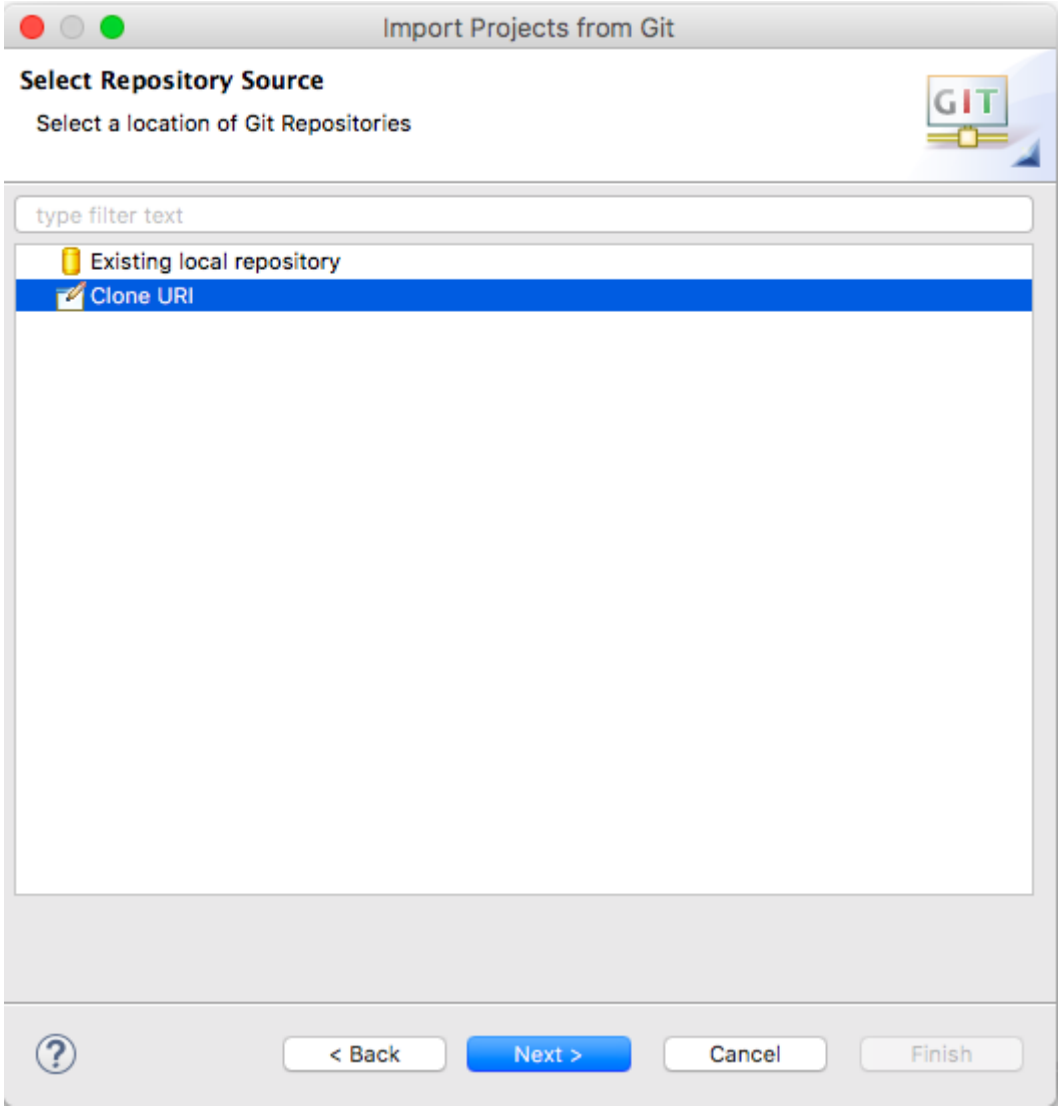
Ahora sí, a clonar

- Para importarlo directo a nuestro workspace necesitamos importarlo directamente desde git. Para eso ir a File -> Import...

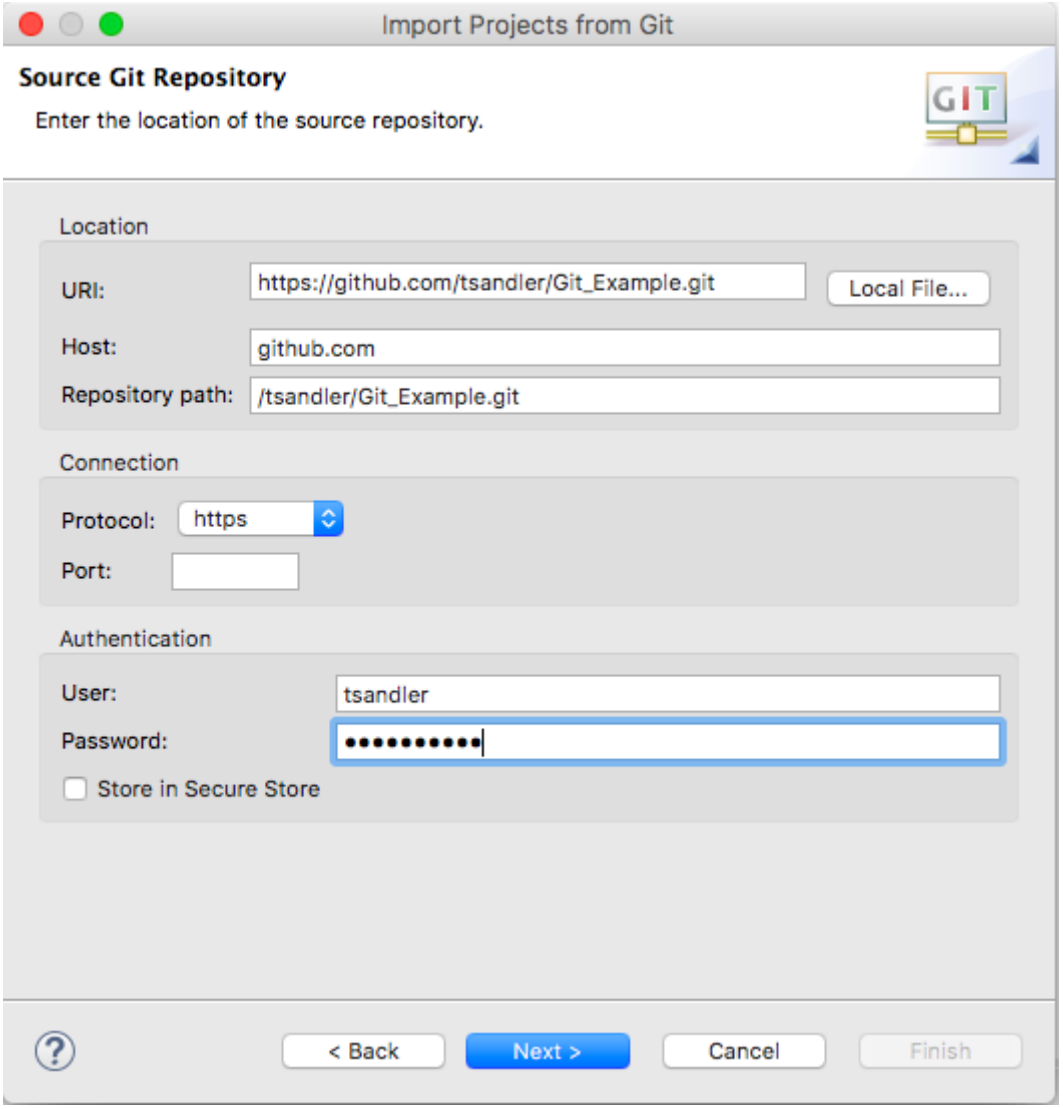


- Luego, seleccionar para importar el proyecto desde git, darle "Next" y elegir la opción "Clone URI"

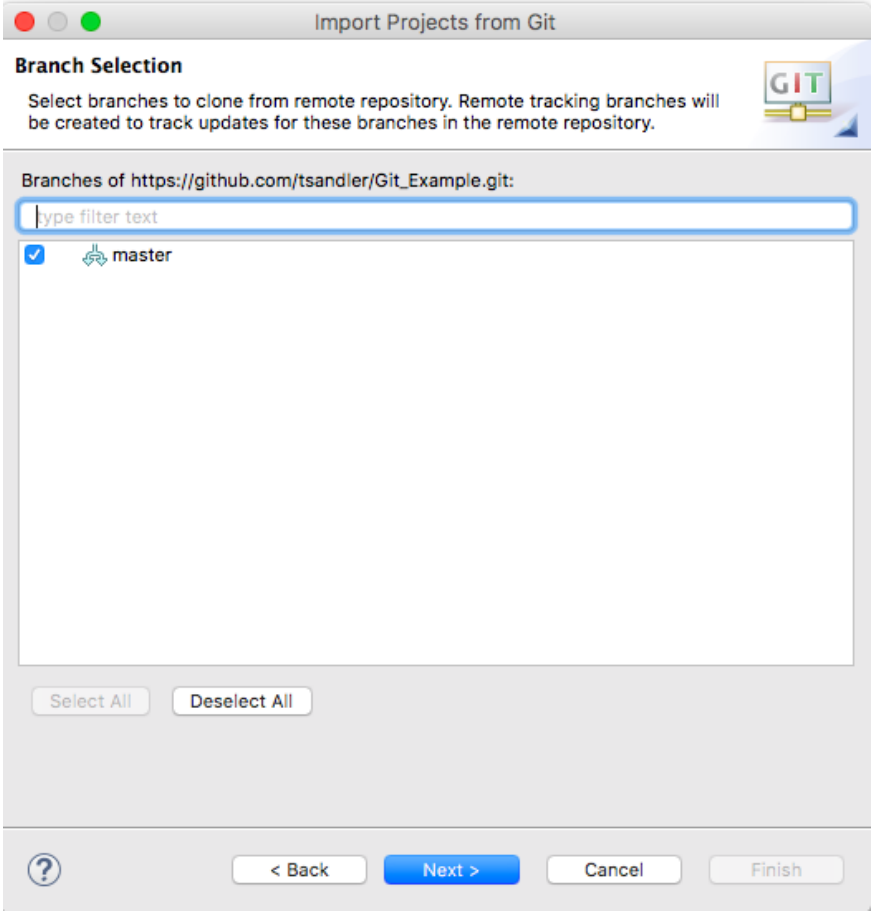



Import Projects from Git
Select Repository Source
Select a location of Git Repositories
GIT
type filter text
Existing local repository
Clone URI
< Back
Next >
Cancel
Finish

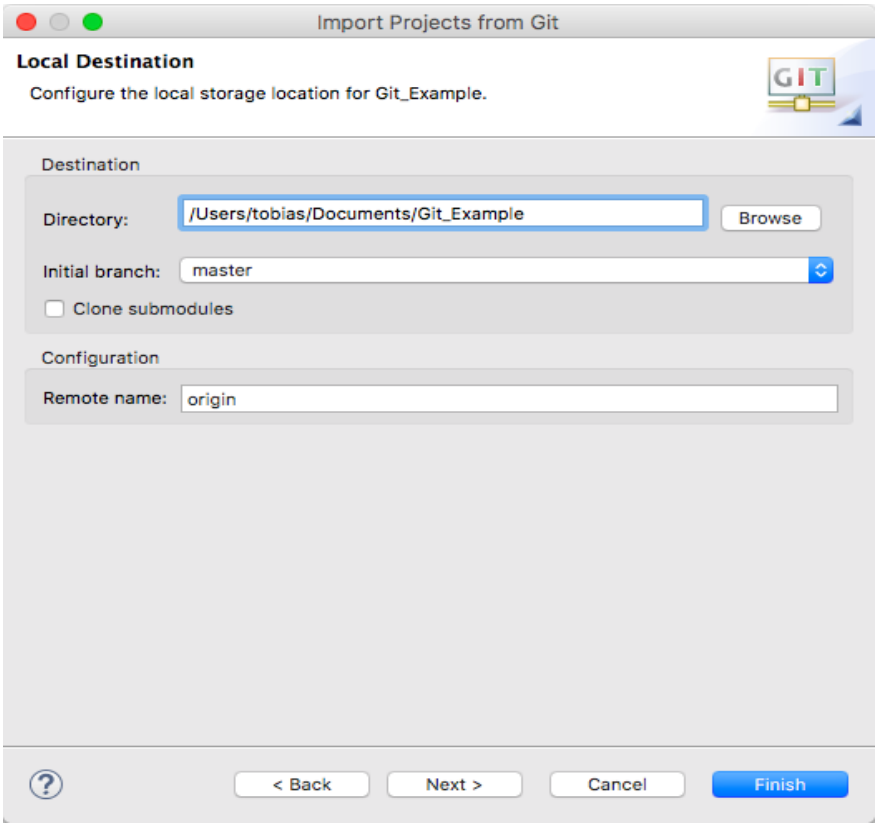
- Ahora, ingresar la dirección que se desea clonar en el campo “URI:” (los campos Host y Repository path son completados automáticamente), el usuario y la password en los campos correspondientes, y presionar “Next”.



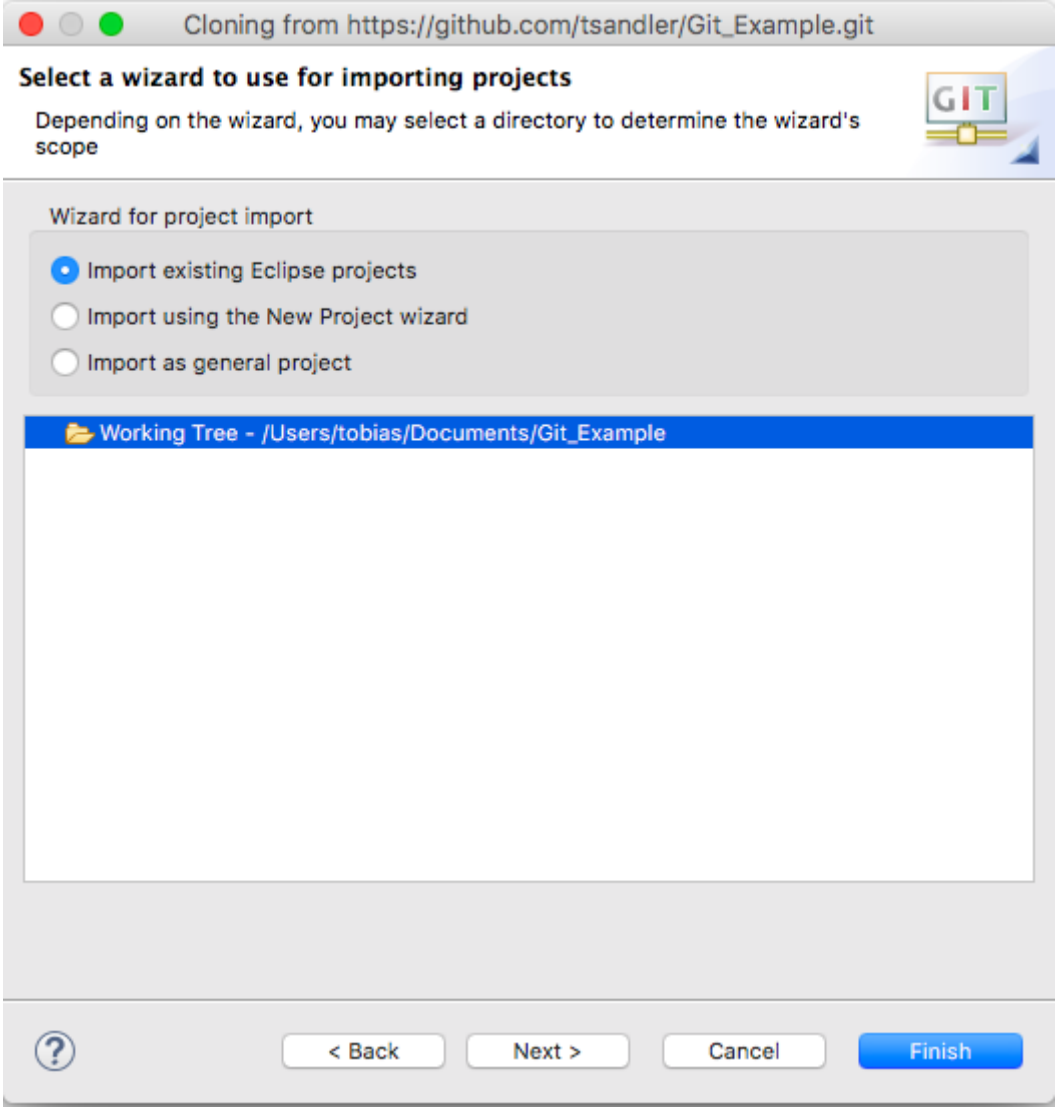
- Luego, nos aparece “Branch Selection”. En otro momento aprenderemos lo que son los branches de Git. Por ahora, con el default (master) estamos. Le damos "Next" nuevamente.



- Luego nos preguntará donde queremos clonar el proyecto. Seleccionan la ruta y le dan "Next".



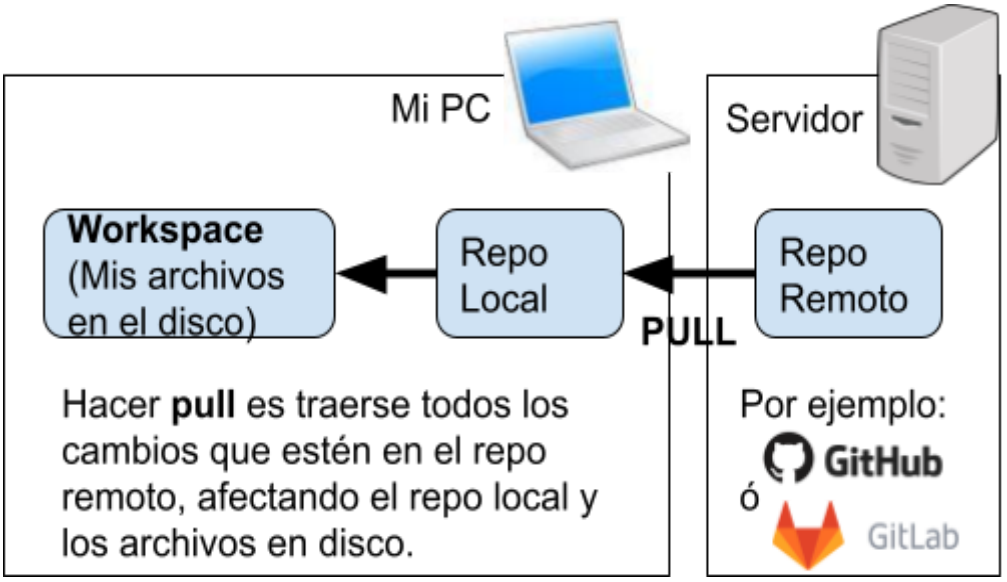
- En el siguiente paso, primero tratará de autenticarlos con el repositorio y, en caso de haber ingresado bien los datos, descargará los archivos remotos. Ahora hay que seleccionar como queremos importar el proyecto a Eclipse, para eso en la siguiente ventana seleccionar "Import existing Eclipse proyect" y darle a "Finish".



Una vez hecho esto ya tenemos clonado el repositorio e importado a nuestro workspace.

Hasta este momento del documento, se muestran cosas que **sólo deben hacer la primera vez**. Una vez que tienen todo esto hecho, no hace falta repetir los pasos.
¡Ya tienen su propia copia del proyecto en su máquina!
A partir de ahora, todo lo que hay que hacer es **commit, pull** y **push**

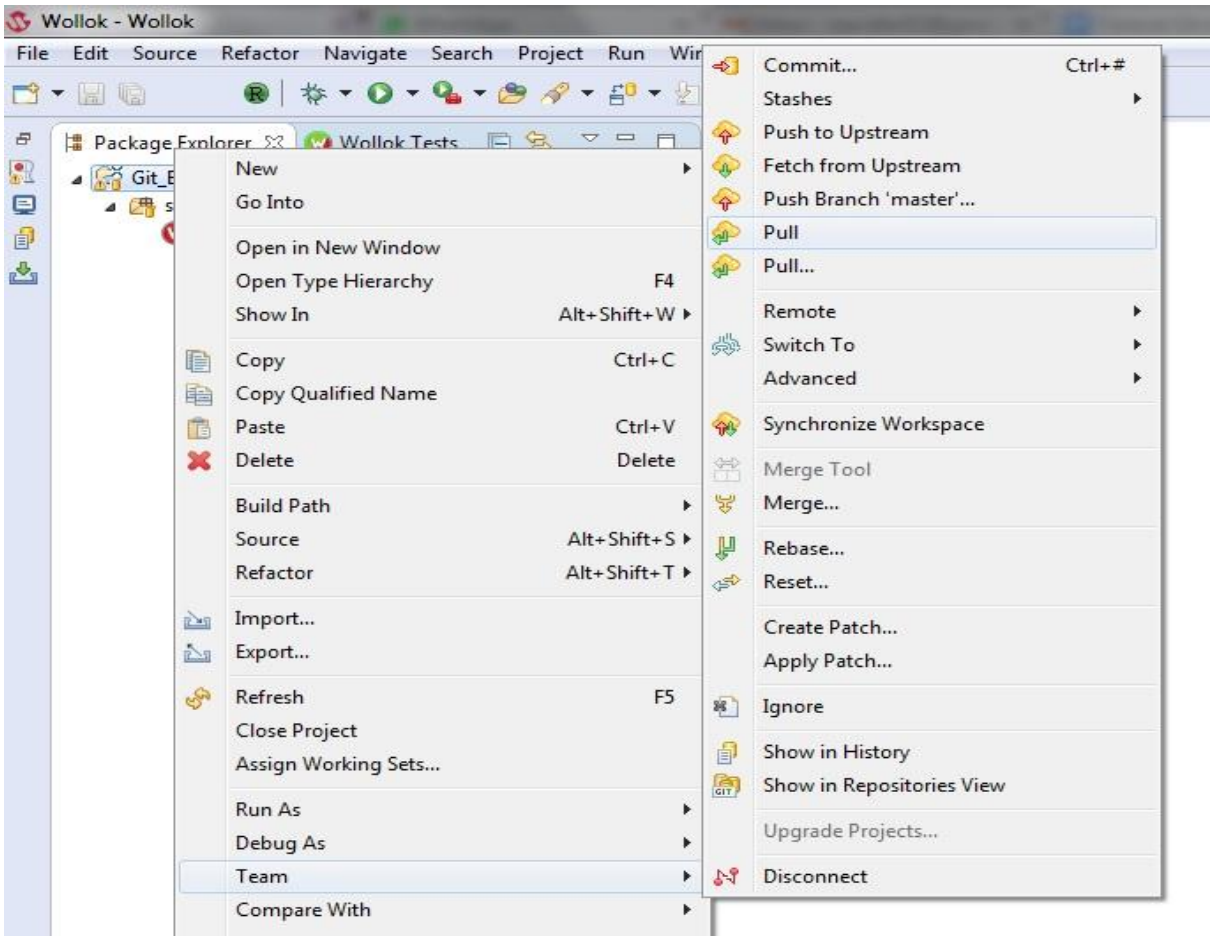
4. Hacer pull usando Eclipse



Qué es lo que hace un pull

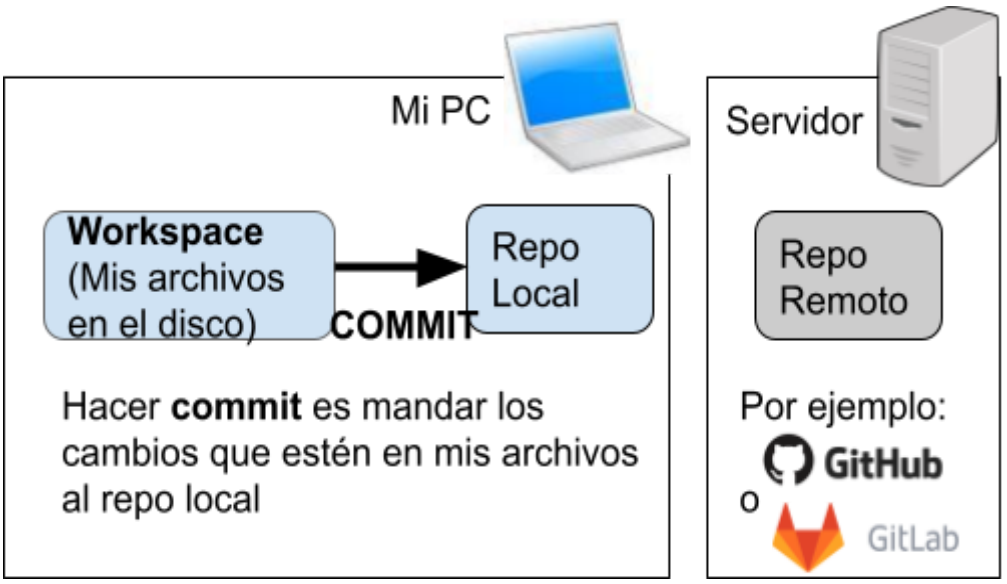
Hacer pull implica descargar los cambios que otro haya realizado en el repositorio remoto y traerlos a mi repositorio local para poder trabajar sobre ellos.

- Para esto hacemos click derecho en el proyecto -> Team -> Pull (seleccionar el primer pull que aparezca porque realiza la descarga por defecto, con el segundo hay que configurar algunas cosas que no nos interesan ahora).

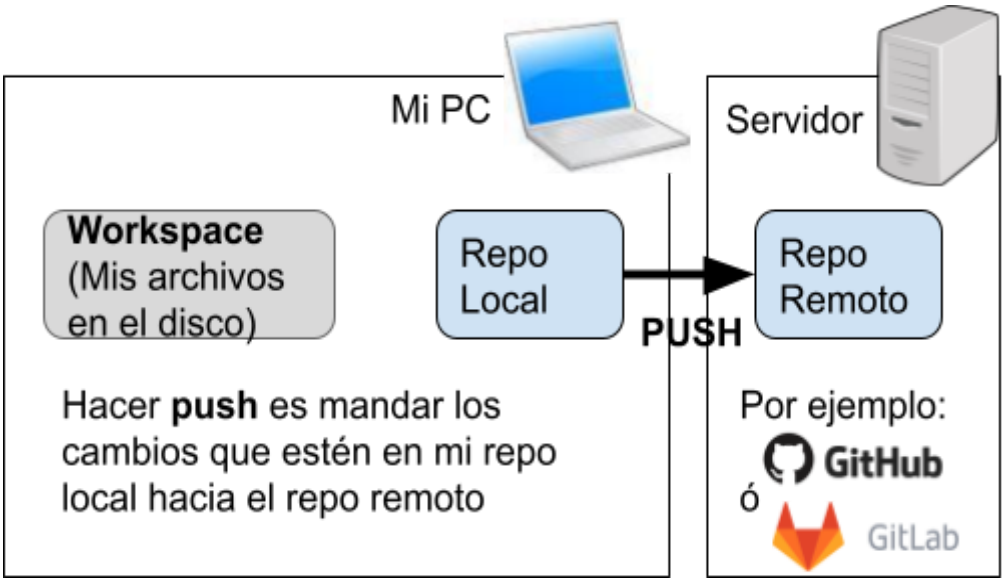


- Ingresar usuario y contraseña y listo, ya van a tener los cambios realizados en el repositorio remoto.

5. Hacer commit y push usando Eclipse



Qué significa hacer commit



Qué significa hacer push

Hacer push implica subir los cambios realizados localmente al servidor remoto.

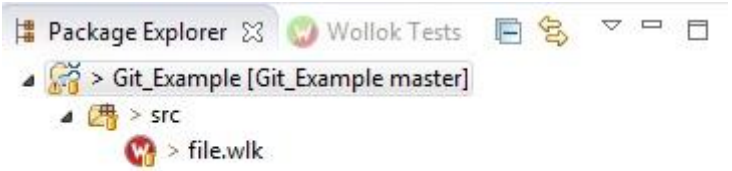
¿Cada cuánto es conveniente hacer un commit?

Esto depende del gusto de cada uno. Lo ideal es que los commits sean chicos y frecuentes, bien enfocados en una funcionalidad que se está desarrollando.

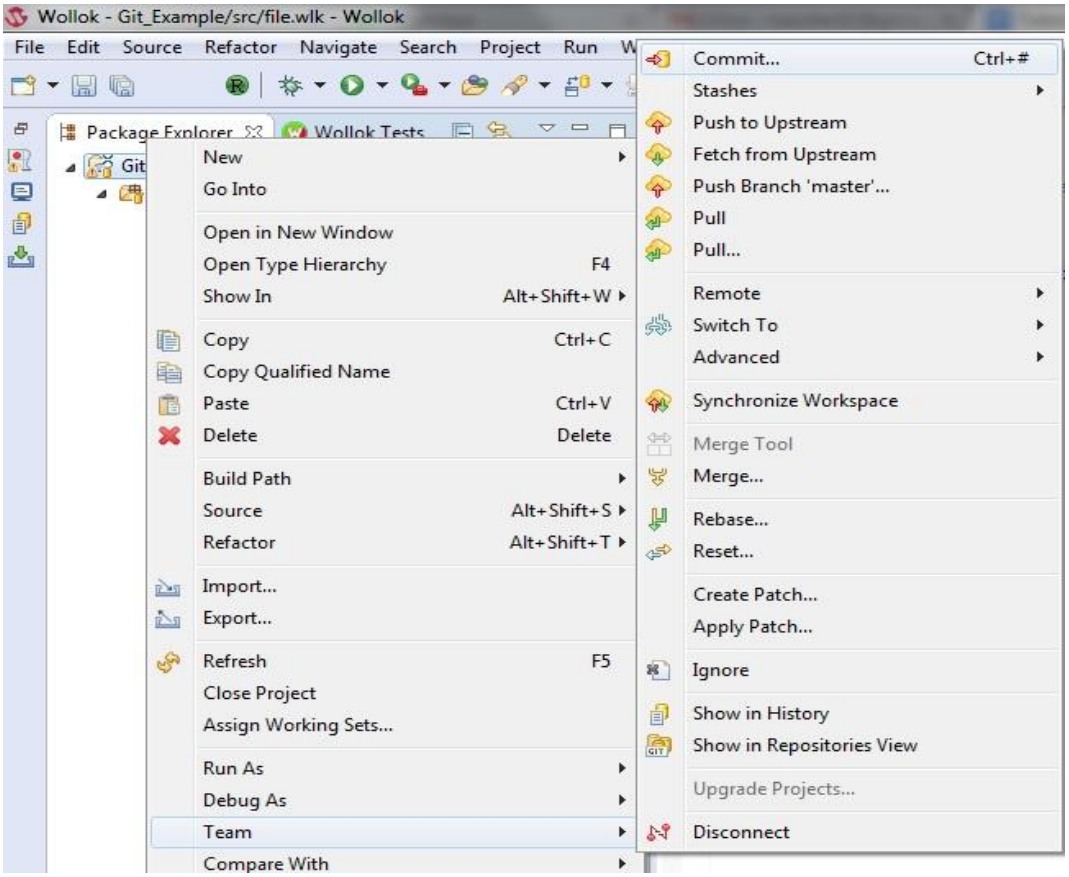
No es conveniente realizar commits constantemente, por cada cambio insignificante, pero tampoco es bueno hacer uno cada 3 días, esto produciría que posiblemente los archivos que se estuvieron modificando hayan sido editados por otra persona del grupo y se complicaría en la hora de subir los archivos. Este tipo de problemas git trata de arreglarlos haciendo un auto-merge, que en el mejor de los casos lo hace bien, pero en caso de que las líneas modificadas sean similares git no puede resolver el conflicto y hay que arreglarlo a mano.

Subiendo archivos al repositorio remoto

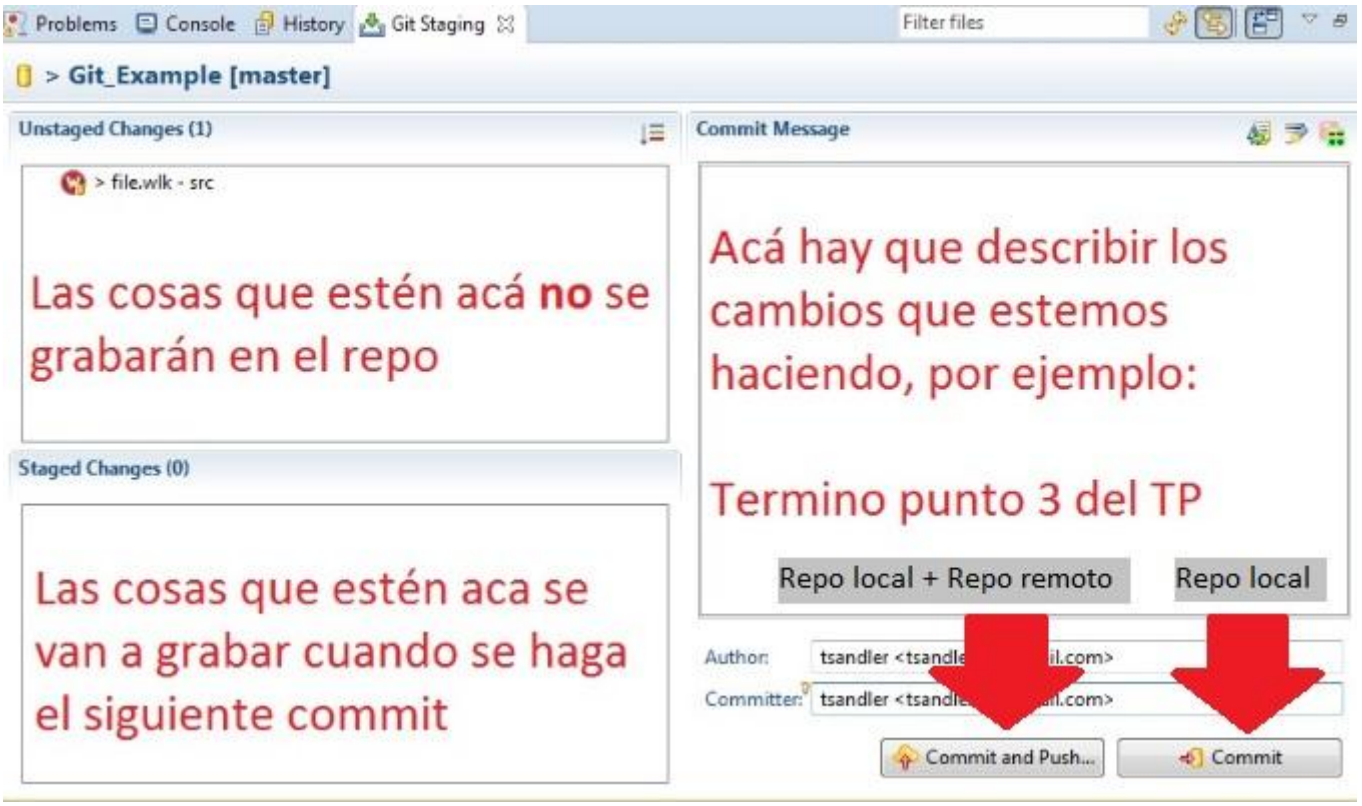
- Inicialmente van a ver que una vez que hayan modificado archivos, los mismos van a aparecer con un “>” adelante del nombre, que indica que no han sido subidos aún.



- Para subirlos hacemos click derecho en el proyecto -> Team -> Commit

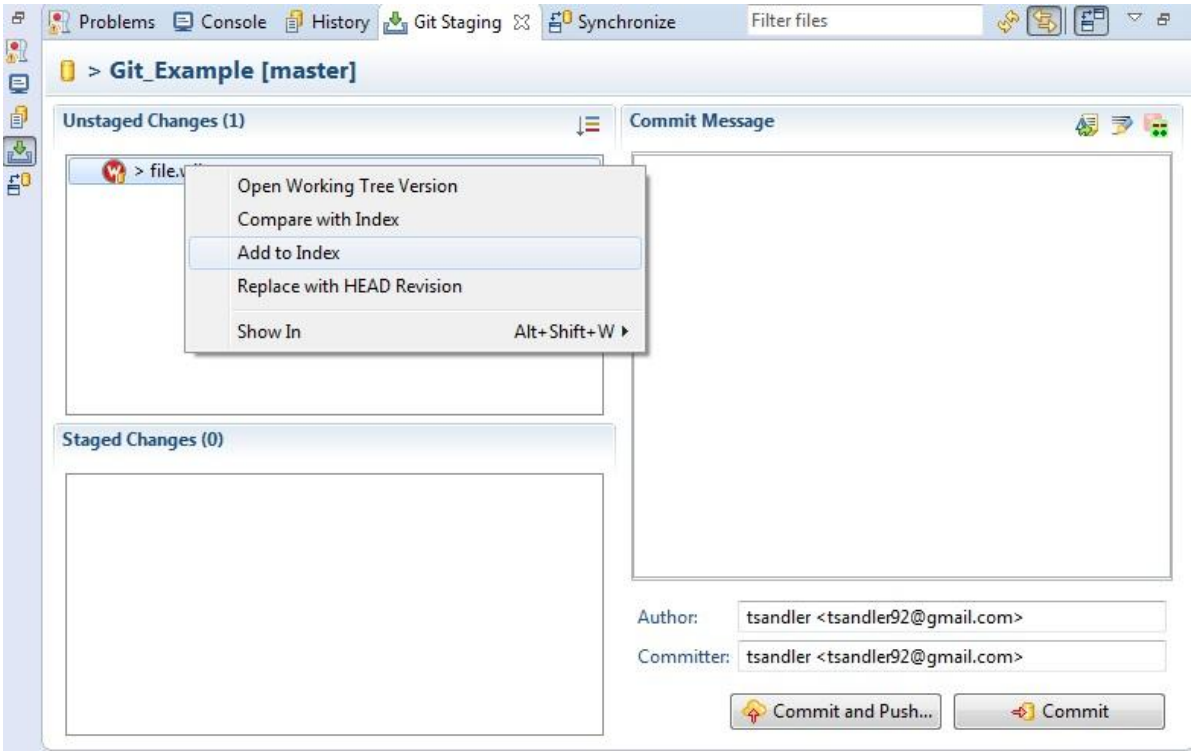


- Una vez hecho esto, verán que aparece una pantalla como la siguiente.

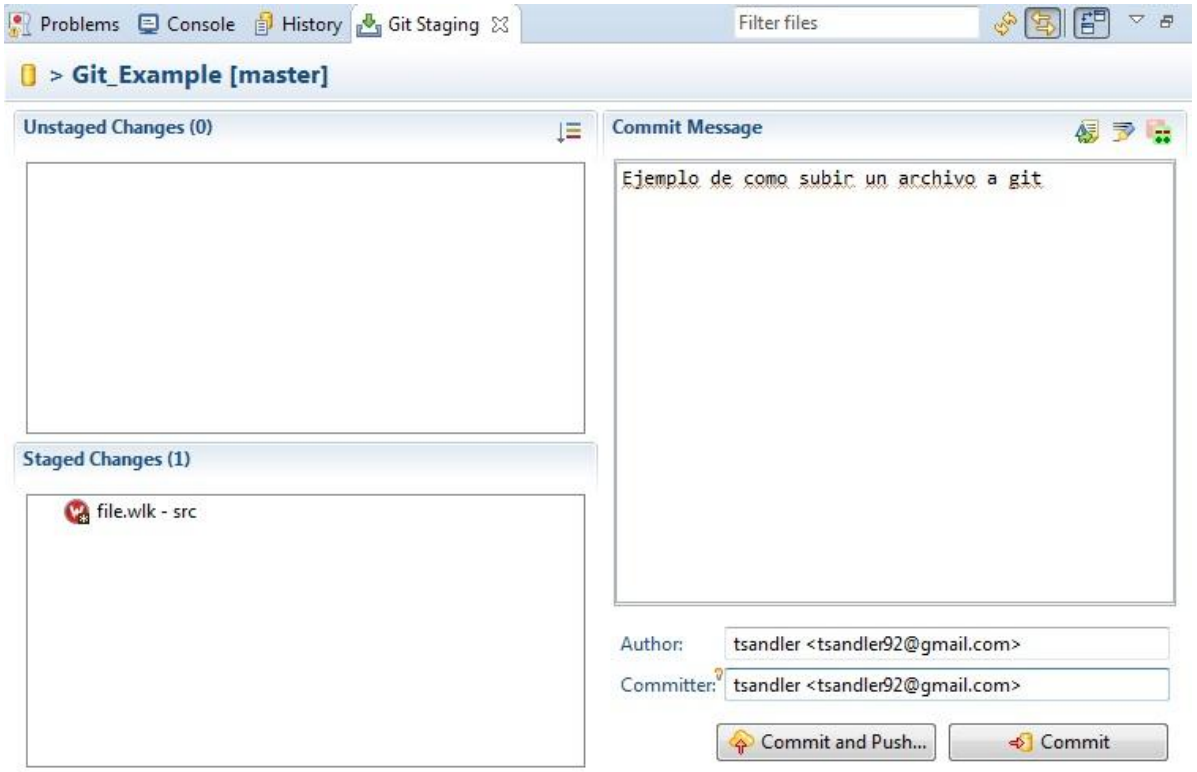


- **Unstaged Changes**: acá se van a listar los archivos que hayan modificados y no hayan sido subidos aún, los archivos presentes acá **no** van a ser enviados en el próximo commit.
- **Staged Changes**: acá se van a listar los cambios que van a ser guardados en el repo, si este listado está vacío entonces no podremos realizar ni commit ni push ya que no hay cambios para guardar.
- **Commit Message**: es un mensaje con el que se van a guardar los cambios en el repo. El mensaje es obligatorio y debería ser lo suficientemente expresivo como para que se entienda qué cambio se está realizando.
- **Commit**: guarda los cambios solo en el repositorio local para ser subidos más adelante. Puede que los cambios no puedan ser enviados al repositorio remoto en el momento del push por diversas razones, como por ejemplo no tener conexión a internet.
- **Commit and Push**: guarda los cambios en el repositorio local y en el repositorio remoto
- **Author y committer**: acá pongan su nombre de usuario (o nombre completo) y mail. Estos son los datos que muestra git por cada commit, sino no puede identificar quién lo hizo.

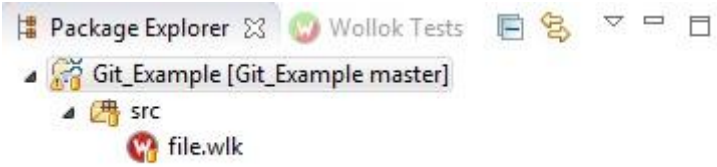
- Para agregar un archivo a la zona de Staged Changes solo hay que hacerle click derecho -> Add to Index



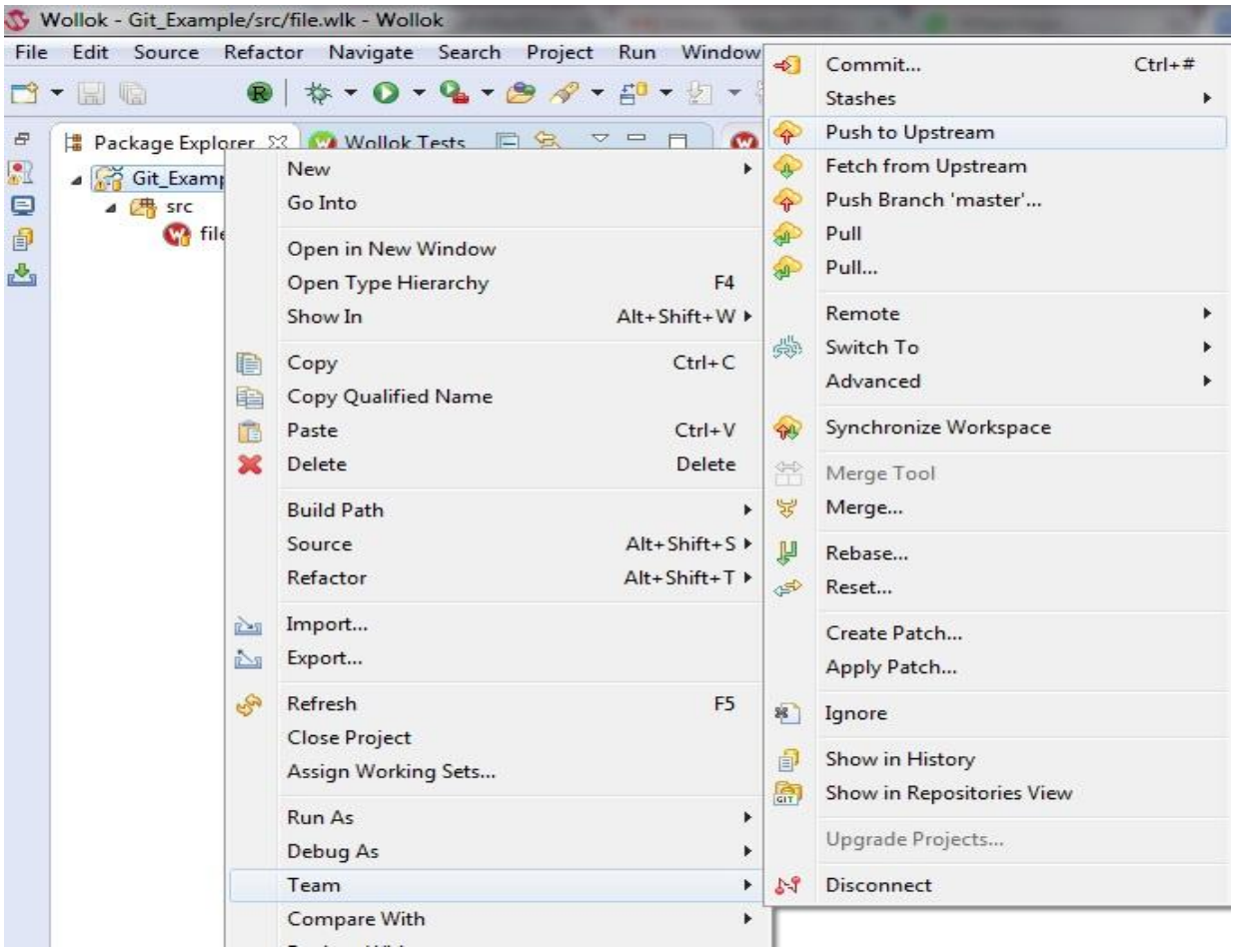
- Verán que el archivo aparece en el listado de Staged Changes, ahora solo se ingresa un mensaje para el commit, se selecciona “Commit and Push…” ó “Commit”, ingresan usuario y contraseña y listo.



- Una vez hecho esto podemos verificar que los cambios se guardaron en el repo ya que el “>” que aparecía delante del nombre de los archivos debió desaparecer.



- En caso de haber seleccionado “Commit and Push” los cambios ya fueron enviados al repositorio remoto, ahora si solo seleccionaron “Commit”, todavía falta enviarlos. Pueden hacer la cantidad de commits que prefieran y enviar todos juntos con un solo push, no hay un máximo posible (no es recomendable hacer demasiados porque puede pasar que en el medio pude subir algo y por no hacerlo otro tocó mi mismo archivo, generando un conflicto que voy a tener que resolver).
  Para subir lo que está en el repositorio local al remoto hay que darle click derecho al proyecto -> Team -> Push to Upstream, ingresan usuario y contraseña y listo.



6. Deshacer mis cambios para hacer pull

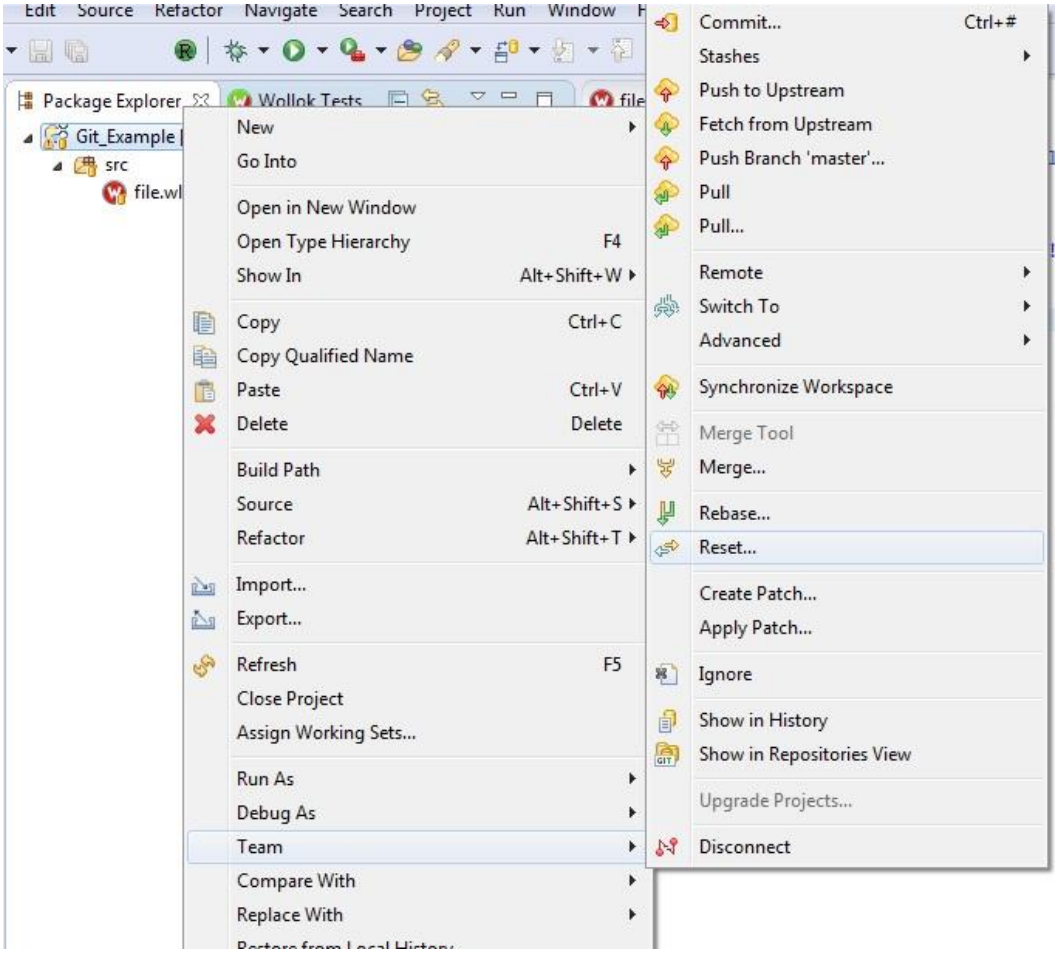
¿Qué pasa si toco uno o más archivos produciendo cambios que posteriormente no me interesa conservar? Este es un problema bastante común, porque puede pasar que yo realice cambios que posteriormente no funcionan o simplemente no quiero mantener. Con decir “esto no me interesa, hago pull y fue” no alcanza, porque git detecta que hubo cambios en tu workspace, por lo que no va a poder descargar los cambios del repo normalmente. La solución **no** es borrar todo y volver a clonar, eso implicaría volver a preparar todo el entorno de trabajo. ¿Tiene sentido hacer esto? Claramente no.

Para solucionar este problema hay dos alternativas:

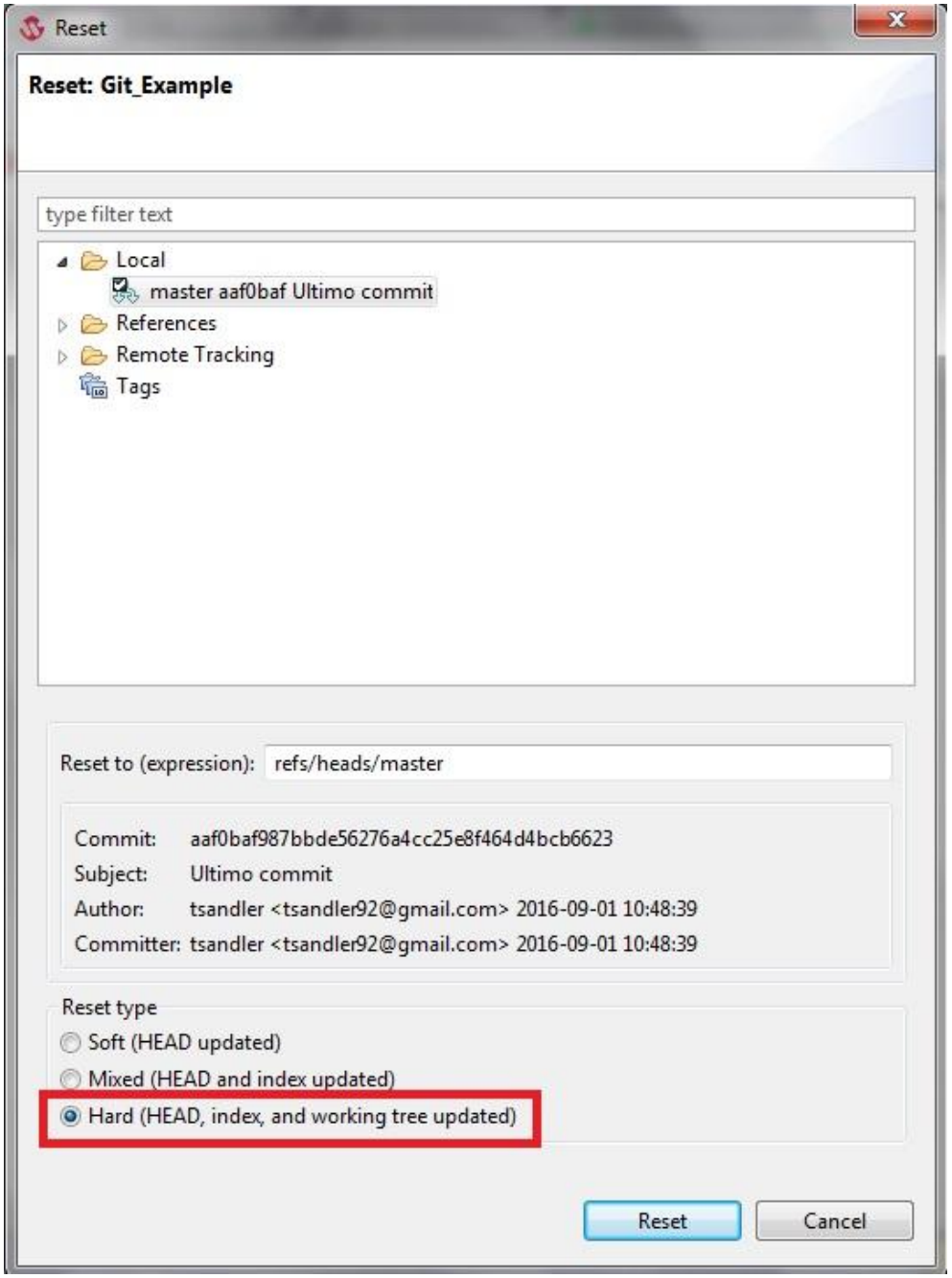
- Hacer un stash: esto sería prácticamente como hacer un Ctrl + X de mis cambios, que puedo posteriormente volverlos a agregar en el código actualizado (entre otras cosas).
- Hacer un checkout del estado del repo en el último commit que tengo: esto implica **borrar** los cambios hechos localmente y **no** hay vuelta atrás, llevándome al último commit hecho, permitiéndome así bajar los nuevos cambios en caso de que haya.

Por el objetivo que tiene este apunte, vamos a ir por la segunda opción ya que la primera tiene más complejidad:

- Primero hay que darle click derecho al proyecto -> Team -> Reset…



- En la ventana que aparece, seleccionar “Hard (HEAD, index and working tree updated)”, darle a “Reset” y aceptar la ventana emergente que sigue.



Como se puede ver, nos indica a qué commit estaríamos retrocediendo, mostrando el código del commit, el mensaje con el que se hizo y quién lo subió.

7. Flujo de trabajo simple

Como resumen, te recordamos que se recomienda siempre el siguiente flujo de trabajo:

1) Bajo cambios (git **pull**)
2) Corro **tests**, si están en rojo hay que avisar al equipo que hay un commit que hizo fallar. Medialunas para el equipo a cargo del que subió el commit.
3) ¡A **trabajar**! Hago ajustes necesarios.
4) Cuando terminé mis modificaciones (incluyendo nuevos tests si corresponde)... ¡¡volver a correr los **tests**!!
5) Ahora sí, hago un **push**

Y ya puedo irme a dormir tranquilo.

Resumen del resumen:

Me **bajo** lo último
corro **tests**
hago mis **cambios**
corro **tests**
subo mis cambios

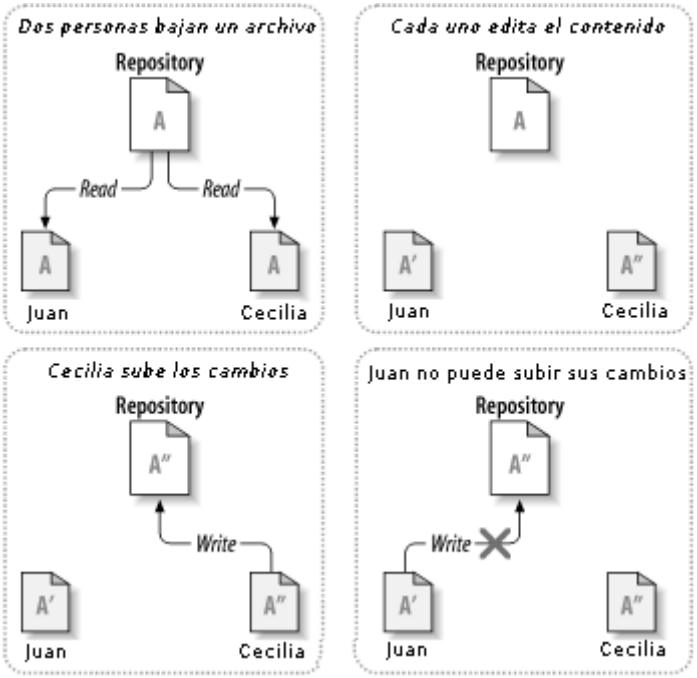
Pueden proceder a tatuarse esto.

Ah, una salvedad: el último paso puede no ser feliz. La siguiente sección aborda la pregunta:

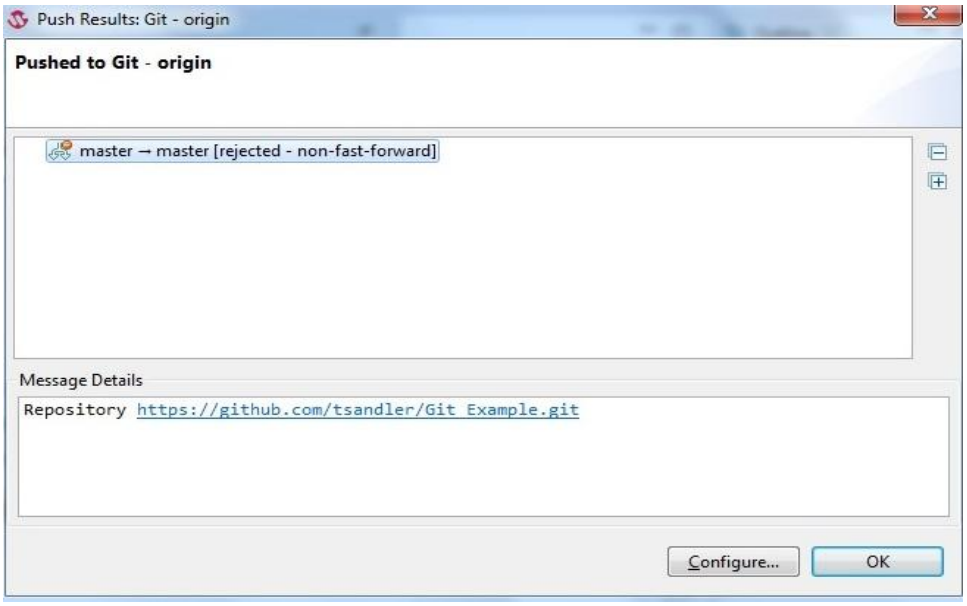
¿Qué pasa si alguien estuvo trabajando **al mismo tiempo** que yo?

7. Resolución de conflictos usando Eclipse

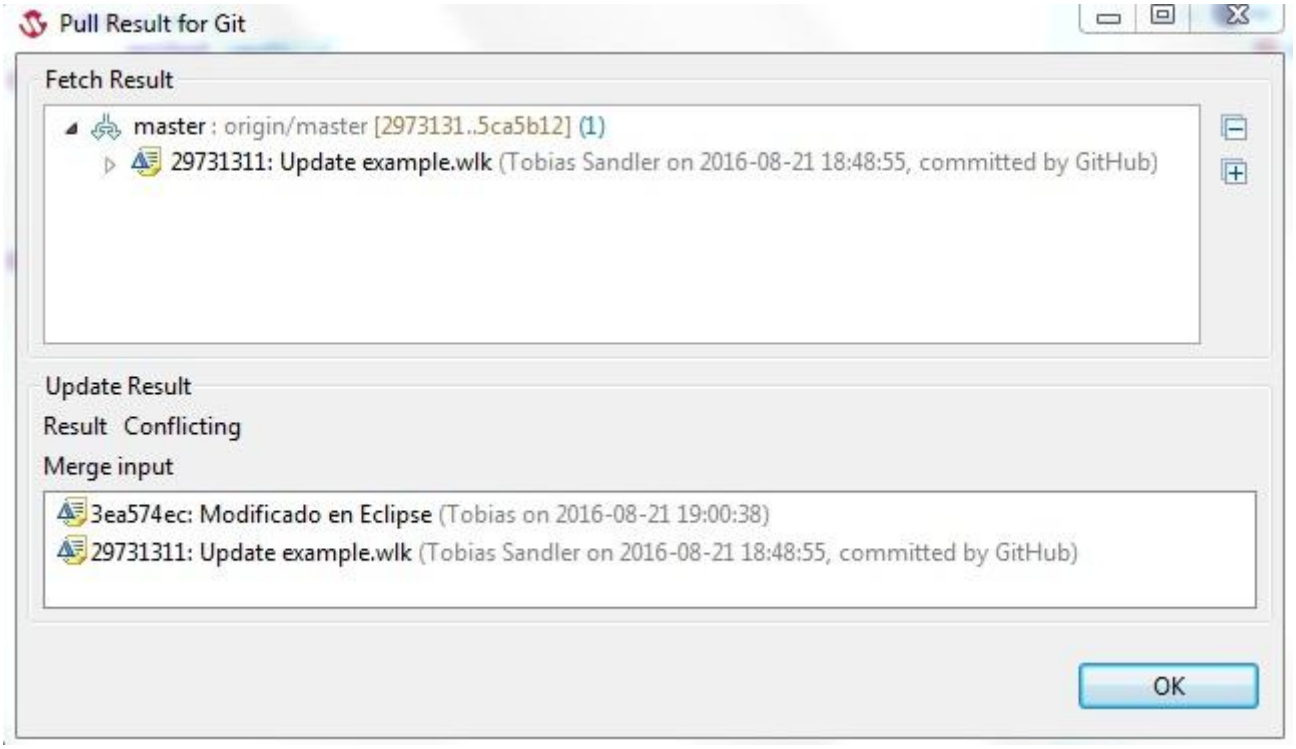
¿Qué ocurre si dos personas tocan simultáneamente el mismo archivo?



En este ejemplo se puede ver que tanto Juan como Cecilia descargaron el archivo A y estuvieron trabajando sobre él paralelamente. A su vez, Cecilia trabajó más rápido y subió sus cambios al repositorio. Entonces Juan al querer subir su versión se entera de que hay una nueva en git, por lo que no va a poder subir sus cambios con un simple commit.
Al querer subirlo posiblemente aparezca un cartel como el siguiente:



Este cartel nos indica que hubo un problema al querer subir los cambios al repositorio remoto.
Para solucionar este problema lo que debemos hacer es tener la última versión del archivo, para eso primero hay que hacer pull normalmente. Una vez hecho eso, van a ver el siguiente cartel:



Aquí se puede observar que hubo un problema de merge, y lo que git hace automáticamente es “romper” el archivo para que sea arreglado manualmente, quedando así:

```
object ex {
    method sayHi(){
<<<<<<< HEAD
        console.println('Modificado en Eclipse')
=======
        console.println('Modificado en github')
>>>>>>> branch 'master' of https://github.com/tsandler/Git_Example.git
    }
}
```

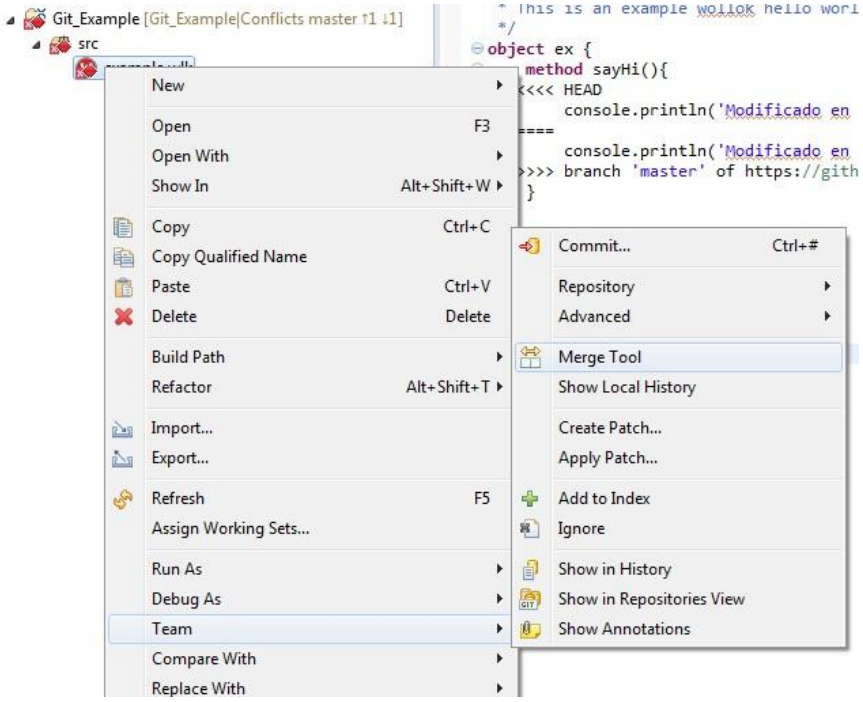
Lo que podemos ver a simple vista es que el programa **no compila**, y eso es con el único fin de que podamos hacer un merge manual del archivo.
Lo que viene luego de “<<<<<<< HEAD” es el código que tenemos en nuestro repositorio local, mientras que lo que viene luego de “=======” es lo que está en el repositorio remoto. Ahora debemos unificar los archivos, modificando o borrando lo que sea conveniente, dejando al archivo nuevamente sin errores, sin problemas de compilación y, claramente, con los tests correspondientes funcionando.

También podemos ver que la estructura del proyecto en el Package Explorer cambió, indicando los archivos que tuvieron conflictos, mostrándose así:

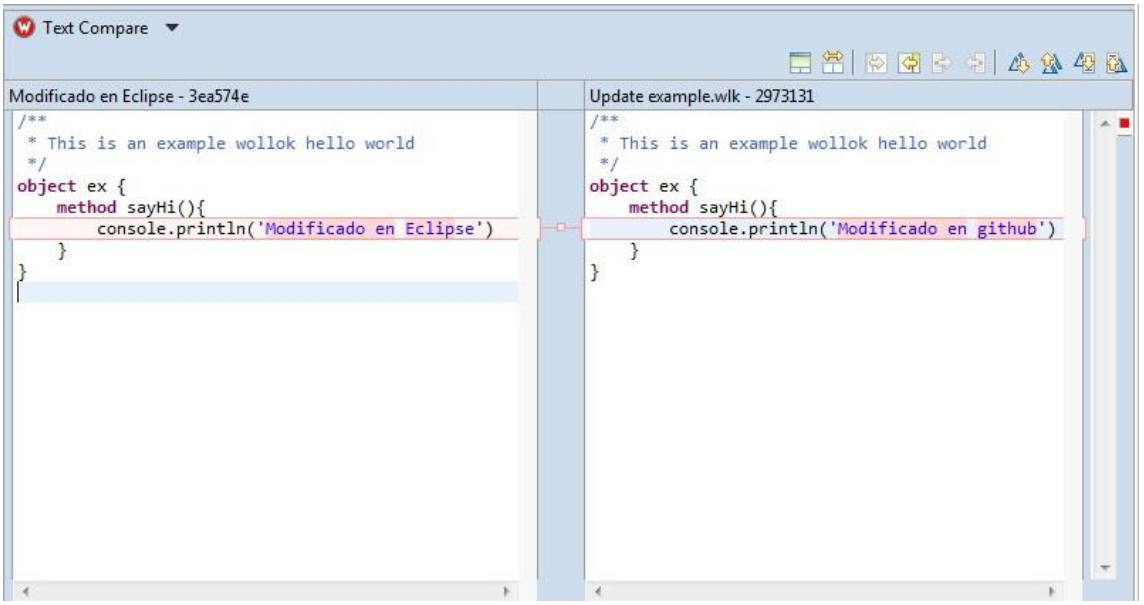
Una vez solucionado este problema se procede a realizar un commit y push normal hacia el repositorio y problema solucionado.

Opción alternativa:

Otra alternativa para solucionar este tipo de conflictos es usar una herramienta que provee Eclipse para tal fin. Para eso hay que hacer click derecho en el archivo -> Team -> Merge Tool.



Luego de hacer eso veremos que se nos abre una ventana como la siguiente:



Acá podemos ver a simple vista los cambios que están en nuestro repositorio local (archivo de la izquierda) y los que están en el repositorio remoto (archivo de la derecha). El objetivo con esta vista es dejar el archivo de la **izquierda** tal y como queremos que sea el archivo final luego del merge.
Una vez hecho esto, le dan click derecho -> Save y les va a guardar los cambios en el archivo correspondiente.
Y ahora que está terminado el merge se puede subir al archivo con un commit y push normal.

8. Flujo de trabajo completo

Un flujo de trabajo completo sería:

1. Hacer pull.
2. Correr tests.
3. Hacer mis cambios.
4. Correr tests.
5. Hacer un **push**,
    a. Si está todo bien ¡Terminamos!.
    b. Si no me deja pushear, entonces hacer lo siguiente:
        - Hago un pull para traerme los cambios nuevos.
        - Corro tests
            - Si está todo bien entonces no hubo conflictos.
            - Ahora, si hay conflictos, ups, hay que resolverlos a mano y correr los tests nuevamente
        - Commitear y pushear una vez que los marcaron como resueltos

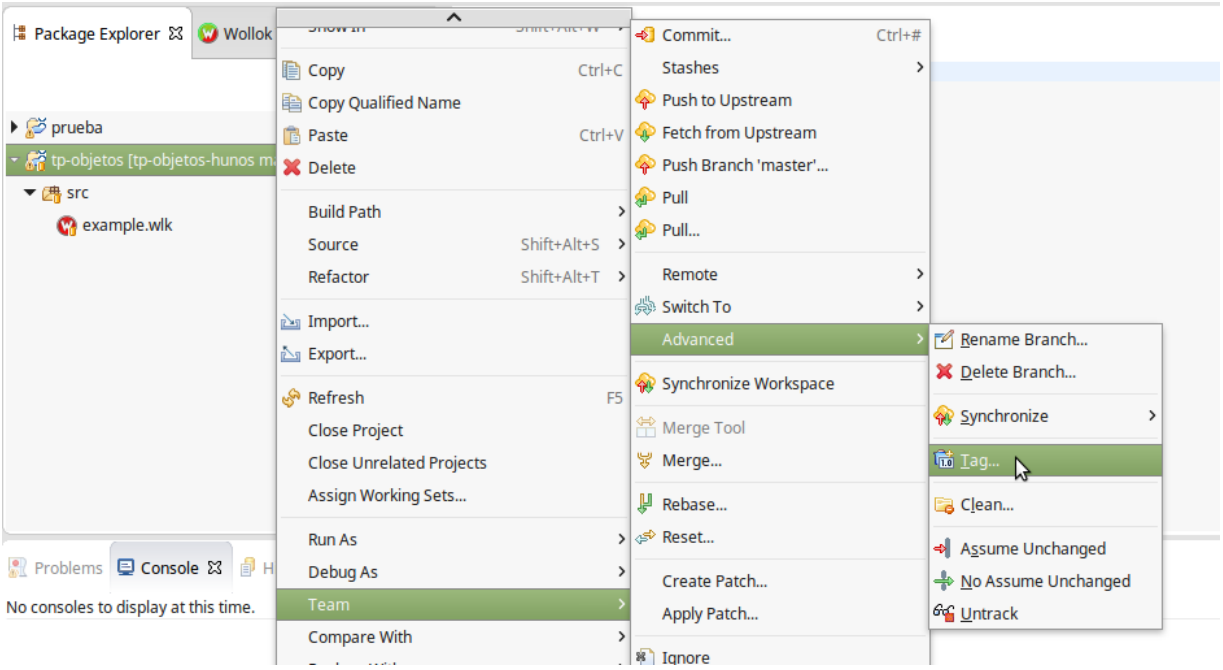
9. Etiquetar un commit (tag)

Cuando realizamos un commit, que es el último de una versión entregable de nuestro programa, nos interesa recordarlo. Pero guardar el nombre del commit es: poco expresivo, complejo -debería llevar en algún otro lugar una tabla que diga *entrega -> commit-*.

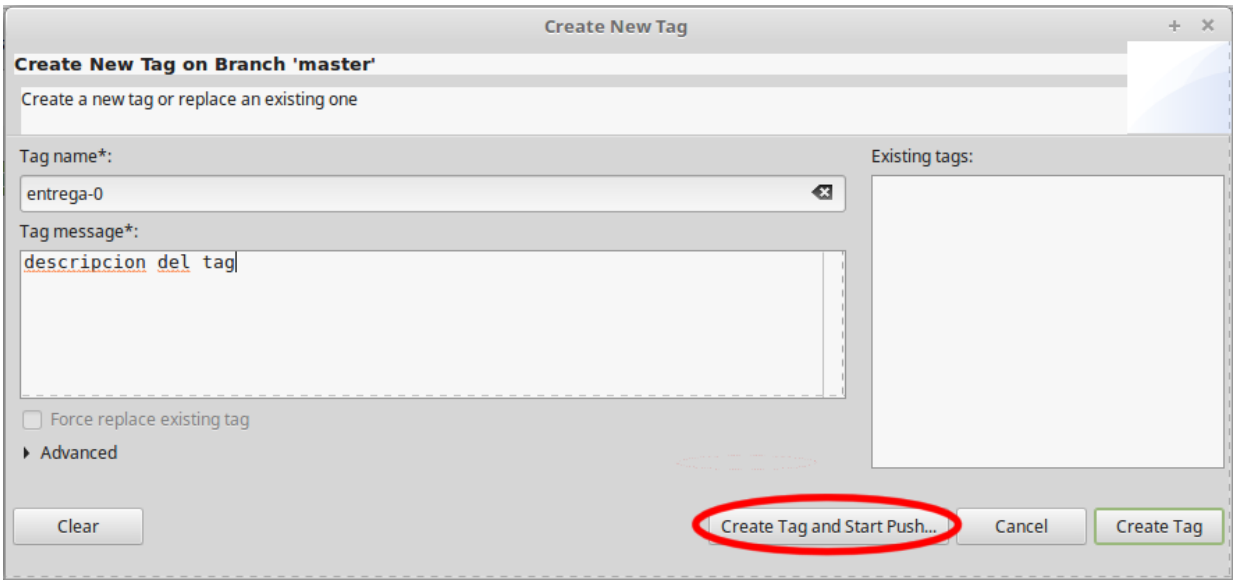
Para esto GIT nos da la funcionalidad de etiquetar (dar un nombre) a un commit particular.

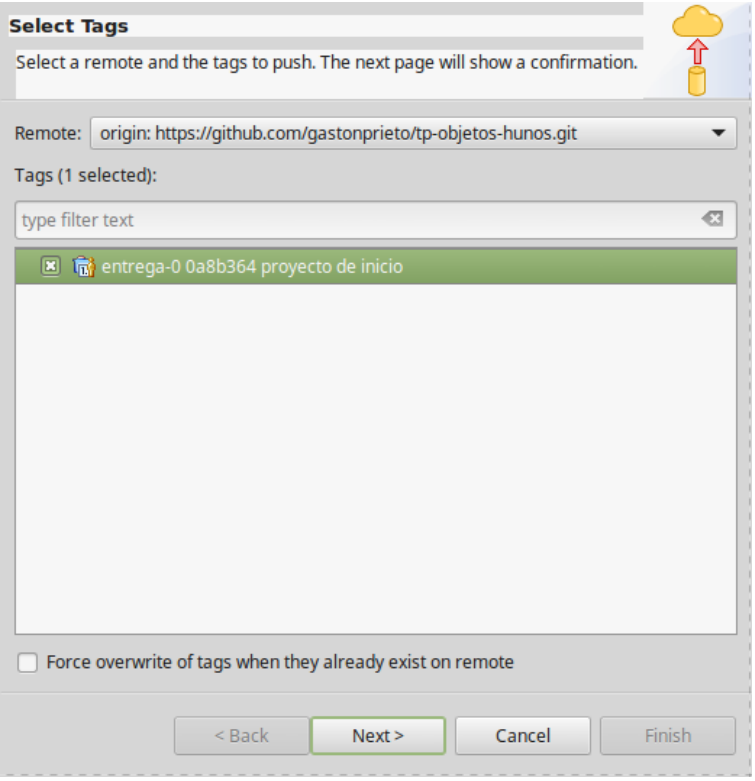
Se realiza de la siguiente manera:

- **Crear el tag:** *Botón derecho sobre el proyecto -> Team -> Advanced -> Tag*
  Nota: Siempre hacer el tag después de haber hechos todos los commits necesarios y cuando se está seguro que se va a entregar esa versión (no genera commits nuevos).

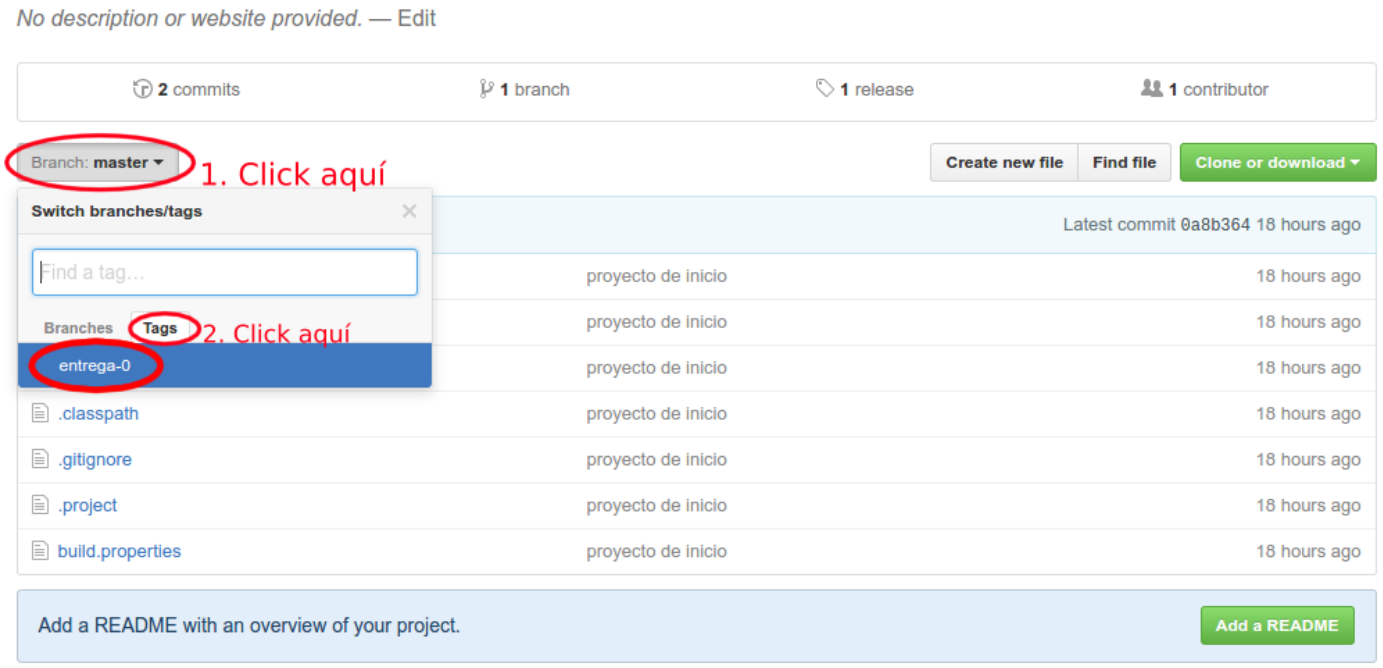


- Agregamos la información referente al tag y luego presionamos *"Create Tag and Start push"*.



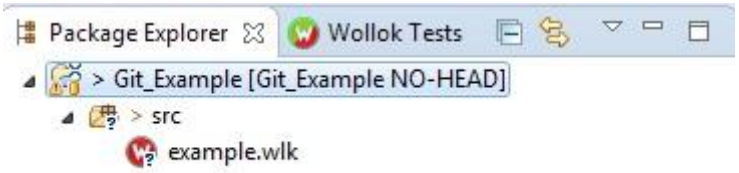


- Esto en GitHub se visualizará de la siguiente forma.



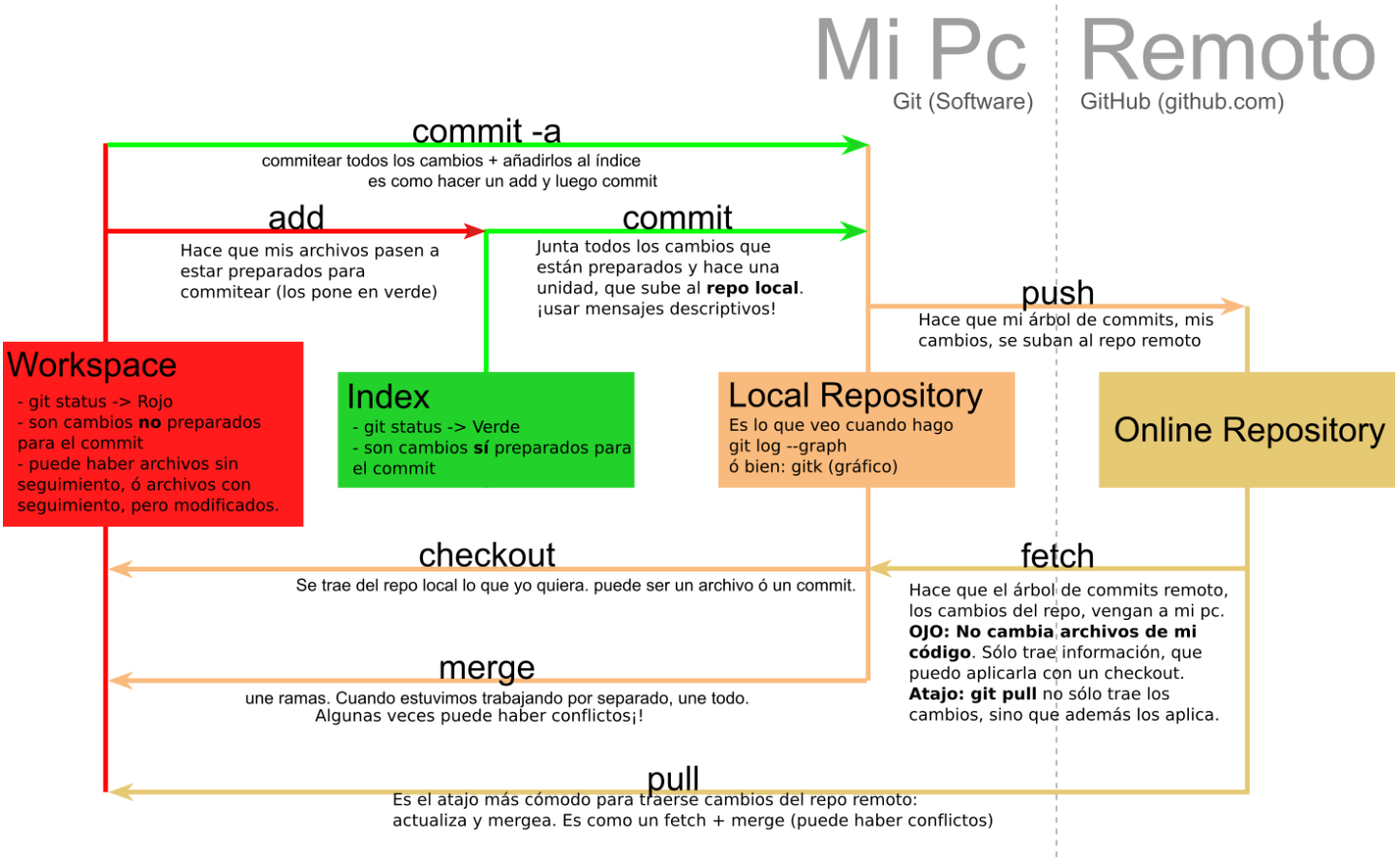
10. Significado de los símbolos

En el project pueden aparecer cosas gráficas a medida de que vayan desarrollando, como por ejemplo algo asi:



- El “>” que aparece antes del nombre del proyecto significa que los cambios locales difieren de los remotos (o sea, que tenemos cosas para **pushear** ó para **pullear**).
- Lo que aparece entre corchetes indica la rama en la que se trabaja (no entraremos en detalle sobre esto, en la cursada se va a usar la principal solamente).
- Los archivos o carpetas que tengan un “?” quiere decir que no existen en el repositorio remoto.
- El cilindro naranja ubicado abajo a la derecha indica que está vinculado con un repositorio remoto.

Extra 1: Terminología completa de Git



Extra 2: Equivalencias con la línea de comandos

¡Git se puede usar por línea de comandos!
De hecho, la mayoría de los desarrolladores lo hacen. Para eso se necesita tener una consola que entienda los comandos:

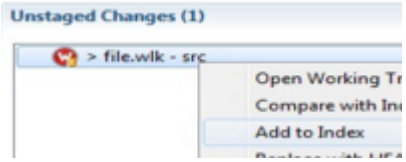
- En Ubuntu directamente desde la consola hay que correr “sudo apt-get install git” y debería descargar todo lo necesario para que funcione correctamente.
- En Windows y Mac recomendamos descargar una consola que se llama git bash.

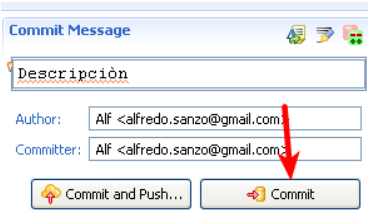
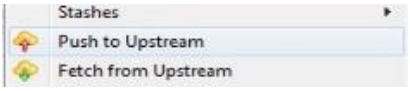
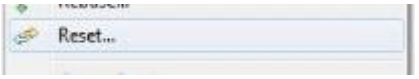
Aquí un pequeño resumen de los comandos de git comparados con el plugin de Eclipse EGit.

Descripción	EGit	Línea de comandos
Clonar el proyecto (bajar el proyecto por primera vez)		`git clone <ruta del repo>`

Cambiar usuario y mail para todos los commits que se realicen en la computadora (para cualquier repositorio).	
	Esto solo lo realiza localmente, para hacerlo así borrar el “--global”
		`git config --global user.name “tsandler”`
		
		`git config --global user.mail “tsandler92@gmail.com”`

Hacer **pull** (bajar los cambios)		`git pull`

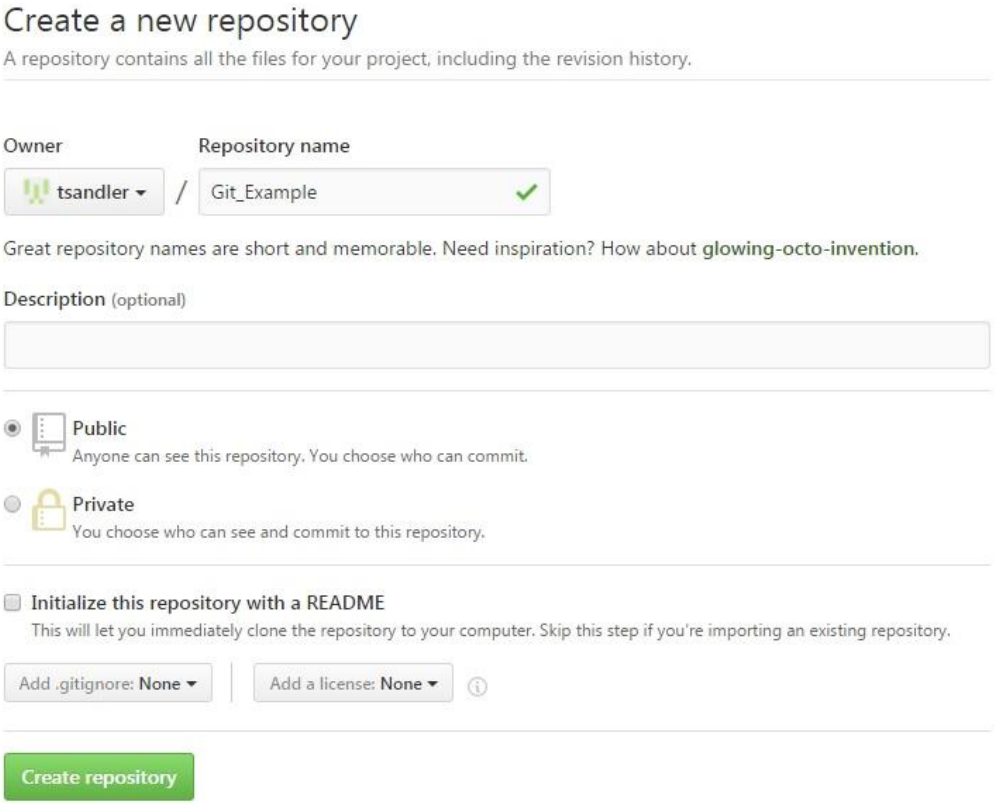
Preparar los cambios para el commit.
(Pasar de Unstaged a Staged = pasar al Index)
		`git add <nombreArchivo>`
		
		`o si no está en el directorio raíz:`
		
		`git add <ruta/nombreArchivo>`

Hacer **commit** (guardar en mi repo **local**)		`git commit -m “descripcion”`
Hacer **push** (subir los cambios commiteados)		`git push`
Hacer **checkout** y eliminar mis cambios para ir al último commit.		`git checkout .`
Conocer el estado actual de las cosas que cambié respecto del repo	Mirando los símbolos También al hacer git commit aparecen los unstaged y staged.	`git status` (aparecen unstaged en rojos y staged/preparados para el commit en verde)

Extra 3: ¿Cómo subo a Github cualquier proyecto mío?

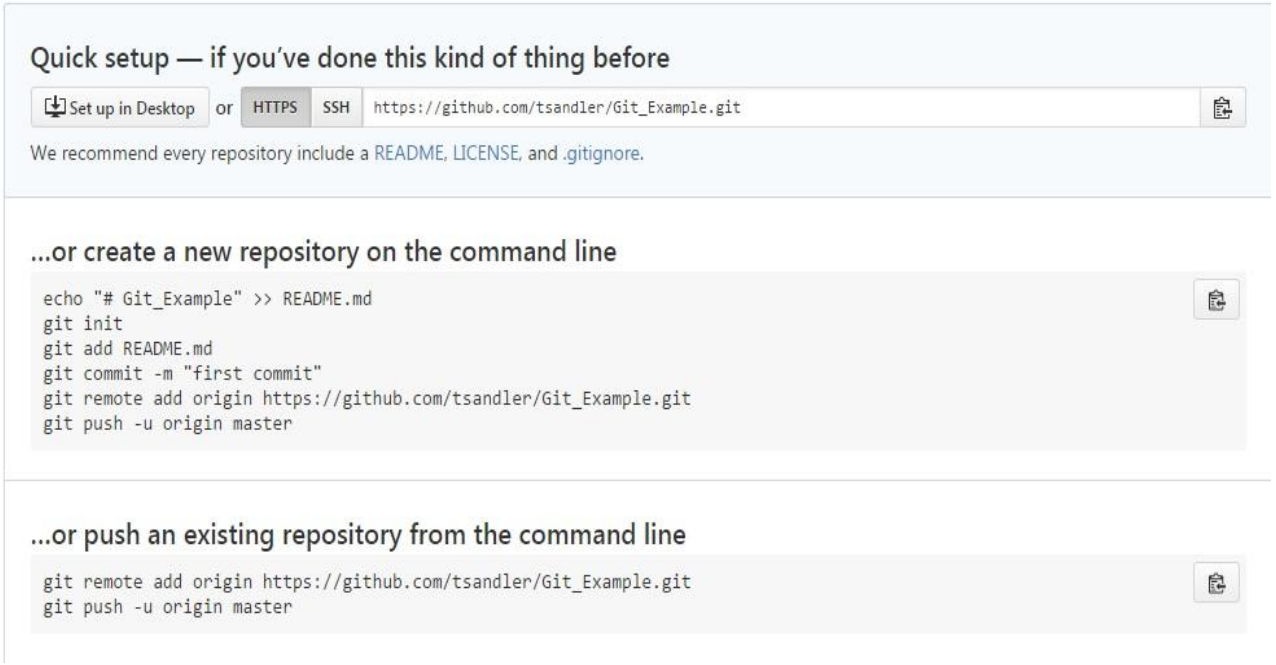
También puede pasar que tengamos un proyecto y querramos subirlo a github, o mismo que necesitemos subir varios archivos al repo desde el inicio, entonces es más fácil crearlo localmente y subirlo en lugar de crearlo remotamente, clonarlo, copiar los archivos, cambiar el directorio dentro del workspace (para poder seguir usándolo desde Eclipse) y subirlo.

- Para esto inicialmente necesitamos crear un repositorio **vacío** en github, para eso hay que ir a la página de github y hacerlo.

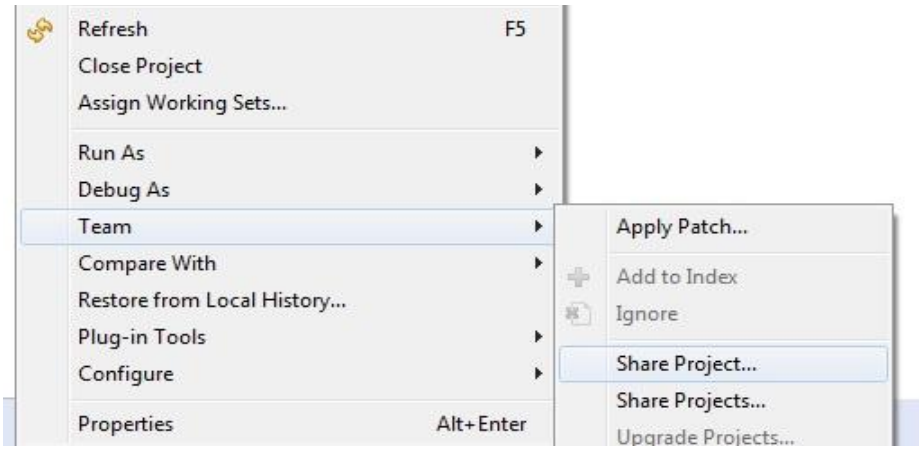


  - Primero indicar el nombre del repositorio y el usuario en el que va a estar.
  - Indicar la privacidad del repositorio (público o privado)
  - Es **importante** dejar **<u>desmarcada</u>** la opción “Initialize this repository with a README”, esto hace que el repositorio se cree vacío.

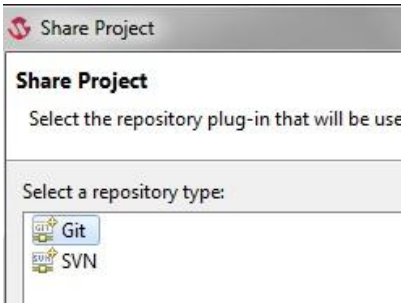
- Al crear el repositorio vacío quedará así, siendo la ruta que aparece en https la que tendremos que usar para subir al repositorio.



- Luego hay que crear un proyecto normalmente y crear un repositorio local para el mismo, para eso hay que hacer click derecho en el proyecto ->Team->Share Project…

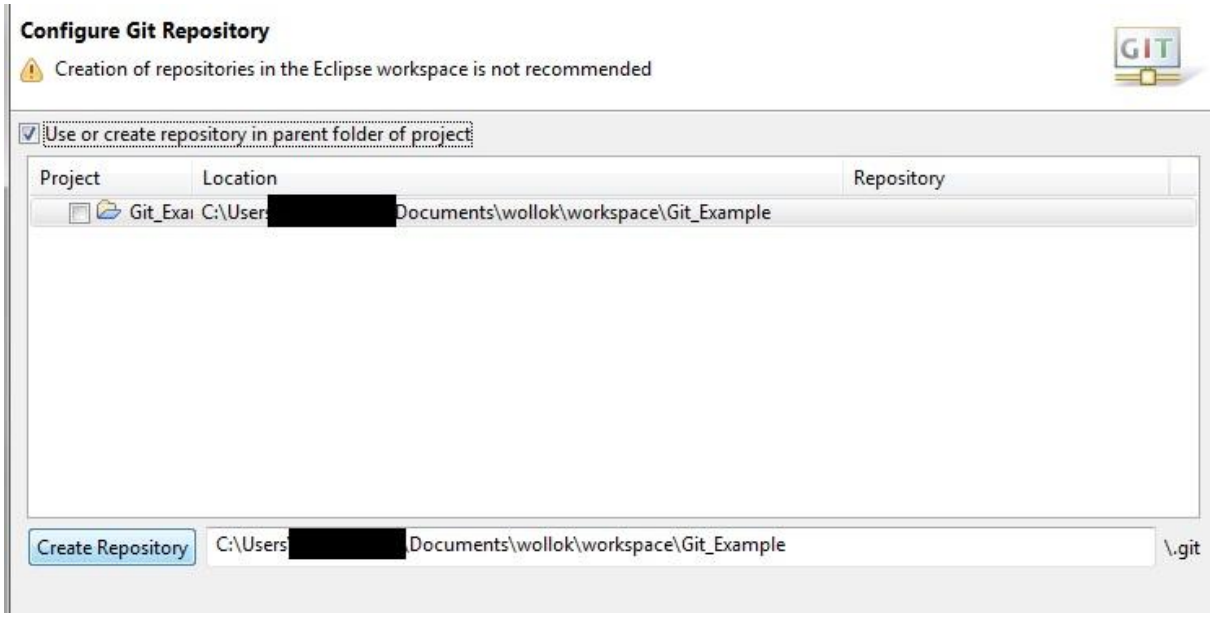


- Seleccionar Git y darle next (este paso no es requerido si solamente tenemos Git instalado, algo que ocurre por defecto desde Wollok 1.6.0)



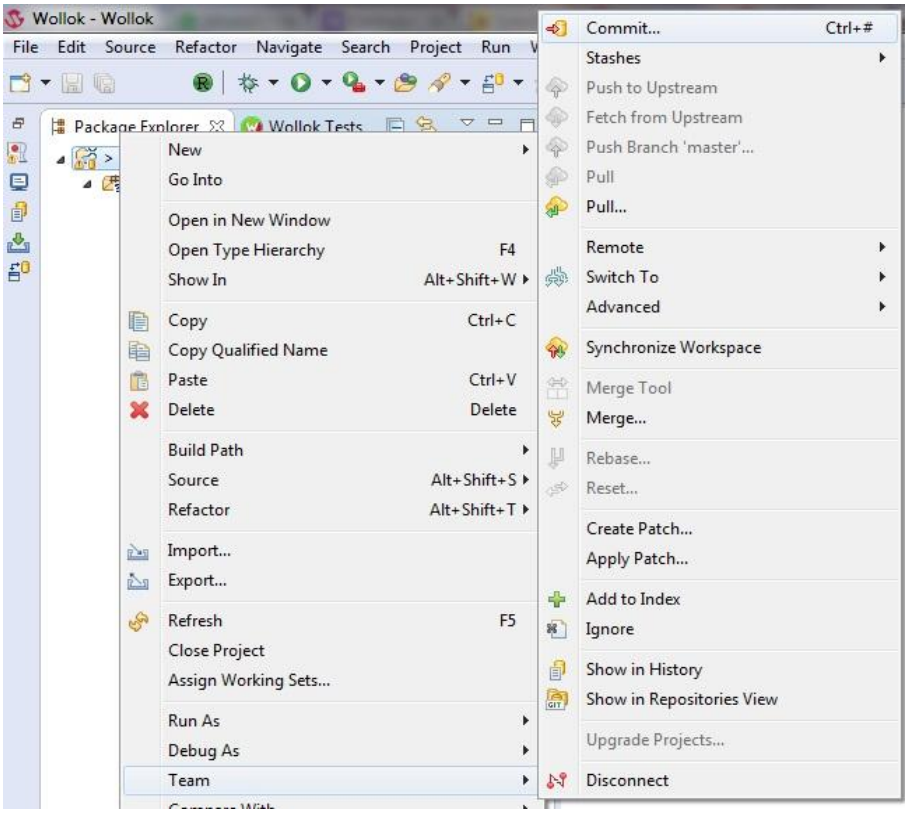
- Tildar el checkbox “Use or create repository in parent folder of project”.

- Seleccionar el proyecto, , ignorar la advertencia, luego el botón “Create repository” y clickear Finish.

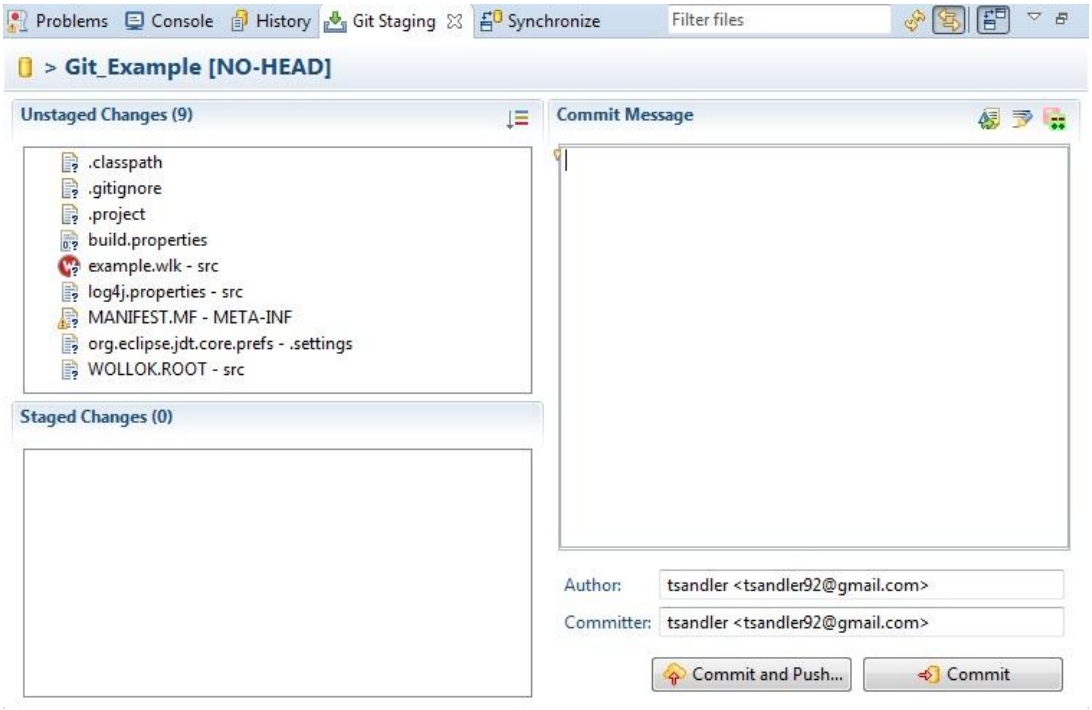


- ¡Listo! Ahora tu proyecto tiene un **repo local**.

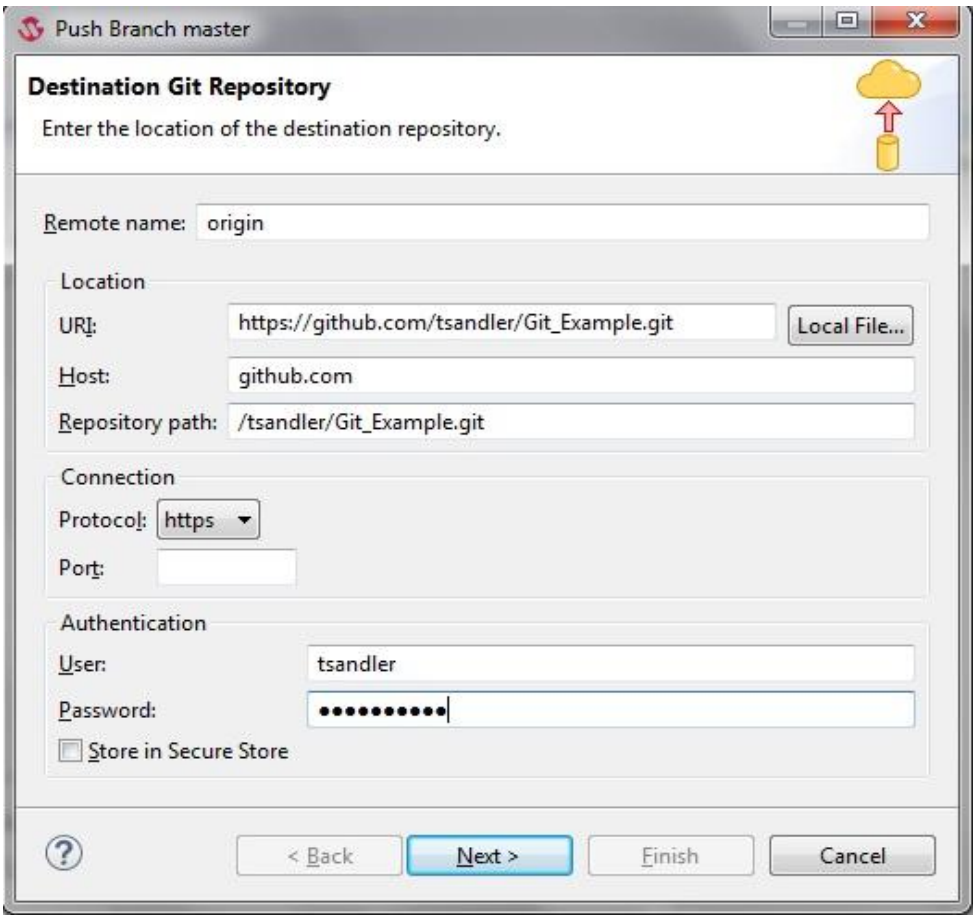
- Para sincronizarlo con un repositorio remoto hay que darle click derecho al proyecto->Team->Commit



- Nos aparecerá esta ventana nuevamente, seleccionamos los archivos que queremos subir, ingresamos un mensaje (en general el primer commit se llama “Initial commit”) y le damos Commit and Push.



- Una vez hecho esto nos aparecerá esta ventana, donde ingresamos la ruta del repositorio, usuario y contraseña y le damos next. Seguimos avanzando, nos logueamos una vez más y listo, el repositorio ya fue inicializado con los archivos del repositorio local que hayamos seleccionado al hacer el commit.



Anexo 1: Shortcuts felices del entorno

Shortcut	Qué hace
Ctrl + 1	Da ayuda contextual para resolver un problema (crear clase, crear variable, etc.)
Ctrl + Shift + L	Muestra la lista de shortcuts activos.
Ctrl + Espacio	Autocompleta al estar escribiendo.
Ctrl + Shift + F	Formatea el código según Window > Preferences: Estilo de código > Formateador **(todavía falta mejorarlo)**
Alt + Shift + R	Renombra una referencia (funciona para métodos, variables y clases, ignorá los warnings que tira)
Alt +Shift + M	Extrae en un nuevo método el código seleccionado.
F2	Renombra un archivo.
Ctrl + S	Guarda el archivo activo.
Ctrl + Shift + S	Guarda todos los archivos.
F3	Ver la definición de una referencia (método, variable o clase)
Ctrl + O	Muestra variables y métodos de un archivo. Un segundo Ctrl + O te muestra variables y métodos heredados también.
Ctrl + Shift + C	Comenta la línea entera.
Ctrl + D	Borra la línea entera.
Ctrl + F11	Ejecuta la acción predeterminada para el archivo activo. Ej: Archivo .wlk abre el repl, archivos .wtest corre los test, etc.
Ctrl + Shift + R	Busca un archivo de wollok y lo abre en el editor.
Ctrl + Shift + F3	Busca un objeto de wollok dentro del espacio de trabajo (útil para buscar las definiciones de los números, fechas, etc.)
Alt + Flecha arriba/ abajo	Mueve el texto seleccionado una línea para arriba / abajo respectivamente.
Ctrl + PageDown	Avanza un tab en los archivos abiertos dentro del Editor.
Ctrl + PageUp	Retrocede un tab en los archivos abiertos dentro del Editor.
Ctrl + L (en REPL)	Limpia el Repl.

Ctrl + M	Maximiza la ventana activa.
Ctrl + Shift +/-	Manejo de zoom de la ventana (Agranda o achica el font).