UTN – FRBA – Paradigmas de Programación
Introducción a la orientación a objetos

Autores:
Carlos Lombardi – car_lombardi@yahoo.com.ar
Nicolás Passerini – npasserini@gmail.com

Índice

Historia del documento

Versión	Fecha	Agrega
1.0	22/04/2005	Versión inicial - Cap 1: Aparecen los objetos / Entes y observadores / Modelo y representación / Objeto y comportamiento / Mensaje y método / Estado interno (y un poco de código) / Encapsulamiento / Conclusiones. - Cap 2: El mismo comportamiento / Polimorfismo
1.2	01/05/2005	Cap 1: Algo más sobre números Cap 2: Interacción y referencias
1.3	20/08/2005	En todos lados: se cambió testGlotón por simulaciónGlotona Cap 1: Se agrega una nota en Mensaje y método. Cap 2: Mensajes a mí mismo – self / Ambiente / Software. Cap 3: Comportamiento común / Cómo se crean instancias / Interactuando con instancias / Refinando la vista de un ambiente de objetos / Forma verdadera del código Smalltalk.
1.4	28/03/2006	Unificación de las distintas acepciones de "comportamiento"

Capítulo 1 – Aparecen los objetos

… si voy a construir software usando orientación a objetos, el primer concepto con el que voy a trabajar, el primero que va a saltar a la vista, es el de *objeto*.

Rápidamente, un objeto es una unidad de software de la cual lo que me importa es: qué le puedo preguntar y/o pedir, y a qué otros objetos conoce. Los objetos responden a los pedidos interactuando con los otros objetos que conoce, y así se va armando una aplicación.

O sea, en vez de enfocarme en rutinas, variables, funciones, punteros, etc.; voy a pensar en objetos: con qué objetos voy a interactuar, sus características, qué les voy a poder preguntar, con qué otros objetos tienen que interactuar estos, y otras cosas que surgen a partir de estas.

De hecho, el resultado de mi trabajo va a ser la descripción de estos objetos, cómo se conocen entre ellos, y cómo interactúan …

Pará man, no entiendo nada. ¿Serías tan gentil de ir un poquitín más despacio?

Entes y observadores

Ahí vamos. Empecemos con la palabra "objeto". Si te digo "objeto", ¿qué se te ocurre?

Objeto, cosa. Una silla, una mesa, un lápiz, una zapatilla, una pelota. También está lo del "objeto del deseo", pero no creo que vaya por ahí …

… estoy de acuerdo, quedémonos con las cosas. Los objetos son exactamente eso, las cosas. Pero fijate que todas las que nombraste son cosas que se tocan y se ven, y son inanimadas; el concepto de "cosa" que le interesa a la orientación a objetos es mucho más amplio. Por ejemplo incluye

- lo que está vivo: personas, animales, plantas, células, pulmones;
- las organizaciones: un equipo de fútbol, un ministerio, un grupo de investigación, un club, una cátedra, una empresa, un ecosistema;
- cosas en lo que se toca es sólo un soporte: un libro, una película, un artículo en una revista, un contrato, una ley;
- cosas bien abstractas: un número, un conjunto, un nombre, una letra, una función matemática.

Todo lo que nombré tiene algo en común: si te pregunto qué es cada una de estas "cosas" vas a poder responder algo, o al menos pensar en posibles respuestas. O sea, para vos cada cosa que nombré es algo, significa algo, tiene alguna entidad.
A este concepto bien amplio de "cosa" lo vamos a llamar "ente".

En una primera aproximación, un objeto es cualquier ente (o sea cualquier "cosa" con el concepto bien amplio que describí recién) que tiene alguna utilidad/significado/sentido *para el que tiene que trabajar con él.*

Esto sí quiere decir que nos va a interesar mucho más lo que un objeto sea "para alguien" que lo que un objeto sea "en sí mismo". Ese "alguien" es quien va a trabajar/interactuar con el objeto; por ahora lo vamos a llamar "observador" y vamos a suponer que los observadores son personas.

Distintos observadores van a ver distintos objetos. Es probable que en la visión de un carpintero que describe su negocio aparezcan sillas y no aves, mientras que en la de un ornitólogo (biólogo especializado en aves) lo más probable es que pase lo contrario.

Fijate que un observador tampoco es alguien en abstracto, es alguien haciendo algo/en un rol; los objetos que va a ver son aquellos con los que necesita interactuar de alguna forma para poder cumplir su rol.

Modelo y representación

… OK, muy linda la clase de filosofía, psicología o lo que sea. Pero yo creía que me ibas a hablar sobre desarrollo de software. ¿Cómo se bajan a código los delirios que estás describiendo?

Buen punto. Una idea fuerte de la orientación a objetos es usar en el desarrollo de software algunos conceptos que vienen de la forma en la que las personas nos relacionamos con el mundo que nos rodea. Por eso el discurso filosófico.

De paso, esta forma de encarar el desarrollo de software no es casual, es bien buscada y te da (o al menos creemos que te da) ventajas respecto de los productos que construís. De eso vamos a hablar un poco más adelante.

Por ahora te pido que te banques un par de minutos más de filosofía; te prometo que vamos a bajar a código antes de lo que estás pensando en este momento, pero tenés que tener un toque más de paciencia. Así que seguimos en nivel delirio.

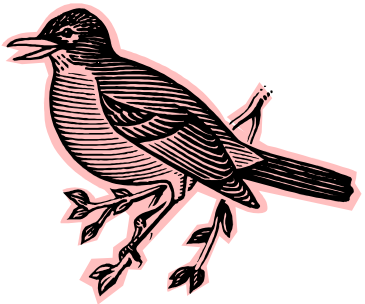
El mundo con todos sus detalles es más complejo de lo que una persona puede entender, entonces cuando tenemos que desempeñarnos en cierto rol necesariamente vamos a enfocarnos en algunos objetos y olvidarnos del resto. Una "persona_en_un_rol" coincide con lo que llamamos "observador" más arriba, de aquí en más quedémonos con "observador".

Un ornitólogo puesto a trabajar va a pensar en aves, plantas, el Sol, sus instrumentos de medición, el calendario, el clima, los ecosistemas; y también en números, relaciones, cálculos. Seguramente no va a pensar en sillas, cucharas de albañil, banderas, clubes de fútbol, números complejos.
(OK, es probable que sí piense en clubes de fútbol, pero no para su trabajo sino como parte del hecho que es una persona, y que es muy difícil que ponga el foco en una cosa sola. Concentrémonos en los objetos que aparecen en el rol específico)

Resumiendo: distintos observadores ven distintos objetos, porque hacen distintos recortes del mundo complejo.

Y esto no es todo: también pasa que distintos observadores interactúan en formas distintas con el mismo objeto, le interesan facetas distintas.

Pongamos como objeto a Pepita, una golondrina tijerita; y analicemos la conducta de distintos observadores.



Doña Pepita

A un ornitólogo le puede interesar p.ej. a qué velocidad vuela, qué come, cómo digiere, cuánto pesa, cuáles son sus patrones de consumo de energía, cuáles son sus ritos de apareamiento; las interacciones (lo que va a hacer el ornitólogo con la golondrina) van a ser pesarla, medirla, analizar su sangre, ponerle comida y esperar que coma, llevar un registro con los momentos en que empieza y para de volar, medir su velocidad, ponerle un anillo de reconocimiento.

A un fotógrafo de la naturaleza le pueden interesar de Pepita cosas como los colores, el brillo, a qué horas aparece, también los ritos de apareamiento; las interacciones van a ser esperarla, sacarle fotos, atraerla.

Un zorro con hambre claramente va a tener otra perspectiva acerca de Pepita: le van a interesar más que nada el peso y la velocidad; y sus interacciones van a ser … bastante primitivas, perseguirla, cazarla y comérsela.

Resumiendo: a distintos observadores les van a interesar distintas características y formas de interacción con los mismos objetos.

Entonces, de todos los objetos con los que podríamos interactuar, nos quedamos con algunos; y de todas las formas de interactuar con cada objeto, nos quedamos con algunas.
Estas operaciones de recorte forman el ***modelo*** que el observador se hace de lo que quiere observar. Armar modelos y trabajar en base a ellos es la forma que encontramos las personas para poder pensar acerca del mundo complejo en el que vivimos (lo que hablábamos al principio).

Hay un paso más, y después ya vamos al software.
¿En qué consiste, dónde está, el modelo que el ornitólogo se hace de Pepita?
En el montón de notas, fotos, cuadros, etc., que tomó a partir de sus observaciones; y *en lo que quedó en su cabeza* que le permite pensar acerca de Pepita.
El modelo es la ***representación*** que el ornitólogo se hizo de Pepita; no es Pepita ni está en ella.

Yo, que soy una persona, digo

> "Pepita vuela moviendo sus alas como buena golondrina que es"

A Pepita no le hacen falta los conceptos de "volar", "ala" ni "golondrina" para hacer eso que yo llamo "volar".

Resumiendo: el modelo está separado del objeto, queda del lado del observador. El observador genera una representación de los objetos con los que interactúa.

Objeto y comportamiento

Y acá es donde entra el software. *Los objetos de la orientación a objetos son las representaciones computacionales de los entes con los que vamos a interactuar.*

Si hago una aplicación para que el ornitólogo registre la info que va obteniendo y genere reportes y otros resultados, sí voy a tener en esa aplicación un objeto que representa a Pepita. Si un usuario quiere saber cuánto voló Pepita entre el 21 y el 25 de febrero, su velocidad standard de vuelo, su peso, cuál fue el vuelo más largo que hizo, etc.; se lo va a pedir al objeto que representa a Pepita. Cuando el ornitólogo quiera registrar un vuelo de Pepita, el resultado de un análisis, el código de su anillo identificatorio; es probable que lo haga pidiéndole a ese objeto que registre la info.

El código que voy a escribir es el que va a implementar el objeto que representa a Pepita, de forma tal de poder registrar todo lo que resulta de las interacciones del ornitólogo, y poder responder correctamente a las consultas que le puedan interesar al mismo ornitólogo o a otro usuario.

Concentrémonos ahora en el primer pedido: cuánto voló Pepita entre el 21 y el 25 de febrero. Supongamos que la respuesta es 32 km.

Para poder formular la pregunta, el usuario tiene que poder representar de alguna forma las dos fechas y/o el período entre ambas. Sí, adivinaste: va a haber objetos que representen las fechas, y objetos que representen períodos de tiempo.

La respuesta también va a estar dada por un objeto, que representa la magnitud "32 km", y que está a su vez compuesto por dos, uno que representa al número 32 y el otro a la unidad de medida kilómetro.

Si le pido el código del anillo identificatorio, me va a responder un código, p.ej. "AZQ48". Si ya tenés claro que hay un objeto que representa al código, y también hay un objeto que representa a cada letra, le estás tomando la onda.

Resumiendo: además de los objetos que representan los entes que tienen sentido para los usuarios (en este caso Pepita), aparecen muchos objetos más que representan conceptos más básicos (fechas, distancias, números, códigos, texto, unidades de medida. *Cualquier* ente que quiera representar, lo tengo que representar mediante un objeto; la orientación a objetos no me da otras formas.

A partir de ahora, en vez de decir "el objeto que representa a X" vamos a decir "el objeto X". P.ej. vamos a hablar del objeto Pepita, los objetos fecha, los objetos número, etc.

Este concepto de objeto sí me está indicando que en un sistema de facturación van a aparecer objetos factura, objetos cliente, objetos deuda, objetos descuento, etc.; y en un sistema para la fabricación de una carpintería van a aparecer objetos silla, objetos garlopa, etc.

Vimos ya que en una aplicación construida según la orientación a objetos, además de los objetos que representan los entes con los que interactúan los usuarios, aparecen muchos otros.

Los objetos con los cuales decida trabajar van a conformar el modelo del cual hablamos en la sección anterior y los vamos a elegir en base al criterio que mencionamos antes: Que tenga sentido interactuar con ellos de alguna forma.

Claramente va a existir un objeto Pepita; es aquel con el que va a interactuar el ornitólogo en las formas que describimos más arriba. A una fecha le puedo pedir el mes, qué día de la semana es, si es o no anterior a otra fecha, y otras cosas; entonces es probable que aparezcan objetos fecha. Lo mismo con una distancia (le puedo pedir que se sume a otra distancia, que me diga a cuántos metros equivale), un número (le puedo pedir que se multiplique con otro número), un texto (le puedo pedir la cantidad de letras, si aparece o no cierta letra).

A las interacciones vistas como una unidad las vamos a llamar ***comportamiento***; es la palabra que se usa en la literatura sobre orientación a objetos.

Ahora sí podemos armar una definición de objeto con bastante precisión:

objeto: representación computacional de cualquier ente que exhiba comportamiento

Si me creo las ideas de la orientación a objetos, en el software que construyo solamente hay objetos.

Mensaje y método

¿Por qué la definición de objeto dice "exhibe" y no "tiene"?
Porque lo que interesa es que lo que el objeto exhibe, lo que un observador puede usar.

¿Y cómo hace un observador para usar un objeto?

Le envía ***mensajes***. Un mensaje es cada una de las formas posibles de interactuar con un objeto. Para pedirle algo a un objeto, lo que hago es enviarle un mensaje. Cada objeto entiende un conjunto acotado de mensajes. P.ej. el objeto Pepita entiende el mensaje "decime cuánta energía tenés" pero no el mensaje "¿cuál es tu mes?", y viceversa para un objeto fecha.

Sí es posible (y altamente deseable, como veremos en un ratito) que haya muchos objetos capaces de entender el mismo mensaje, incluso aunque sean bastante distintos entre sí. Eso permite ir armando un vocabulario mediante el cual uno interactúa con los objetos.

En cada acción de envío de mensaje:

- hay un emisor
- hay un receptor
- hay un nombre, que identifica el mensaje que estoy enviando entre todos los que entiende el receptor.
  Como estamos hablando de software, un nombre toma la forma de un identificador. P.ej. el objeto Pepita podría entender los mensajes `energía`, `velocidad`, `distanciaRecorrida`.
  En la jerga de Smalltalk, a este nombre se lo llama ***selector***.
- puede haber parámetros. P.ej. si le quiero indicar al objeto Pepita que Pepita (la golondrina de verdad) comió 30 gramos de alpiste (lo que el objeto Pepita necesita saber porque eso aumenta su energía), el "30" (o el "30 gramos") va a ser un parámetro. Si no, el objeto Pepita debería entender un mensaje "comiste 30 gramos de alpiste", otro distinto "comiste 50 gramos de alpiste", y así hasta el infinito
- puede haber un resultado, que es la respuesta que obtiene el emisor. P.ej. si le pido al objeto Pepita la energía, un resultado posible es "48 joules".

Está claro que tanto los parámetros com el resultado son objetos, en los ejemplos de recién el parámetro es *el objeto* 30 (o "30 gramos") y el resultado es *el objeto* 48 (o "48 joules").
Los parámetros y resultados no son necesariamente objetos básicos (números, Strings, etc.). Si el objeto Pepita entendiera el mensaje `padre`, el resultado sería otro objeto golondrina.

Pasamos en limpio las definiciones prolijas de mensaje y envío de mensaje[1]

mensaje: cada una de las formas posibles de interactuar con un objeto, que se identifica por un nombre y puede llevar parámetros.

envío de mensaje: cada interacción con un objeto. Tiene: emisor, receptor, selector, eventualmente parámetros (que son objetos), y eventualmente un resultado (que es otro objeto).

Hasta acá queda claro que si el ornitólogo quiere saber cuánta energía tiene Pepita en un determinado momento, lo que tiene que hacer es enviarle el mensaje `energía` al objeto `pepita`. Bajemos un poco a detalle.

El ornitólogo trabaja en un entorno Smalltalk, que podría ser un Dolphin[2] como el que vamos a usar en la materia.

En el entorno en el que está trabajando, el ornitólogo tuvo que haber configurado al objeto Pepita.
Para poder interactuar directamente con un objeto, se le pone un nombre dentro del entorno en el que estoy trabajando; p.ej. `pepita`. En Smalltalk los nombres de los objetos empiezan en minúscula, salvo excepciones que veremos más adelante.

La configuración va a incluir pasarle a `pepita` las observaciones relevantes; si lo que le interesa es ir viendo la evolución de la energía, le va a pasar los registros de los eventos que la afectan, p.ej. las veces que comió (que aumentan su energía) y las veces que voló (que la disminuyen).

Después de esto, sólo queda enviarle a `pepita` el mensaje `energía`. En código, esto se hace así

```
pepita energía
```

En el Dolphin podemos (y lo vamos a hacer): abrir un entorno de trabajo, configurar ahí a `pepita`, después escribir lo que dice arriba, pintarlo y pedirle "mostrame el resultado de esto", y anda.

La sintaxis de Smalltalk para enviarle un mensaje a un objeto es

```
objeto mensaje
```

donde "objeto" es el nombre que le di al objeto en el entorno en donde estoy escribiendo el código. Así se entiende el ejemplo de arriba; `pepita` es el (nombre del) objeto, `energía` es el (selector del) mensaje.

Y va exactamente así, sin puntos ni paréntesis ni nada.

[1] en otros ámbitos se llama "mensaje" a lo que nosotros llamamos "envío de mensajes". Hasta donde sabemos, no existe una definición unánimemente aceptada en la literatura sobre orientación a objetos. Elegimos separar los dos conceptos para poder pensar por separado en qué mensajes puede recibir un objeto, y qué mensaje está recibiendo en un momento determinado

[2] O bien el entorno Squeak que son los dos smalltalks que pueden usar en la cátedra

Volviendo: cuando el ornitólogo envía el mensaje `pepita energía` … ¿qué pasa? Obviamente, pasa que se ejecuta código.

¿Qué código? Ahí vamos. Para que el ornitólogo pueda interactuar con `pepita`, alguien la tuvo que programar; donde "programar" quiere decir: indicar qué mensajes va a entender `pepita`, y para cada uno escribir el código que se va a ejecutar cuando `pepita` lo reciba.

O sea: fue un programador el que decidió que `pepita` va a entender el mensaje `energía`, y escribió el código asociado a ese mensaje. Ese código es el que se va a ejecutar cuando el ornitólogo ejecute `pepita energía`.

La sección de código que se asocia a un mensaje se llama ***método***. Un método tiene un nombre, que es el del mensaje correspondiente; y un cuerpo, que es el código que se ejecuta.
Un detalle: en la jerga de objetos, se usa mucho el término "evaluar" en vez de "ejecutar".

Con todo esto, pasamos la definición en limpio junto con un corolario

método: sección de código que se evalúa cuando un objeto recibe un mensaje. Se asocia al mensaje mediante su nombre.

Corolario: casi todo el código que escribamos va a estar en métodos, que definen qué mensajes entiende cada objeto, y qué pasa cuando los recibe.

Lo que **no** vimos es dónde se escriben los métodos. Para eso falta un rato.

Antes de pasar a otro tema; arriba dijimos "el ornitólogo tuvo que haber configurado al objeto Pepita". ¿Cómo hizo? Obviamente enviándole los mensajes que le dicen a `pepita` lo que le pasó a Pepita. Dicho en general, los que le indican a la representación computacional lo que le pasó al objeto representado.

Insistimos, la única forma que tengo de interactuar con un objeto es enviándole mensajes.
P.ej. para indicarle que Pepita comió 30 gramos de alpiste una forma posible es

```
pepita comióAlpiste: 30
```

Los nombres de mensaje con un parámetro terminan con dos puntos, eso indica que atrás viene un parámetro; los dos puntos forman parte del nombre. La sintaxis de envío es

```
objeto mensaje parametro
```

Acá estamos suponiendo que `pepita` entiende el mensaje `comióAlpiste:`, y que el parámetro es un número que expresa la cantidad en gramos.

En el apunte vamos a ir introduciendo más elementos de la sintaxis de Smalltalk.

Estado interno (y un poco de código)

Vamos a programar una versión muy simplificada de `pepita`. Los que queremos que `pepita` haga es:

- lo único que quiero es poder saber en cada momento cuánta energía tiene pepita,
- los únicos eventos relevantes son comer alpiste y volar,
- el alpiste le brinda 4 joules por cada gramo que come,
- en cada vuelo consume 10 joules de "costo fijo" (es lo que le cuesta arrancar), más un joule por cada minuto que vuela,

Para hacerla fácil vamos a representar los joules, gramos y minutos con números, p.ej. la cantidad "30 joules" la vamos a representar con el número 30.

Los mensajes que va a entender `pepita` son:

- `energía`, que devuelve la energía expresada en joules
- `comióAlpiste:`, que registra la ingestión de alpiste; el parámetro es la cantidad expresada en gramos
- `voló:`, que registra un vuelo; el parámetro es la duración expresada en minutos (no vamos a tener en cuenta los errores, como que la energía quede negativa después de volar)

Para implementar este comportamiento, hagamos que pepita recalcule la energía después de cada acción; cuando le preguntan la energía, devuelve la que tiene calculada.

¿Dónde guardo la energía calculada? Ahí vamos. A cada objeto le puedo asociar un conjunto de variables, cuyo valor vive entre envíos de mensajes al mismo objeto.

El conjunto de variables que va a contener un objeto se llama ***estado interno***.

Una "variable" es el nombre de un objeto, entonces lo que me estoy guardando en realidad es qué objeto está nombrando cada variable. Esto va a ir quedando claro a medida que avancemos. Por ahora alcanza que quede claro que una variable **no** es un nombre de una sección de memoria.

Resumiendo:

estado interno de un objeto: conjunto de variables que contiene. Se define al programar el objeto.

Volviendo, con el planteo que hicimos alcanza con que pepita tenga una sola variable, llamémosla energíaCalculada. El código para `pepita` es como sigue, indentando para marcar las distintas partes

```
pepita
    variables
        energíaCalculada

    métodos
        comióAlpiste: gramos
            "Le indican a pepita que comió alpiste;
             el parámetro es la cantidad expresada en gramos"
            energíaCalculada := energíaCalculada + (gramos * 4)
```

```
voló: minutos
    "Le indican a pepita que voló;
     el parámetro es la duración del vuelo expresada en minutos"
    energíaCalculada := energíaCalculada – (10 + minutos)

energía
    "Devuelve la energía del receptor medida en joules"
    ^energíaCalculada


reset
    "Vuelve pepita a un estado inicial"
    energíaCalculada := 0
```

La sintaxis es la de Smalltalk; resaltamos algunos elementos

- la asignación es con :=
- los comentarios van entre comillas dobles
- para indicar el resultado de un método se pone ^resultado (el circunflejo es análogo al "return" en otros lenguajes).

Agregamos `reset` al comportamiento de `pepita` para poder hacer muchas pruebas empezando siempre de 0.

Una observación interesante: el selector para pedirle la energía a `pepita` es `energía`, no `dameTuEnergía`, `getEnergía` u otras variantes. Es la convención Smalltalk, los selectores que piden info tienen el nombre de lo que piden sin ningún agregado; en principio conviene adherir a las convenciones del lenguaje/entorno que usamos. Además, se escribe menos y el código queda más legible.

Ahora usemos a pepita. En Dolphin, esto se puede hacer creando un entorno en el cual se interactúa con el objeto (obviamente, enviándole mensajes). Ese entorno es una ventana de texto; el texto se escribe, se pinta y se evalúa; si hay un resultado, se muestra en la misma ventana. Los envíos de mensajes se separan con un punto decimal, como si cada uno fuera una oración.

Supongamos que la simulación que le interesa al ornitólogo es esta secuencia: comer 50 gramos de alpiste, volar 5 minutos, comer otros 30 gramos, volar 10 minutos. En la ventana de texto escribimos:

```
pepita reset.
pepita comióAlpiste: 50.
pepita voló: 5.
pepita comióAlpiste: 30.
pepita voló: 10.
```

Pintamos y ponemos "evaluar", y listo.

Hasta acá no obtenemos ningún resultado interesante, la evolución quedó en el estado interno de `pepita`, a la que ahora le podemos pedir la energía

```
pepita energía
```

pintamos solamente esta línea, y ponemos "evaluar". Ahora sí me va a dar un resultado interesante, que es 285 como esperábamos.

Ahora cansemos un poco más a pepita, evaluando

```
pepita voló: 30
```

Volvemos a pedir la energía y ¡sí! obtenemos 245 como corresponde. Tenemos a pepita full operativa.

Si quiero empezar de 0, le envío reset y después lo que quiero. Acá queda claro el uso del reset, poder usar el mismo objeto para muchas pruebas.

Una característica interesante de muchos entornos Smalltalk es que puedo hacer que los objetos que creo queden "vivos" en un entorno que a su vez vive dentro del Smalltalk. Puedo crear objetos y después interactuar con ellos cuando quiero.
P.ej. yo pude, dentro de un entorno, haber configurado a `pepita` ayer con la data de una prueba y cerrado el Dolphin; hoy lo abro de nuevo, va a volver a aparecer ese entorno y ahí puedo pedirle la energía y/o hacer cualquier interacción con `pepita`, que está como la dejé ayer.

Un detalle final… en un entorno Smalltalk las cosas son un poco distintas a como las contamos en esta sección. En realidad, el código no se asocia al objeto, y el objeto no se crea no al escribir código sino en el entorno de uso. Esto lo vamos a ver en el capítulo 3, recién ahí vamos a poder hacer funcionar este ejemplo usando las herramientas standard del Dolphin.

Tanto el código en sí como la ventana de interacción sí "son así" en Dolphin, ahí no mentimos nada.

Encapsulamiento

Un programador un poco más rebuscado podría haber implementado pepita con un estado de dos variables: una para la energía ingerida, y otra distinta para la energía consumida. Quedaría así:

```
pepita

    variables
        energíaIngerida
        energíaConsumida

    métodos
        comióAlpiste: gramos
            "Le indican a pepita que comió alpiste;
             el parámetro es la cantidad expresada en gramos"
            energíaIngerida := energíaIngerida + (gramos * 4)


        voló: minutos
            "Le indican a pepita que voló;
             el parámetro es la duración del vuelo expresada en minutos"
```

```
    energíaConsumida := energíaConsumida – (10 + minutos)


energía
    "Devuelve la energía del receptor medida en joules"
    ^energíaIngerida – energíaConsumida


reset
    "Vuelve pepita a un estado inicial"
    energíaIngerida := 0.
    energíaConsumida := 0
```

Esta implementación es bastante distinta a la anterior.

Ahora viene el ornitólogo y quiere hacer las mismas pruebas de la sección anterior, usando la nueva implementación de pepita.

Repasamos la parte de uso de pepita, ¿qué impacto tuvo el cambio de implementación en la forma de usar a pepita? Claramente **ninguno**, todos los mensajes que el ornitólogo le enviaba a la "vieja pepita" se los puede enviar a la "nueva pepita", obteniendo los mismos resultados.

Digámoslo de nuevo en términos más generales.
Cambiamos bastante la implementación de un objeto, y el usuario de ese objeto no se enteró.
¿Cómo logramos esto? Porque no cambiamos el protocolo del objeto, o sea el conjunto de mensajes que entiende; y lo único que ve el usuario del objeto son los mensajes que entiende, no ve cómo el objeto lo implementa "adentro suyo" (métodos y estado interno).

Trabajando en Smalltalk, el ornitólogo no tiene forma[3] de conocer la forma del estado interno de pepita, ni sus valores; y tampoco puede acceder a detalles sobre cuál es la implementación que está detrás de los mensajes que le envía. Lo único que conoce es el comportamiento que `pepita` exhibe; es lo único que necesita conocer para poder interactuar con el objeto.
Esta idea de que un observador no ve todos los aspectos de un objeto sino solamente aquellos que le le sirven para interactuar con él se llama ***encapsulamiento***; la idea del nombre es que los aspectos internos del objeto se *encapsulan* de forma tal que los observadores no pueden verlos.
La idea de encapsulamiento también puede observarse en la relación que nosotros tenemos con el mundo que nos rodea: para interactuar con una videocasettera me alcanza con entender el control remoto, no necesito (¡afortunadamente!) saber p.ej. por dónde pasan las correas que llevan el movimiento del motor a los cabezales; para darle de comer a una golondrina no necesito la composición química de sus jugos gástricos, y miles de etc.

Al encapsular un objeto, estoy al mismo tiempo acotando y explicitando (haciendo explícitas) las formas posibles de interacción; sólo se puede interactuar con un objeto mediante el comportamiento que exhibe. Esto nos da varias ventajas que pasamos a comentar.

Como las formas de interacción son acotadas y las maneja quien programa el objeto, se hace más sencillo probar si un objeto se comporta correctamente.

[3] En realidad sí tiene, pero usando características bien "hackerosas" del Smalltalk, que se usan para ciertas tareas de bajo nivel y que un desarrollador (¡o usuario!) de `pepita` ni se le va a ocurrir usar en este contexto.

En el ejemplo, al programar a pepita sabemos que las únicas formas de interactuar con ella son los cuatro mensajes que definimos; por lo tanto, para probar si pepita anda bien o no alcanza con probar secuencias de envíos de mensajes y ver que en todos los casos se comporta correctamente, que en el caso de pepita es básicamente pedirle la energía y que devuelva el valor esperado.

También resulta más controlable cualquier cambio que haga en la implementación, siempre que respete el conjunto de mensajes definido; eso es lo que vimos con el ejemplo de cambio de una variable a dos variables.
En una aplicación sencilla como la que calcula la energía de pepita eso puede aparecer como poco valioso; a medida que va creciendo el tamaño de las aplicaciones que construimos, la necesidad de cambios de implementación se va haciendo más fuerte; se puede dar por necesidad de mejorar la performance y/o uso de recursos, de ampliar el comportamiento de los objetos con los que trabajamos (p.ej. con el cambio de implementación podemos informar la energía consumida, con la implementación anterior no podíamos hacer esto), etc..
A su vez, como es más sencillo probar si un objeto "funciona bien" o no, se simplifica darnos cuenta de si con un cambio de implementación "rompemos algo".

Otro efecto es que al explicitar el comportamiento lo estamos documentando, entonces para alguien que quiera usar el objeto que construimos es fácil saber qué usos le puede dar y cómo usarlo. De alguna forma el comportamiento que exhibe el objeto forma un "manual de uso".
Eso se potencia si seguimos la sabia política de poner abajo del nombre de cada método un comentario que explica qué se puede esperar, como lo hicimos en los ejemplos de código. Es una convención de Smalltalk que definitivamente conviene respetar en el código que escribamos nosotros.

Lo que hablamos sobre encapsulamiento explica por qué tenemos que definir un mensaje energía aunque tengamos la variable energíaContenida. En Smalltalk no hay forma de que un usuario del objeto acceda a la variable, entonces es necesario. Además, el definir el mensaje permite hacer cambios de implementación como el que vimo, entonces es conveniente.

Resumiendo

encapsulamiento: quien usa un objeto sólo ve lo que necesita para poder interactuar con él, que el comportamiento que exhibe. Los detalles internos quedan encapsulados en el objeto.
Así quedan acotadas y explicitadas las formas posibles de interactuar con un objeto.

Algo más sobre números

Dijimos que los números se representan mediante objetos, porque cualquier cosa que quiera representar la voy a representar mediante números.
A partir de esto, entendamos un poco mejor la expresión

```
2 + 3
```

Para empezar, ¿qué son `2` y `3`? Son nombres de objetos, al igual que `pepita`[4]. Los nombres `2` y `3` "ya vienen con el Smalltalk" mientras que `pepita` no, pero conceptualmente es la única diferencia.

[4] En el capítulo 3 veremos que los nombres pepita y 0 son en realidad distintos a lo que describimos en este capítulo; en ese sentido los capítulos 1 y 2 son una aproximación y la versión precisa aparece recién en el capítulo 3.

Entonces esta expresión ¿es un envío de mensaje? Exactamente, con la sintaxis que vimos

```
objeto mensaje parámetro
```

El objeto (receptor) es el 2, el mensaje (selector) es el +, el parámetro es el 3.

Esto ¿quiere decir que en algún lado hay un método +? Sí, y sí se lo puede encontrar en el Dolphin; pero para buscarlo tenemos que ver un par de cosas más.

¿Pero cómo, si es un mensaje que lleva parámetro el selector no debería terminar con :? Buen punto. Smalltalk asume que los selectores con ciertos caracteres (+ - * / y algunos más) llevan parámetro aunque no lleven un : detrás.

Pero lo importante es lo otro: que los números son objetos conceptualmente iguales a pepita (o a los objetos que representan fechas, o a cualquier otro objeto que vaya a aparecer), y que lo que vemos como "operaciones matemáticas" son envíos de mensaje como cualquier otro.

Conclusiones

Mencionamos rápidamente algunas conclusiones que conviene tener claras para leer lo que sigue. Esto **no** es un resumen, en el capítulo se dicen más cosas que en esta sección: en particular fíjense que no copiamos ninguna definición.

1. todo lo que quiera representar, lo voy a representar mediante objetos: objetos que representan golondrinas, objetos que representan números, objetos que representan fechas, etc.. A partir de ahora lo diremos más corto: objetos golondrina, objetos número, objetos fecha.
2. los entes que voy a representar, o (lo que es lo mismo) los objetos que van a aparecer, van a ser aquellos con los cuales necesitaremos interactuar, o yo o los otros objetos que defino.
3. la forma de interactuar con un objeto es enviándole mensajes, y analizando las respuestas que obtengo, que son otros objetos. Podemos pensar en que el observador habla con los objetos, y los mensajes que entiende el objeto forman el lenguaje de esa conversación.
4. programar es: definir el comportamiento de los objetos que debo modelar, y escribir el código que haga falta para que cada objeto exhiba el comportamiento definido.
5. lo único que ve un usuario (observador u otro objeto) de un objeto es su comportamiento. Hay aspectos de un objeto que quedan "adentro" del objeto. De esta forma se precisa la definición de cuáles son las interacciones posibles con un objeto.

Capítulo 2 – Relaciones entre objetos: polimorfismo y referencias

Está claro que en cualquier aplicación van a aparecer muchos objetos. Esto ya nos pasó con el ejemplo del capítulo anterior, en donde además del objeto pepita aparecieron objetos número.

Al construir software usando orientación a objetos, el asunto de cómo se relacionan los objetos es muchas veces tanto o más importante que cómo implementar cada uno.

En este capítulo vamos a poner el foco en conceptos que tienen que ver más con formas en las que los objetos se relacionan que con características de un objeto visto aisladamente. Para esto vamos a extender el ejemplo del capítulo anterior.

Los mismos mensajes

En principio, además de pepita va a aparecer un picaflor llamado pepe. Al ornitólogo le interesa hacer con pepe lo mismo que con pepita: estudiar la evolución de su cantidad de energía a medida que come y que vuela. Obviamente, pepe tiene distintos patrones de consumo de energía:

- el alpiste le brinda 16 joules por gramo que come, pero cada vez que come hace un esfuerzo fijo que le demanda 45 joules.
- en cada vuelo consume 30 joules, que es lo que le cuesta arrancar y parar. Es tan liviano, que el costo de mantenerse en vuelo y de moverse es despreciable.

¿Qué mensajes queremos que entienda pepe? Una respuesta posible es

> Voy a hacer que pepe entienda los mismos mensajes que pepita, porque las formas de interacción del ornitólogo con los dos van a ser las mismas. El ornitólogo le va a enviar los mismos mensajes a los dos, lo que puede cambiar son las respuestas de cada uno.

Esta respuesta, esta forma de pensarlo, está perfecta. Si tu intuición te dijo esto (o algo así), vas por un camino muy bueno. Hay dos aspectos que destacamos:

- al pensar en un objeto, lo pienso desde las formas en que el observador va a querer interactuar con él. Dicho de otra forma, al configurar un objeto lo pienso desde los usuarios que va a tener ese objeto.
- si los patrones de interacción con dos objetos son los mismos, lo lógico es que tengan un vocabulario en común. Yendo un poco más allá en esta idea, si hago esto le va a ser más fácil al observador hablar con uno o con el otro indistintamente.

En la visión de este apunte, tener presente estos dos aspectos es muy importante y valioso para aprovechar las fortalezas de la orientación a objetos, y de paso divertirse en el proceso (bueno, o al revés).

Obsérvese que dijimos ***usuario***: los usuarios de un objeto incluyen los observadores que interactúan directamente con ellos, y también los otros objetos que les envían mensajes.
Algunas cosas que dijimos en el capítulo 1 para observadores son válidas para cualquier usuario de un objeto, en particular el envío de mensajes y que lo único que conoce son los mensajes que entiende.

Polimorfismo

OK, vamos despacito. Empecemos por codificar a pepe:

```
pepe

    variables
        energíaCalculada

    métodos
        comióAlpiste: gramos
            "Le indican a pepe que comió alpiste;
             el parámetro es la cantidad expresada en gramos"
            energíaCalculada := energíaCalculada + (gramos * 16) – 45


        voló: minutos
            "Le indican a pepe que voló;
             el parámetro es la duración del vuelo expresada en minutos"
            energíaCalculada := energíaCalculada – 30


        energía
            "Devuelve la energía del receptor medida en joules"
            ^energíaCalculada


        reset
            "Vuelve pepe a un estado inicial"
            energíaCalculada := 0.
```

Retomemos la simulación que definió el ornitólogo, que ya usamos en el capítulo 1

```
pepita reset.
pepita comióAlpiste: 50.
pepita voló: 5.
pepita comióAlpiste: 30.
pepita voló: 10.
pepita energía
```

Si quiero ejecutar la misma simulación con pepe en lugar de con pepita ¿qué tengo que cambiar en el código? Solamente el objeto al que le envío los mensajes. Al observador le es felizmente indistinto hablar con uno o con el otro, no necesita ninguna adaptación.

Ahora demos un paso más. El ornitólogo tiene preparados distintas simulaciones, que evalúan lo que pasaría con la energía de un ave en distintos escenarios; tiene datos de varias aves y quiere aplicarles los distintos escenarios a cada una y ver con cuánta energía quedan.

A esta altura está clarísimo que vamos a representar cada ave con la que quiere trabajar el ornitólogo mediante un objeto, y que todos estos objetos van a entender los mensajes `reset`, `comióAlpiste:`, `voló:` y `energía`.

Observemos que además de trabajar con muchas aves, el ornitólogo quiere tener a mano muchas simulaciones. Entonces, de repente tiene onda representar simulaciones así como representamos aves. Una vez decidido esto, debería quedarnos claro que van a aparecer objetos simulación, porque (otra vez) cualquier ente que quiera representar, lo voy a representar mediante un objeto.

¿Cómo va a interactuar el ornitólogo con un objeto simulación? Pasándole el (objeto) ave con el que quiere que trabaje, y después diciéndole sencillamente algo así como … "dale masa".
Ergo, los objetos simulación van a entender dos mensajes, a los que vamos a llamar `trabajáCon:` y `daleMasa`.

Empecemos armando una simulación en el cual los pájaros comen mucho, al que vamos a llamar simulaciónGlotona.
Pensemos un poco en cómo implementarlo.
En el método daleMasa vamos a indicarle al ave con la que estamos trabajando la secuencia de eventos propios de la simulación, y a devolver la energía que tiene el ave después de estos eventos.
La simulación necesita acordarse del ave que le pasaron para trabajar, esto lo va a hacer en el método trabajáCon:.

Con esto, es fácil armar el código de la simulaciónGlotona

```
simulaciónGlotona

    variables
        ave

    métodos
        trabajáCon: unAve
            "Le ordenan trabajar con un ave determinada"
            ave := unAve

        daleMasa
            "Le ordenan ejecutar la simulacion"
            ave reset.
            ave comióAlpiste: 50.
            ave voló: 5.
            ave comióAlpiste: 30.
            ave voló: 10.
            ^ave energía.
```

Ahora al ornitólogo le resulta mucho más fácil evaluar la simulación para un ave, p.ej. pepita. El código en la ventana de interacción queda así:

```
simulaciónGlotona trabajáCon: pepita.
simulaciónGlotona daleMasa.
```

Para usarlo con pepe, el cambio es trivial:

```
simulaciónGlotona trabajáCon: pepe.
simulaciónGlotona daleMasa.
```

Nota antes de seguir: observamos que el parámetro que recibe trabajáCon: es un ave. Queremos destacar esto para mostrar que cualquier objeto puede ser parámetro, no solamente los números u otros objetos "simples".

Volviendo, ¿tuvimos que cambiar el código de simulaciónGlotona para que trabaje con pepe o con pepita?
No, para la simulación trabajar con pepe o con pepita es exactamente lo mismo. De hecho, la idea es que para la simulación sea lo mismo trabajar con cualquier ave que yo quiera representar, eso tiene mucha onda porque puedo usar la misma simulación para todas las aves.

Obviamente para que esto pase, cada ave que represente tiene que entender los mensajes `reset`, `comióAlpiste:`, `voló:` y `energía`; porque si no en el método `daleMasa` le voy a estar enviando al ave un mensaje que no entiende.
De hecho, la simulación podrá interactuar con cualquier objeto que entienda estos cuatro mensajes, el que sea un ave o no a la simulación no le importa nada.

Lo que termina pasando es que la simulación puede interactuar con muchos objetos distintos, de los cuales lo único que conoce es cómo de interactuar con ellos, y sin tener que hacer ninguna modificación en la simulación para que trabaje con uno o con el otro.

A esta característica de poder trabajar con objetos distintos en forma transparente la llamamos ***polimorfismo***; decimos que pepita y pepe son ***objetos polimórficos***.
Ahora, ¿a quién le sirve que pepita y pepe sean polimórficos? ¿A pepita? ¿A pepe? ¡Claro que no! Le sirve al usuario de pepita y de pepe, que en este caso es la la simulaciónGlotona.

Hay muchas formas distintas de definir polimorfismo, a nosotros nos gusta esta:

> ***polimorfismo***: decimos que dos objetos son polimórficos para un tercero, si éste puede trabajar indistintamente con cualquiera de ellos.[5]

Otra forma de decirlo es que puedo intercambiarlos en forma transparente; el usuario no tiene que adaptarse para trabajar con uno o con el otro.

Una forma distinta de definir el polimorfismo es decir que dos objetos son polimórficos si entienden los mismos mensajes.

En Smalltalk sí es cierto que alcanza con que dos objetos entiendan los mismos mensajes para poder usarlos polimórficamente, pero en otros lenguajes no y más adelante veremos por qué.[6]
Esta es una de las razones por las que preferimos la definición que dimos más arriba.

Hay una frase típica: "El polimorfismo es una de las características más importantes de la orientación a objetos". OK, muy linda la frase, ahora vamos a ver un poco qué queremos decir con esto. El que los objetos que uso sean polimórficos me da:

[5] Decimos "son polimórficos para un tercero" porque eventualmente pueden aparecer distintos usuarios para un mismo objeto, que le envíen distintos mensajes; entonces podría pasar que dos objetos sean polimórficos para un usuario pero no para otro. De esto vamos a hablar más adelante, y se lo van a encontrar después de la materia.
[6] Esto tiene que ver con la idea de tipado, en un lenguaje fuertemente tipado debo declarar de antemano que dos objetos podrán ser utilizados polimórficamente. Como ya se habrán dado cuenta Smalltalk es débilmente tipado, los objetos no tienen ninguna indicación explícita de tipo.

- La capacidad de escribir código que puede aplicarse a muchos objetos, incluso objetos todavía no implementados. Esto se ve claramente en la la simulaciónGlotona, que sirve para todas las aves que tenga definidas y también para las que vaya a definir en el futuro, siempre sin tocar la la simulación.
- Una guía al implementar nuevos objetos, porque empiezo a entender qué mensajes esperan los que van a ser usuarios. En el ejemplo, al implementar una nueva ave me queda claro qué mensajes tiene que entender. En Smalltalk este tipo de información va en comentarios u otra documentación; en otros lenguajes hay construcciones específicas para esto.
- La posibilidad de usar los objetos que construyo en formas inesperadas. Si defino varias simulaciones, siempre respetando los mensajes trabajáCon: y daleMasa (o sea, de forma tal que las simulaciones sean potencialmente polimórficos); después puedo armar una interfaz gráfica en la que: muestro las simulaciones y aves disponibles, permito al ornitólogo elegir una simulación y un ave, y con un botón "dale masa".

… y más ventajas que iremos viendo.

En resumen, usar y definir objetos polimórficos me da ventajas tanto para la potencia de los objetos que construyo como para su simplicidad de uso.

Para cerrar esta parte: ¿cuántos objetos se necesitan para que haya polimorfismo?
La respuesta es ¡tres!, porque para que haya polimorfismo tiene que haber dos objetos polimórficos y otro que los usa. En nuestro ejemplo los objetos son `pepe`, `pepita` y `simulaciónGlotona`.

Interacción y referencias

Volvamos a la forma que ahora tiene el ornitólogo para evaluar la simulaciónGlotona sobre pepita:

```
simulaciónGlotona trabajáCon: pepita.
simulaciónGlotona daleMasa.
```

Supongamos que el ornitólogo evalúa estas dos líneas de a una y en orden. Vamos a ver en detalle qué es lo que pasa cuando se evaúa cada una, metiéndonos adentro de cada objeto; mediante este análisis esperamos aclarar algunas cosas.

Empecemos con la primera línea. El ornitólogo le envía a la simulaciónGlotona el mensaje trabajáCon: con pepita como parámetro: se evalúa el método correspondiente, que transcribimos

```
trabajáCon: unAve
    "Le ordenan trabajar con un ave determinada"
    ave := unAve
```

En esta evaluación el parámetro `unAve` "es" `pepita`. La única línea de código es una evaluación, que asigna a la variable `ave` de la `simulaciónGlotona` aquello que "sea" (o "tenga") el parámetro `unAve`; es lógico pensar que después de que esa línea se evalúa, esta variable pase a "ser" (o "tener") a `pepita`.

¿Qué es exactamente eso que llamamos "ser" o "tener"? Para entenderlo tenemos que hacer algunas definiciones.

Las variables que forman el estado interno de cada objeto, y los parámetros de los métodos en cada envío del mensaje correspondiente, son ***referencias*** a objetos.

Una variable en Smalltalk (y en muchos lenguajes que implementan la orientación a objetos) es una referencia a un objeto[7].

La forma de entender la asignación es verla así

```
variable := expresión
```

después de evaluar la asignación, la variable referencia al resultado de la expresión, que es un objeto. La expresión puede ser otra variable (como en este caso, `ave := unAve`), cuyo resultado es el objeto al que apunta la variable.

El envío de un mensaje también es una expresión, cuyo resultado es (obviamente) el objeto que devuelve el mensaje.

Al enviar un mensaje, el parámetro (recordar la notación `objeto mensaje parámetro`) es una expresión; en la evaluación del método correspondiente, el parámetro del método referencia al resultado de esa expresión.

El objeto también es una expresión, el mensaje se le envía a su resultado. P.ej. los números enteros entienden el mensaje `even`, que devuelve `true` o `false`[8] según si el receptor es par o no. Si evaluamos

```
(3+5) even
```

el objeto que recibe el mensaje `even` es el 8.

Hagamos un paréntesis del ejemplo anterior y veamos otro rápidamente para repasar lo que aprendimos recién. Supongamos que le agregamos este método a pepita

```
amigo
    "Un amigo de pepita"
    ^pepe.
```

que en un momento dado la energía de pepita es 80, y que evaluamos

```
(pepita amigo) comió: ((pepita energía) - 50)
```
[9]

La evaluación de esta línea resulta en cuatro envíos de mensajes:

- se le envía a `pepita` el mensaje `amigo`
- se le envía a `pepita` el mensaje `energía`

[7] Una variable ¿puede no estar referenciando a ningún objeto? No, siempre está referenciando a algún objeto. Hay un objeto especial, llamado `nil`, que representa "el no-objeto". Si quiero que una variable referencie "a nada", pongo

```
variable := nil
```

y listo

[8] Obviamente los valores de verdad también son objetos; `true` y `false` son sus nombres.

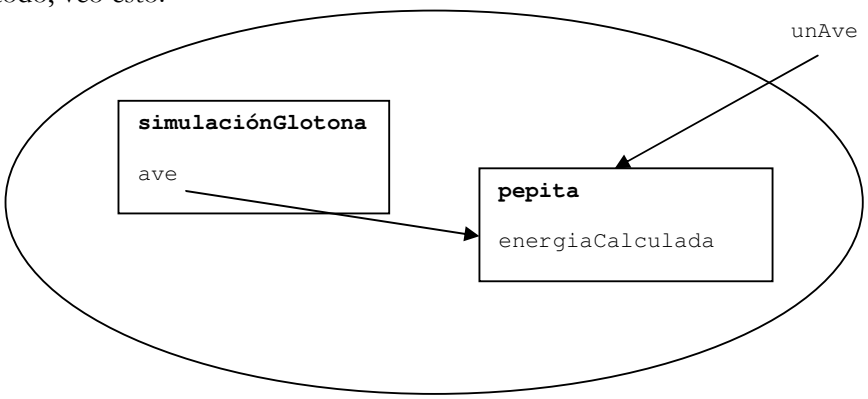
[9] No todos estos paréntesis son necesarios, como veremos más adelante.

- se le envía a `80` (resultado de `pepita energia`) el mensaje − con parámetro `50`
- se envía el mensaje `comió:` con receptor `pepe` (por ser el resultado de la expresión `pepita amigo)` y parámetro `30` (por ser el resultado del mensaje anterior).

En particular, en el envío de `comió:` se ve bien que tanto el receptor como el parámetro son el resultado de expresiones.

Volviendo al ejemplo anterior: `unAve` es una referencia a `pepita` (por lo que dijimos del parámetro aplicado a `simulaciónGlotona trabajáCon: pepita`), y luego de evaluar la asignación la variable `ave` del `simulaciónGlotona` también referencia a `pepita` (por lo que dijimos de la asignación aplicado a `ave := unAve`).

Si saco una "foto" de los objetos luego de evaluar la asignación, y antes de que termine la evaluación del método, veo esto:



Vemos que pepita está referenciada tanto por la variable ave de la simulaciónGlotona como por el parámetro del método trabajáCon: . Cuando se termine el método, la segunda referencia se va y queda solamente la de la simulaciónGlotona.

La variable ave no referencia a "los datos de pepita" o "una copia de pepita" sino al mismo objeto pepita, eso es lo que marca la flecha en el dibujo.

Pasemos entonces a la segunda línea, en la que se envía a la simulaciónGlotona el mensaje daleMasa.
En esta evaluación intervienen varios objetos: simulaciónGlotona, pepita (porque es el objeto referenciado por la variable ave de la simulaciónGlotona), y varios números. En particular los números van entrando y saliendo de escena a medida que transcurre la evaluación.

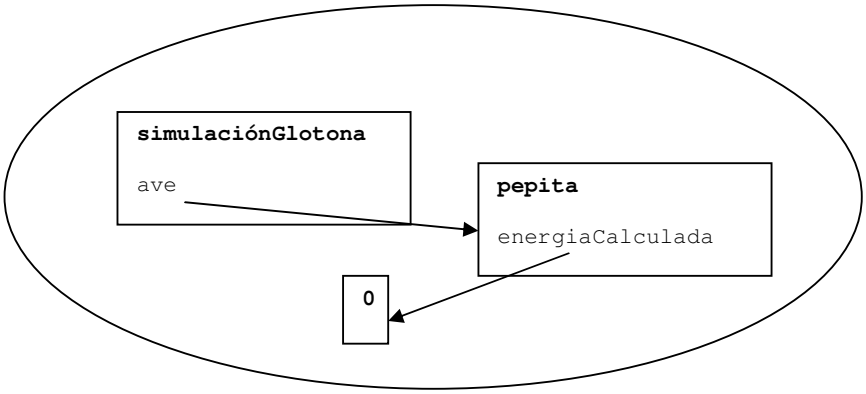
La primer línea del método daleMasa dice

```
ave reset.
```

¿a quién se le envía el mensaje reset? Al objeto referenciado por la variable ave en ese momento, que (por lo que vimos recién) es pepita. El método reset de pepita dice

```
energíaCalculada := 0
```

Después de evaluar esta línea, el dibujo con los objetos queda así:



Vayamos ahora a la siguiente línea del método daleMasa

```
ave comióAlpiste: 50.
```

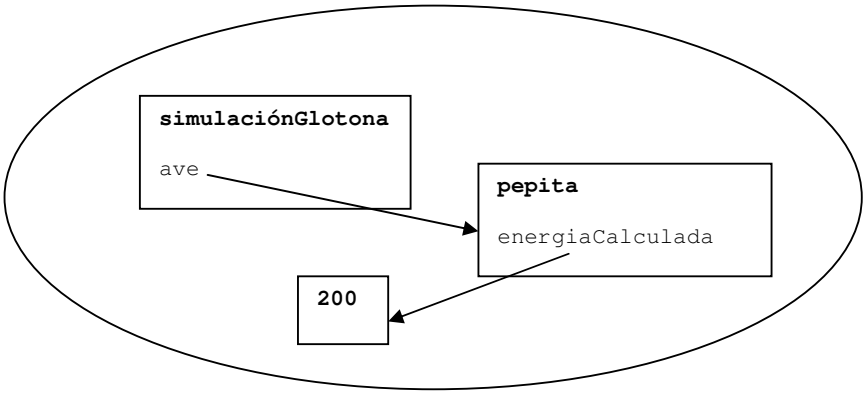
donde el método correspondiente es

```
comióAlpiste: gramos
   energíaCalculada := energíaCalculada + (gramos * 4)
```

aplicando lo que aprendimos queda claro que

- en esta evaluación, gramos referencia a 50
- en esta línea se envía primero el mensaje * a 50 con 4 como parámetro, y después el mensaje + a 0 (objeto referenciado por energíaCalculada) con 200 como parámetro.
- la asignación hace que la variable energíaCalculada apunte al resultado de la expresión de la derecha, que es 200.

El dibujo queda así



y el objeto 50 que usamos en un momento desapareció de escena.
El análisis del resto de daleMasa es similar; no vale la pena detallarlo. Lo único para destacar es que el resultado de un método (en este caso `^ave energía`) también es una expresión.

Resumiendo lo que aprendimos en esta sección:

- las variables son referencias a objetos,

- la asignación indica a qué objeto va a referenciar una variable,
- en varios contextos donde se espera un objeto (parte derecha de la asignación, receptor y parámetro en un envío de mensaje, resultado de un método) lo que se escribe es una expresión, y el objeto al que "le pasa eso" (que la variable referencie a ese objeto, el objeto que recibe el mensaje, etc.) es el resultado de la expresión.

Mensajes a mí mismo – self

Cambiemos de dominio por un ratito. Ahora tenemos un objeto que representa a un cliente de un banco tal como lo ve el oficial de crédito.

El cliente entiende dos mensajes `montoTopeDescubierto` y `montoCreditoAutomatico`, que devuelven hasta cuánto el cliente puede girar en descubierto, y hasta cuánto se le puede dar de crédito sin necesidad de aprobación.

Los dos métodos correspondientes obtienen los valores haciendo cuentas complicadísimas.

Ahora el oficial nos pide a nosotros, que programamos al cliente, que le agreguemos la capacidad de entender el mensaje `montoTotalCredito`, que es la suma de los dos anteriores.

¿Cómo escribo el método, qué tengo que poner? Veamos.

Cuando escribo un método, estoy parado en el objeto que va a recibir el mensaje que estoy habilitando, en este caso el cliente.

Si soy el cliente y me preguntan mi `montoTotalCredito`, lo que tengo que devolver es la suma entre mi `montoTopeDescubierto` y mi `montoCreditoAutomatico`.

Estos dos montos ¿cómo los obtengo? Enviándome los dos mensajes a mí mismo, para no repetir las cuentas que están en esos dos métodos.

¿Y cómo hago para enviarme mensajes a mí mismo? Fácil, usando la palabra clave ***self*** que incluye el Smalltalk. Self es exactamente "yo mismo", enviarle un mensaje a self es enviármelo a mí, donde "yo" soy el objeto que recibió el mensaje por el que se disparó el método.
Entonces queda

```
montoTotalCredito
    "El monto total de credito que me asignaron"
    ^self montoTopeDescubierto + self montoCreditoAutomatico
```

La definición formal:

Self: en un método, referencia al objeto que recibió el mensaje correspondiente, que sirve para enviarme mensajes "a mí mismo"

Ambiente

Prestemos más atención a los dibujos que aparecieron en la sección anterior. ¿Qué es lo que están mostrando esos dibujos? Parecería algún lugar en el cual los objetos "están", "viven", o algo así.

Pasa exactamente eso: para que haya objetos que interactúen, se envíen mensajes y se conozcan entre sí; tienen que estar en algún lugar, tiene que haber algo que los contenga.
A ese lugar lo llamamos ***ambiente*** de objetos.

El Dolphin (como casi cualquier versión de Smalltalk) es ni más ni menos que un ambiente de objetos, en el cual viven los objetos con los que jugamos. Cuando creamos un objeto, se agrega al ambiente.

Ahora, para que un ambiente no crezca indefinidamente, tiene que haber alguna forma de sacar objetos. ¿Cómo hago para sacar un objeto del ambiente?

La respuesta para la mayoría de los lenguajes que soportan la orientación a objetos, incluido Smalltalk, es "el ambiente se arregla". Dicho en forma sencilla, se da cuenta cuando un objeto no tiene referencias, y lo saca. Esta característica es la que se conoce como ***garbage collector***.

Nota antes de seguir: aunque sí es cierto que hay una relación fuerte entre un ambiente de objetos y la memoria de la PC donde corre, no conviene pensar en el ambiente como "el formato que una aplicación con objetos le da a la memoria". Algunos ambientes Smalltalk swappean a disco, existen ambientes distribuidos, hay técnicas para "deshidratar" objetos de un ambiente en una base de datos relacional y después volver a "hidratarlos" cuando hacen falta.

Software

Observemos otra cosa interesante del ejemplo que estamos estudiando. En un momento pasa que:

- el observador le envió a la simulaciónGlotona el mensaje daleMasa, para lo cual
- la simulación debe enviarle a pepita, que es al ave que conoce, el mensaje comióAlpiste: con parámetro 50.
- para resolver ese envío de mensaje, pepita le tiene que enviar al objeto 50 (el parámetro de comióAlpiste:) el mensaje * con el 4 como parámetro.

y esto es congelando la imagen en un momento. Si analizamos todo lo que se dispara a partir del envío del mensaje daleMasa a la simulaciónGlotona, nos damos cuenta que para obtener el resultado hay varios objetos (la simulación, pepita y varios números) que interactúan entre sí.

¿Cómo interactúan los objetos? De la única forma permitida, que es enviándose y recibiendo mensajes.

De esto se trata exactamente el software según el paradigma de objetos:

software (según objetos): objetos que viven en un ambiente y que interactúan entre sí enviándose mensajes.

A partir de esta definición queda más claro que programar consiste en definir qué objetos necesito, definir los mensajes que va a entender y que van a formar su comportamiento, y escribir el código

que soporta cada objeto definido, en donde se incluye enganchar mediante las variables de cada objeto las referencias necesarias para que cada objeto conozca a quienes debe enviarle mensajes.

… ok, ok. Muy linda la idea de pensar en objetos y comportamiento, y el ambiente donde los objetos viven felices, se envían mensajitos y eso anda.

Hasta me puedo creer que puedo escribir la simulación, pepita y pepe así como los contaste (OK, o casi), usar la ventana mágica de interacción, y eso anda y todo.

Pero para hacer software en serio hay mucho más de lo que me tengo que ocupar: manejo de memoria, base de datos, performance, etc.

Si voy a andar definiendo un objeto para cada cosa que necesite, me va a reventar la memoria enseguida. Si cada cosa que quiera hacer tiene que ser un envío de mensaje, se vaa a arrastrar. ¿Cómo hago para construir software en serio si me creo el paradigma de objetos?

No es para nada cierto lo que decís. De hecho, hay proyectos de software nada pequeños que se desarrollaron o se desarrollan usando los conceptos del paradigma en una forma bastante razonable, y que se comportan bastante bien tanto en performance como en uso de recursos[10]. Cuando hubo problemas, más bien se debieron al uso imcompleto o inadecuado de estos conceptos, y no al revés.
Hay muchísimo para decir sobre esto, por ahora destacamos dos factores

- los conceptos de objetos, estos básicos que aparecieron hasta ahora: objeto, mensaje, polimorfismo, interacción, referencias; bien usados, permiten manejar modelos bastante complejos sin que se te vayan de las manos. Esto permite que tengas menos necesidad de hacer chequeos redundantes, repetir cosas, etc., lo que en escala provoca que el soft construido con objetos se hace más eficiente, y no menos.
- en un software bien construido con objetos los problemas de performance y uso de memoria terminan quedando bastante focalizados en lugares bien específicos; al trabajar sobre estos puntos, se logran con relativamente poco trabajo ganancias dramáticas (que un programa corra 10 veces más rápido después de un trabajo de performance no es algo raro).

[10] … en algunos de estos trabajan los autores de este apunte

Capítulo 3 – Clases

Como habrán visto, hasta acá cada uno de los objetos que creamos tienen su propio comportamiento distinto al de todos los demás.

Seguramente a esta altura ya se habrán hecho la pregunta: ¿Qué pasa si quiero tener muchas golondrinas que se comporten igual … o parecido? ¿Tengo que codificar lo mismo de nuevo cada vez?

La forma más habitual (aunque no la única[11]) de resolver ese problema es a partir del concepto de ***clase***. En lugar de definir cada golondrina por separado, defino una clase con las características que serán comunes a las golondrinas, y luego voy a crear los objetos que representan a cada golondrina a partir de esta clase …
… mejor vamos más despacio.

Definición de comportamiento común

La idea de clase nos permite definir un molde para luego a partir de ahí crear muchos objetos (golondrinas en nuestro caso) que se comporten de la misma manera.
Ahora, ¿qué es lo que compartirían todas las golondrinas?, y ¿qué es lo que no queremos que compartan?

Tal vez la primera cosa en la que pienso es en que todas las golondrinas compartan los mismos métodos. Puedo pensar que la fórmula de aumento de energía al comer

```
energíaCalculada := energíaCalculada + (gramos * 4)
```

es válida para todas las golondrinas y entonces quiero ponerla en un único lugar. Ese lugar será la clase.

Bien, eso suena fácil, pero… ¿qué pasa con la energía? Es claro que no queremos que todas las golondrinas tengan la misma energíaCalculada, cada una tendrá su propia energíaCalculada que dependerá de cuánto haya volado y comido.
Por otro lado, hay algo que tienen que compartir, es decir todas tienen que tener una energíaCalculada. Eso permite que la formula de arriba tenga sentido para cualquier golondrina.
Entonces, cada golondrina tendrá su propio valor de energíaCalculada pero a nivel de clase deberemos indicar que todas las golondrinas tienen una variable energíaCalculada.

Hasta acá vemos que la clase me estaría definiendo dos cosas:

- El comportamiento común (métodos) a todas las golondrinas
- El conjunto de variables que tiene que tener cada golondrina.

Veamos cómo quedaría eso:

[11] En otros lenguajes como Self o JavaScript, el concepto de clase no existe y la definición de múltiples objetos similares se hacen a partir de las ideas de prototype, trait y mixin. La programación orientada a aspectos también permite agregar comportamiento común a muchos objetos diferentes.

```
Clase Golondrina

    variables
        energíaCalculada

    métodos
        comióAlpiste: gramos
            "Le indican a la golondrina que comió alpiste;
             el parámetro es la cantidad expresada en gramos"
            energíaCalculada := energíaCalculada + (gramos * 4)


        voló: minutos
            "Le indican a la golondrina que voló;
             el parámetro es la duración del vuelo expresada en minutos"
            energíaCalculada := energíaCalculada – (10 + minutos)


        energía
            "Devuelve la energía del receptor medida en joules"
            ^energíaCalculada


        reset
            "Vuelve a la golondrina a un estado inicial"
            energíaCalculada := 0
```

(obsérvese que el nombre de las clases empieza con mayúscula)

Se darán cuenta que no cambió nada el código que habíamos puesto en el primer capítulo, simplemente pasa que ese código y definición de variables ya no está asociado a pepita sino a todas las golondrinas.
A partir de esta clase, de la definición de los métodos y de las variables que tiene una golondrina, ahora voy a poder crear muchas golondrinas que compartan esta definición.

Cómo se crean objetos

Hasta aquí en ningún momento nos ocupamos del problema de la creación de los objetos; siempre asumimos que una vez definida pepita, esta existía en el ambiente. La aparición del concepto de clase nos permite describir con precisión cuándo y cómo se crean objetos.

El hecho de crear la clase y definir sus variables y métodos no crea ningún objeto.
Entonces, si quisiera tener una nueva golondrina, ¿cómo hago? Escribo una sentencia en la ventana en la que escribo el código que usa los objetos que defino; ahora los voy a crear antes de usarlo.

Si bien a esta altura tal vez no les resulte obvio cómo le pido a Smalltalk que cree un nuevo objeto, hay algo que podemos intuir, ¿qué es?

Obviamente para crear una golondrina[12], en algún lugar de la sentencia tendré que referenciar a la clase Golondrina, ya que es el molde que voy a usar para crearla.
La sintaxis en Smalltalk es la siguiente:

```
Golondrina new
```

Esto me devolverá un objeto nuevo, de la clase Golondrina. Decimos que este nuevo objeto es ***instancia*** de la clase Golondrina, ya que fue creado utilizando a la clase como molde.
Por el momento tomaremos esa sentencia como está, más adelante analizaremos con más detalle lo que está pasando ahí[13].

Interactuando con instancias

Ahora veamos cómo utilizar el objeto que acabamos de crear desde la ventana de interacción.

```
pepita := Golondrina new.
pepita reset.
pepita comióAlpiste: 50.
pepita voló: 5.
pepita comióAlpiste: 30.
pepita voló: 10.
pepita energía
```

Como habrán notado, lo único que cambia es la primera línea, en la cual le damos un valor a la variable pepita.

Analicemos más detenida mente esa primera línea, ¿qué está pasando ahí?
En primer lugar, al decir `Golondrina new` se crea un objeto que es instancia de la clase `Golondrina`; decimos que se está ***instanciando*** una `Golondrina`.

Al mismo tiempo, además se hace referencia a la variable `pepita` por primera vez, eso hace que se cree la variable `pepita` en nuestro entorno de trabajo.

Y finalmente se asigna (`:=`) a la variable `pepita` una referencia al objeto que acabamos de crear. A partir de ahora, `pepita` referencia a ese nuevo objeto.

Luego de crear a pepita, la usamos enviándole mensajes.
¿Qué mensajes entenderá pepita? Aquellos cuyo nombre coincide con el nombre de un método de la clase de la que pepita es instancia, que es Golondrina.
Y cuando pepita recibe un mensaje, el código que se ejecuta es el del método de Golondrina que tiene el mismo nombre. El código pasó de los objetos a las clases.

Nota antes de seguir: en adelante diremos "la clase de pepita" como una forma abreviada de "la clase de la cual pepita es instancia".

Clases e instancias – recapitulemos un poco

Volvamos un poco a la palabra ***instancia***. Dijimos que el objeto creado por la sentencia

[12] En rigor, un objeto que representa una golondrina; en adelante nos tomaremos la licencia de decir "un X" cuando en realidad nos referimos a un objeto que representa a un X.
[13] Para los curiosos: sí se está respetando la sintaxis **objeto mensaje** … y los detalles van más adelante.

```
Golondrina new
```

es una ***instancia*** de la clase Golondrina.

Entonces ¿qué son las instancias? Son los objetos, exactamente eso. La palabra instancia se usa cuando queremos remarcar la relación entre los objetos y las clases, p.ej. para preguntarnos de qué clase es instancia un determinado objeto.

Cada objeto es instancia de exactamente una clase, que es la que se usó de molde para crearlo[14].

Terminemos esta sección bajando las definiciones de clase e instancia.

clase: molde a partir del cual se crean los objetos; y en el que se definen los métodos y el conjunto de variables que tendrán los objetos que se creen a partir del molde

instancia: cada objeto es instancia de la clase que se usó como molde para crearlo.
Cada objeto es instancia de exactamente una clase.
Las instancias de una clase entienden los mensajes para los cuales hay métodos definidos en la clase[15].

Refinando la vista de un ambiente de objetos

Pasemos ahora al ejemplo del capítulo 2 en el que una simulación glotona interactúa con una golondrina.

Como todo objeto es instancia de exactamente una clase, eso también va a pasar con la simulación. Llamemos a la clase SimulaciónGlotona, y supongamos que tiene el mismo código que indicamos para la simulación en el capítulo 2.

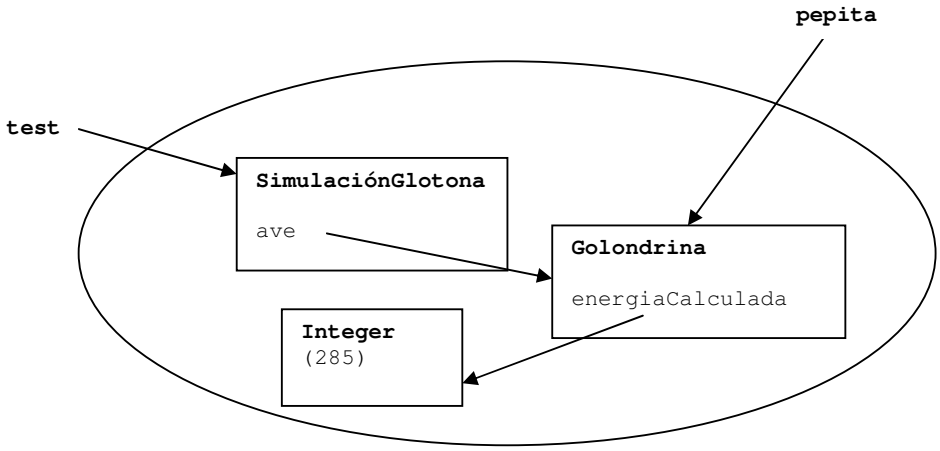
Si tuviéramos otra simulación, deberíamos crear otra clase porque el código del método daleMasa sería distinto al de la simulación glotona, y si el código de un método es distinto entonces ya no puede ser la misma clase.
Esto nos lleva a otra pregunta: ¿cómo hacemos si queremos dos objetos cuyos métodos sean muy parecidos, pero no iguales, se puede hacer algo así? Sí se puede, y lo vamos a ver un poco más adelante.

En la ventana de interacción deberemos instanciar una simulación y una golondrina, asociarlos, y pedirle a la simulación que se ejecute. Nos queda

```
pepita := Golondrina new.
simulación := SimulaciónGlotona new.
simulación trabajáCon: pepita.
simulación daleMasa.
```

[14] El manejo de la creación de objetos es un aspecto muy importante en el diseño de una aplicación construida usando la orientación a objetos. Por eso, es probable que las sentencias que uses para crear objetos en varios contextos son sean `Clase new`, sino envíos de mensajes que adentro invocan al new de la clase que corresponda.
Esto no quita que la creación del objeto, su agregado en el ambiente, ocurre al evaluarse el new; esto es siempre así vea uno o no el new en el código que escribe.
[15] … pero no solamente, hay mensajes que un objeto entiende y cuyo método no está en su clase. Volveremos sobre esto más adelante.

Luego de ejecutarse este código, la "foto" de los objetos queda de esta forma



A grandes rasgos el dibujo es igual que antes: hay objetos que tienen referencias entre ellos.

Hay una diferencia importante: los objetos ya no tienen nombre propio, los únicos nombres que aparecen son los de las variables que los referencian.

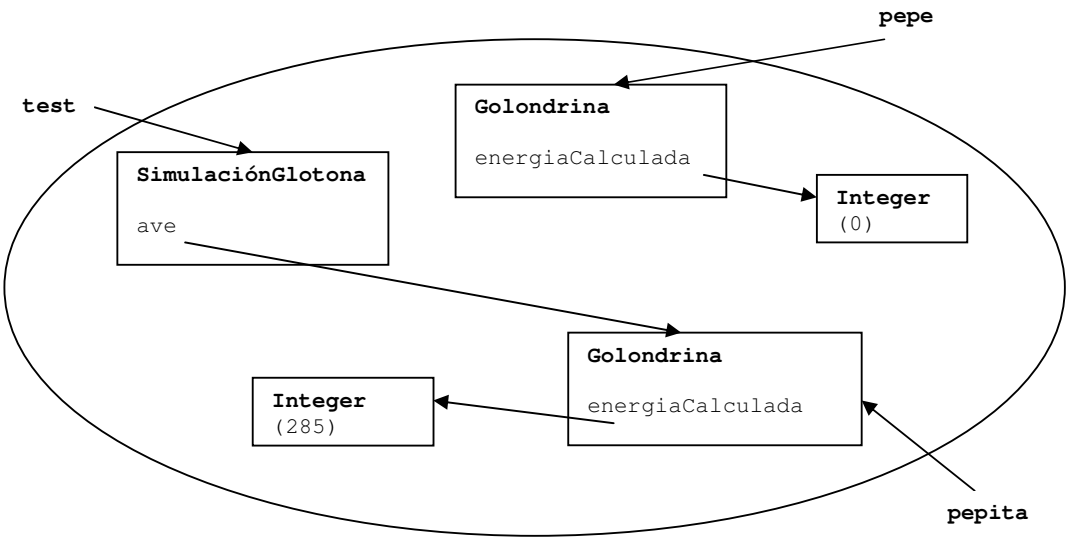
En cada caja que representa un objeto indicamos su clase.

Para el número elegimos una notación especial, que intenta reflejar que representa el 285 por sí mismo, no tiene una referencia al "valor" 285. Insistimos en que no puede haber algo como un "valor", todo lo que quiera representar lo deberé representar mediante un objeto.

Si instanciáramos otra golondrina, agregando al código anterior

```
pepe := Golondrina new.
pepe reset.
```

tendríamos este dibujo



Hay dos golondrinas, y la simulación actúa sobre una de ellas.

Forma verdadera del código Smalltalk

Con lo que vimos sobre clases e instancias, ahora sí podemos escribir código Smalltalk usando las herramientas standard del Dolphin, y jugar con los objetos que quedan definidos.

En Smalltalk las cosas son como las describimos en este capítulo, en particular las que difieren de lo que hablamos en los dos anteriores:

- Se definen clases.
- Los métodos se escriben en las clases, y las variables se definen para las clases. O sea, todo el código está en las clases, excepto el que ponemos en ventanas de interacción.
- Los objetos se crean instanciándolos, para lo cual se usa la sentencia Clase new.
- Los objetos no tienen nombre propio.

El código que aparece en este capítulo no puede ser transcripto así como está en el Dolphin por un detalle: Dolphin no acepta acentos en los nombres de métodos ni en los de de variables. Por esta razón, en lo que sigue vamos a usar nombres sin acentos.