Valeria Samoilenko, Fotolia

Codificación fácil con Smalltalk y Squeak

PEQUEÑO AMIGO

El venerable lenguaje Smalltalk está viendo resurgir un nuevo interés hacia él gracias a nuevas implementaciones como la del popular Squeak. **POR RANDAL L. SCHWARTZ**

¿Por qué habría que usar Smalltalk si ya se usa su heredero natural, Ruby? De hecho, Smalltalk ha resurgido en los últimos años en parte debido a la fabulosa aplicación *Smalltalk Seaside*, un framework web. Smalltalk combina adecuadamente funcionalidades que fomentan y alientan la programación de desarrollo rápido (fueron programadores de Smalltalk los que iniciaron los movimientos de programación *agile* y *test-driven*). Smalltalk es también estupendo para *metaprogramación*: modificando o ampliando el lenguaje y el entorno para tareas particulares. En Ruby, esto a veces se vende como nueva característica a modo de “lenguajes específicos de dominio”, pero los programadores de Smalltalk ya lo hacían mucho antes de que lo hiciera Ruby. Smalltalk es también más amigable, con una sintaxis y un diseño de base simples y unas potentes capacidades de reflexión y auto-modificación.

De hecho, hay al menos dos sólidas implementaciones de Smalltalk de código abierto, además de varias implementaciones comerciales.

Un Lenguaje Simple

Una de las cosas que suele evocar Smalltalk es su elegante simplicidad. Todo en Smalltalk son objetos. Eso es, todo. Cada objeto pertenece a una clase, que es a su vez un objeto. Casi todas las clases tienen una sola superclase, de la que todos los objetos de esa clase heredan métodos. La excepción es la clase *Object*, que reside en la cúspide de la jerarquía. Los objetos tienen estados individuales almacenados en variables de instancia. Ese estado lo gestiona directamente el propio objeto.

Variables

Para dar nombre a las variables se emplean identificadores alfanuméricos, es decir, solamente números y letras. La mayoría de las variables de nombres compuestos están escritos en *CamelCase* (esto es, palabras que tienen su primera letra en mayúscula y se unen para formar una única palabra), y casi todas las variables locales empiezan por minúscula, como *rate* o *accelerationRate*.

Las variables empiezan almacenando un valor inicial nulo (un valor especial creado para este propósito). Ese valor se puede cambiar después mediante mutaciones o mediante la asignación de un nuevo valor:

```
rate := 30
Message Sends
```

La unidad *message sends* consiste simplemente en un método a invocar, sin argumentos. La sintaxis es muy simple: objeto a la izquierda y nombre del método a la derecha, del siguiente modo:

```
rate squared
```

Que sería similar a un *$rate->squared* en Perl, o *rate.squared()* en Java o JavaScript. Cada mensaje enviado devuelve un valor. El receptor (conocido como *self* dentro del propio método) se devuelve por defecto.

Los messages sends binarios se indican mediante un nombre de símbolo construido a partir de un número limitado de signos de puntuación seguidos de un argumento. Normalmente se usan para operaciones matemáticas típicas:

```
rate * time
```

En este caso, se le pasa el mensaje * al objeto *tasa* con *time* como único argu-

mento. Nótese que es asimétrico, ya que el método * sólo se busca en la clase receptora.

Finalmente, con uno o más argumentos, un message send clave identifica los argumentos mediante etiquetas terminadas en dos puntos:

```
rate raisedTo: 2.5
reate between: 5 and: 10
```

Ya que no existen mensajes con más de un argumento sin etiquetas, se ha eliminado la necesidad de "recordar el orden de los parámetros", que sí es necesario en otros lenguajes.

Métodos

Un método enpieza con una firma, que identifica si se trata de un método unario, binario o clave. La firma tiene la misma sintaxis que el message send, exceptuando el objeto receptor (el primer identificador), ya que está implícito:

```
squared
* unNumero
raisedTo: unNumero
between: numeroBajo and: ➚
numeroAlto
```

Convencionalmente, el parámetro formal da a entender el tipo válido para el parámetro real. La firma podría ir seguida de un número indeterminado de "variables locales" situadas entre barras verticales. Por ejemplo, si un método necesitase las variables cuenta y suma, podríamos escribir:

```
| cuenta suma |
```

El cuerpo del método consta de uno o más message sends separados por puntos. El messages send final podría ir precedido de un acento circunflejo (^), que se interpretará a modo de "respuesta", para indicar que la expresión final es el valor respondido. Por tanto, una definición completa de *squared* podría ser:

```
squared
^self * self.
```

Esta definición significa que cuando enviemos el mensaje *squared* al objeto, éste devolverá el resultado de enviarse * a sí mismo, consigo mismo como único parámetro.

Bloques

Los bloques son similares a los métodos, en cuanto a que tienen parámetros, variables locales y uno o más envíos. A diferencia de los parámetros de los métodos, los de los bloques son siempre posicionales (aunque los bloques suelen tomar solamente uno o ningún parámetros). Los bloques se definen entre corchetes y constan de tres partes (opcionales todas): la lista de argumentos

```
:arg1 :arg2 |
```

seguida de los temporales

```
| temp1 temp2 |
```

y seguidos de la secuencia de messages sends (declaraciones):

```
temp1 := arg1 hazEsto. ➚
temp2 := arg2 hazEsoCon: ➚
temp1. temp2 + temp1
```

A diferencia de las definiciones de métodos, la respuesta predeterminada de un bloque es la última expresión evaluada. Si incluimos una "respuesta" (^) en un bloque, ésta sale del método en el que se definió el bloque.

El bloque se suele usar para definir el código de un *callback* (similar a las subrutinas anónimas de Perl).

```
sumaTres := [:n | n + 3].
```

En este punto, se puede invocar el código con un parámetro:

```
resultado := sumaTres value: 15.
```

Esta declaración pasa *15* como primer parámetro al bloque y copia el resultado (18) a *resultado*.

Bloques para el Control de Estructuras

Aunque los bloques se suelen usar para demorar la ejecución de una parte del código, también se usan como estructuras de control. Por ejemplo, la típica estructura "ejecuta esto si aquello es cierto" se representa así:

```
unBoleano ifTrue: [algun. ➚
codigo. aqui].
```

No hay ninguna sintaxis aquí que no hayamos visto ya. Este ejemplo no es más que un mensaje clave, en el que el argumento es un bloque. En esta simplicidad es donde radica la belleza de la sintaxis de Smalltalk – con lo único que se implementa el resto de "formas especiales" de otros lenguajes es con lo que hemos visto hasta ahora. Por ejemplo, un "if-then-else" no es más que un mensaje clave de dos argumentos:

```
algunaExpresionBoleana ifTrue: ➚
[esto. se. ejecuta. de. ser. ➚
cierta] ifFalse: [y. esto. de. ➚
ser. falsa].
```

Por conveniencia, Smalltalk las define también a la inversa:

```
algunaExpresionBoleana
ifFalse: [aquí la rama falsa]
ifTrue: [aquí la verdadera].
```

Y los bucles se definen de forma similar:

```
[cosas. para hacer aqui. ➚
algo queDevuelve unBoleano] ➚
whileTrue. [otraCosa. ➚
  algunaExpresionBoleana]➚
whileTrue: [haz esto. ➚
antes de empezar de nuevo].
```

También existe una versión *whileFalse* de esos bucles. Por ejemplo, supongamos que queremos hallar la menor potencia de dos de un tamaño mínimo determinado. Una implementación simple podría ser:

```
algunNumero := 18.
powerOfTwo := 1.
[powerOfTwo < algunNumero]
whileTrue: [powerOfTwo :=
powerOfTwo * 2].
```

Squeak

Squeak es la implementación de código abierto más popular de Smalltalk. Una instalación de Squeak consta de cuatro partes:

- Un archivo *image*, que comienza como una publicación de Squeak particular; luego le vamos añadiendo o quitando cosas hasta formar nuestro entorno o despliegue específico.
- Una máquina virtual (MV) de arquitectura específica, que es la única parte que podría variar en caso de mover cosas de una máquina a otra.
- Un archivo *sources* en el mismo directorio que la MV, que depende del archivo de imagen con el que hayamos empe-

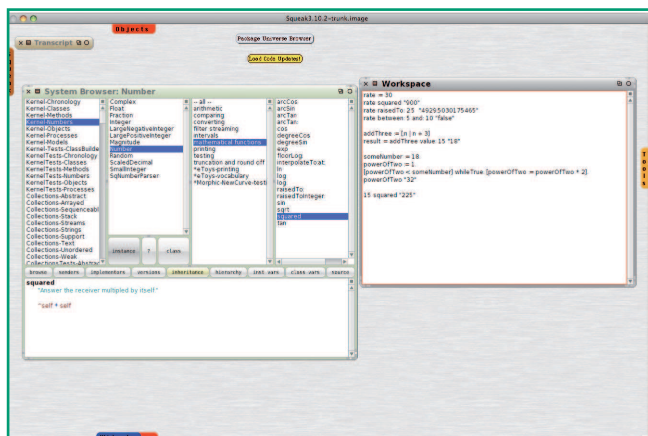

Figura 1: Creamos un espacio de trabajo para probar expresiones simples.

zado, pero que cambia una sola vez cada pocos años.
- Un archivo *changes* para reflejar las diferencias del código entre el archivo de fuentes y nuestra imagen particular.

En general, tendremos una sola MV y un solo archivo de fuentes en el disco, así como uno o más pares coordinados de archivos de imagen y de cambios conforme trabajamos en varios proyectos.

En la página de descargas de Squeak [1] están enumeradas las cuatro partes. En caso de no encontrar la máquina virtual precompilada para una arquitectura específica, disponemos también de los fuentes, aunque su compilación es algo delicada. Además, querremos usar el archivo *SqueakV3.9.sources* para las imágenes modernas.

A la hora de elegir las imágenes que contendrán nuestro entorno de desarrollo y “el núcleo de Squeak”, que es como se le suele llamar, disponemos de dos opciones. La más simple es “Basic Squeak Release” (actualmente la 3.10.2), que aparece listada en la página de descargas, un par imagen/cambios con unos dieciocho meses de antigüedad (la fecha de la última publicación principal). Sin embargo, tiene muchos bugs conocidos y su desarrollo se ha descarrillado un poco. Puede que para cuando salga este artículo publicado se haya liberado ya la versión Squeak 4.0, en cuyo caso bastaría con seguir las instrucciones que se dispusieran en la página de descargas.

Arrancar la Imagen

Para arrancar la imagen desde el entorno de escritorio debemos asegurarnos de que la MV y el archivo de fuentes se encuentran en el mismo directorio, y de que la imagen y el archivo de cambios están también en un mismo directorio (que puede ser el directorio que contiene las fuentes y la MV, o ser otro distinto). Arrastrando a Squeak el archivo con la imagen, ésta debería arrancar.

Aparecerá entonces una sola ventana de sistema operativo con el “escritorio” de Squeak en ella. Squeak dispone de su propio escritorio, llamado *World*, y un estilo de ventanas propio. La razón por la cual se ejecuta Squeak de una forma precisa en plataformas tan diversas, es que tienen lugar exactamente las mismas interacción y presentación en *Windows*, en *X11*, en *Mac OS X* o en cualquier otra interfaz de ventanas de carácter exótico. Por desgracia, quizá signifique esto que tengamos que invertir algo más de tiempo familiarizándonos con el entorno, pero todo el que haya probado antes más de un sistema de ventanas, sabrá que no es tarea imposible.

Ratones y Colores

Squeak usa también terminología un poco rara en lo que a clicks de ratón se refiere. Con las máquinas originales en las que se desarrolló Squeak se usaban ratones de tres botones. Los botones izquierdo, central y derecho estaban coloreados (literalmente) en rojo, amarillo y azul respectivamente.

El botón rojo, el izquierdo, se usa para cosas que se consideran apropiadas para un click “normal” (o click izquierdo). Por tanto, nuestros clicks primarios funcionarán bien cuando la intención sea seleccionar y mover cosas de un lado para otro.

El botón amarillo, el central, se usa para las cosas correspondientes al click “contextual”, ya que su pulsación suele mostrar un menú que varía dependiendo de la posición en que se encuentre el ratón en ese momento. Este click se correspondería con el botón derecho de una máquina corriente, o incluso con el central de algunas; queda en manos del lector determinar con cuál se correspondería en su caso. Y para las máquinas con ratones de un solo botón, habrá que usar una tecla modificadora, como *Ctrl* o *Mayús*, para poder acceder a este menú.

El botón azul o derecho se usa a modo de segundo click “contextual”. En Squeak, se utiliza principalmente para mostrar un halo alrededor de los widgets de la pantalla (llamados *Morphs*). Una vez más, en nuestro entorno particular podría tratarse del click derecho, el central, o el obtenido mediante una tecla modificadora.

Un Espacio para Trabajar

A fin de probar expresiones simples, podemos abrir un espacio de trabajo (ver Figura 1). Un espacio de trabajo no es más que un área de texto en el que podemos escribir cosas y luego ejecutarlas, bien para realizar una acción (un *do it*) o bien para ver qué valor resulta de la acción (un *print it*).

De no haber ningún espacio de trabajo en el World actual, haremos click izquierdo (botón rojo) sobre el escritorio. Con esto, se muestra el menú del World, en el que, cerca de la parte superior, encontraremos el elemento *Workspace*. Al seleccionarlo, aparecerá un espacio de trabajo. Dentro de ese espacio de trabajo podremos introducir cualquier expresión, como por ejemplo *15 squared*.

Con el cursor en el lado derecho de la expresión, seleccionamos el menú contextual (click central, botón amarillo) y escogemos *print it* (cuyo atajo sería la letra *p*). Nada más seleccionarlo aparecerán los resultados (*225*).

Para saltarnos el menú contextual usaremos los atajos. Al pulsar la tecla de atajo (que sólo podrá ser una de dos, mayúscula o minúscula, por lo que habrá que prestar atención al detalle) con la tecla modificadora adecuada (que podría ser una de *Ctrl*, *Opt*, *Alt* o *Cmd*) se invocará la misma operación.

¿Algo Más?

Con lo explicado en este artículo no hemos hecho más que arañar la superficie de la programación con Smalltalk y Squeak, quedándose en el tintero mucho sobre cómo usar la imagen y cómo escribir el código. La mejor fuente de Squeak en estos momentos es un libro “donationware” que hay disponible para descarga en [2]. También existen otros recursos en el sitio web de Squeak, como una lista de correo para principiantes, un canal de IRC o docenas de otros libros disponibles de forma gratuita. ■

RECURSOS

[1] Squeak: http://www.squeak.org/Download/

[2] *Squeak by Example*, por Oscar Nierstrasz, Lulu.com, 2009: http://squeakbyexample.org/