Pharo por Ejemplo

Andrew P. Black Stéphane Ducasse
Oscar Nierstrasz Damien Pollet

with Damien Cassou and Marcus Denker

Version of 2013-05-12

This book is available as a free download from http://PharoByExample.org.

Published by Square Bracket Associates, Switzerland. http://SquareBracketAssociates.org

ISBN 978-3-9523341-4-0

First Edition, October, 2009. Cover art by Samuel Morello.

Contents

Prefacio

¿Qué es Pharo?

Pharo es una implementación completa, moderna y de código abierto, del lenguaje y ambiente de programación Smalltalk. Pharo es un derivado de Squeak[1], una reimplementación del clásico sistema Smalltalk-80. Mientras que Squeak fue desarrollado principalmente como una plataforma para desarrollo experimental de software educacional, Pharo se esfuerza en ofrecer una plataforma limpia y de código abierto para desarrollo profesional de software y una plataforma robusta y estable para investigación y desarrollo en lenguajes y ambientes dinámicos. Pharo sirve como la implementación de referencia para el framework Seaside de desarrollo web.

Pharo resuelve algunos temas de licenciamiento de Squeak. A diferencia de versiones anteriores de Squeak, el núcleo de Pharo contiene sólo código que ha sido contribuido bajo licencia MIT. El proyecto Pharo comenzó en Marzo de 2008 como una bifurcación de Squeak 3.9, y la primera versión beta 1.0 fue liberada el 31 de Julio de 2009.

Aunque Pharo remueve muchos paquetes de Squeak, también incluye numerosas características que son opcionales en Squeak. Por ejemplo, las fuentes true type estan incluidas en Pharo. Pharo también incluye soporte para block closures. Las interfaces de usuario han sido simplificadas y revisadas.

Pharo es altamente portable — incluso su máquina virtual está escrita enteramente en Smalltalk, haciendo fácil su depuración, análisis y cambio. Pharo es el vehículo para un ámplio rango de proyectos innovadores desde aplicaciones multimedia y plataformas educacionales hasta ambientes comerciales de desarrollo web.

[1] Dan Ingalls et al., Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97). ACM Press, November 1997 ⟨URL: http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html⟩.

Hay un aspecto importante detrás de Pharo: Pharo no debe ser sólo una copia del pasado sino verdaderamente *reinventar* Smalltalk. Los enfoques Big-bang raramente triunfan, por eso Pharo favorecerá cambios evolutivos e incrementales. Queremos ser capaces de experimentar con características o bibliotecas nuevas importantes. Evolución significa que Pharo acepta equivocaciones y no es su objetivo ser la solución perfecta en un sólo paso grande — incluso aunque nos encantaria. Pharo favorecerá cambios incrementales pequeños pero una multitud de ellos. El éxito de Pharo depende de las contribuciones de su comunidad.

¿Quién debería leer este libro?

Este libro está basado en *Squeak by Example*[2], una introducción de código abierto a Squeak. El libro ha sido libremente adaptado y revisado para reflejar las diferencias entre Pharo y Squeak. El libro presenta varios aspectos de Pharo, comenzando con lo básico y continuando con temas más avanzados.

El libro no enseña cómo programar. El lector debería tener cierta familiaridad con lenguajes de programación. Algún conocimiento de programación orientada a objetos también sería de ayuda.

El libro presenta el ambiente de programación Pharo, el lenguaje y las herramientas asociadas. Serás expuesto a modismos y prácticas comunes, pero el foco está en la tecnologia, no en el diseño orientado a objetos. Cuando sea posible, te mostraremos muchos ejemplos. (Hemos sido inspirados por el excelente libro de Smalltalk[3] de Alec Sharp.)

Hay numerosos otros libros de Smalltalk gratuitamente disponibles en la web pero ninguno de ellos se enfoca específicamente en Pharo. Por ejemplo: http://stephane.ducasse.free.fr/FreeBooks.html

Un consejo

No te fustres por aquellas partes de Smalltalk que no entiendas inmediatamente. ¡No tienes que saber todo! Alan Knight expresa este principio de la siguiente manera: [4]:

[2] http://SqueakByExample.org

[3] Alec Sharp, *Smalltalk by Example*. McGraw-Hill, 1997 ⟨URL: http://stephane.ducasse.free.fr/FreeBooks/ByExample/⟩.

[4] http://www.surfscranton.com/architecture/KnightsPrinciples.htm

No te preocupes.
Los programadores Smalltalk principiantes amenudo tienen problemas porque piensan que necesitan entender todos los detalles de como funciona una determinada cosa antes de poder usarla. Esto significa que les lleva un rato antes de que puedan dominar el `Transcript show: 'Hello World'`. Uno de los grandes avances de la OO es la posibilidad de responder a la pregunta "¿Cómo funciona esto?" con "No me importa".

Un libro abierto

Este libro es un libro abierto en el siguiente sentido:

- El contenido de este libro es publicado bajo licencia Creative Commons Attribution-ShareAlike (by-sa). Resumiendo, puedes compartir y adaptar este libro, siempre que respetes las condiciones de la licencia disponibles en la siguiente URL: http://creativecommons.org/licenses/by-sa/3.0/.
- Este libro describe sólo el núcleo de Pharo. Idealmente nos gustaría movitar a otro para que contribuyan con capítulos de otras partes de Pharo que no hemos descripto. Si quieres participar en este esfuerzo, por favor contáctanos. ¡Nos gustaria ver crecer este libro!

Para más detalles, visita http://PharoByExample.org.

La comunidad Pharo

La comunidad Pharo es amistosa y activa. Esta es una corta lista de recursos que puedes encontrar útil:

- http://www.pharo-project.org es el sitio principal de Pharo.
- http://www.squeaksource.com es el equivalente a SourceForge para proyectos Pharo. Muchos paquetes opcionales para Pharo viven aquí.

Ejemplos y ejercicios

Usamos dos convenciones especiales en este libro.

Hemos intentado proveer tantos ejemplos como nos fue posible. En particular, hay muchos ejemplos que muestran un fragmento de código que puede ser evaluado. Utilizamos el símbolo ⟶ para indicar el resultado que obtienes cuando seleccionas una expresión y ejecutas `print it`:

```
3 + 4  ⟶  7   "if you select 3+4 and 'print it', you will see 7"
```

En caso que quieras jugar en Pharo con estos fragmentos de código, puedes descargar un archivo de texto plano con todo el código de ejemplo desde el sitio web del libro: http://PharoByExample.org.

La segunda convención que usamos, es mostrar el ícono para indicar cuando hay algo para que hagas:

 ¡Adelante y lee el siguiente capítulo!

Agradecimientos

En primer lugar queremos agradecer a los desarrolladores originales de Squeak por hacer disponible como proyecto de código abierto este asombroso entorno de desarrollo Smalltalk.

También queremos agradecer a Hilaire Fernandes y Serge Stinckwich quienes nos permitieron traducir parte de sus columnas en Smalltalk, y a Damien Cassou por contribuir el capítulo de streams.

Queremos agradecer especialmente a Alexandre Bergel, Orla Greevy, Fabrizio Perin, Lukas Renggli, Jorge Ressia y Erwann Wernli por sus detalladas revisiones.

Agradecemos a la Universidad de Berna, Suiza, por gentilmente apoyar este proyecto de código abierto y por albergar el sitio web de este libro.

También queremos agradecer a la comunidad de Squeak por su apoyo al proyecto de este libro, y por informarnos de los errores encontrados en la primera edición del mismo.

Translators. The spanish translators are: Gabriel Falcone, Federico Rodriguez, Rafael Luque, Pablo Lalloni, Max Sarno, Alan Rodas, Patricio Plaza, Mirian Quinteros, Carlos Rodriguez, Pablo Barrientos, Nicolas Paez, Jordi Delgado and Carlos Rodriguez.

Part I

Getting Started

Chapter 1

Un paseo rápido por Pharo

En este capítulo te daremos un paseo de alto nivel por Pharo para ayudarte a sentirte cómodo con el ambiente. Habrá muchas oportunidades para probar cosas, por lo que sería una buena idea que tengas una computadora a mano mientras lees este capítulo.

Usaremos el ícono:  para marcar lugares en el texto donde deberías probar algo en Pharo. En particular, ejecutarás Pharo, aprenderás acerca de las diferentes formas de interactuar con el sistema, y descubrirás algunas de las herramientas básicas. Aprenderás también cómo definir un nuevo método, crear un objeto y enviarle mensajes.

1.1 Primeros pasos

Pharo está disponible de forma gratuita para descargar en http://pharo-project.org. Hay tres partes que necesitas descargar, que consisten en cuatro archivos (ver Figure 1.1).

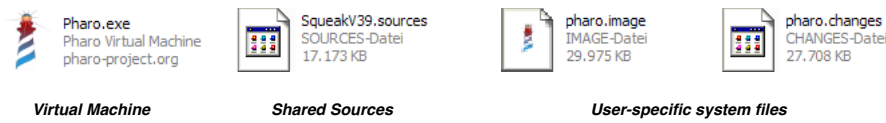


Figure 1.1: Los archivos de Pharo para descargar de una de las plataformas soportadas.

1. La *máquina virtual* (VM) es la única parte del sistema que es diferente para cada sistema operativo y procesador. Hay disponibles máquinas virtuales precompiladas para la mayor parte de los ambientes. En la

Figure 1.1 vemos la máquina virtual para la plataforma seleccionada, su nombre es *Pharo.exe*.

2. El archivo *sources* contiene el código fuente de todas las partes de Pharo que no cambian muy frecuentemente. En la Figure 1.1 su nombre es *SqueakV39.sources*.[1]

3. La *imagen de sistema* actual es un snapshot de un sistema Pharo corriendo, congelado en el tiempo. Consiste de dos archivos: un archivo *.image*, que contiene el estado de todos los objetos en el sistema (incluidos clases y métodos, dado que son objetos también), y un archivo *.changes*, que contiene un log de todos los cambios en el código fuente del sistema. En la Figure 1.1, sus nombres son *pharo.image* y *pharo.changes*.

Descarga e instala Pharo en tu computadora.

Te recomendamos que uses la imagen provista en la página Pharo by Example.[2]

La mayor parte del material introductorio de este libro funciona con cualquier versión por lo que si ya tienes una versión de Pharo, puedes continuar usándola. Sin embargo, si notas diferencias entre la apariencia o el comportamiento de tu sistema y el descripto aquí, no te preocupes.

Al trabajar en Pharo, el archivo de imágen y el archivo de cambios se modifican por lo que necesitas asegurarte que ambos pueden ser escritos. Mantén siempre estos archivos juntos. Nunca los edites directamente con un editor de texto, ya que Pharo los usa para almacenar los objetos con los que está trabajando y para registrar los cambios que haces en el código fuente. Es una buena idea guardar una copia de los archivos de imágen y cambios descargados para poder siempre empezar de una imágen limpia y recargar tu código.

El archivo *sources* y la máquina virtual pueden estar solo para lectura — pueden ser compartidos entre diferentes usuarios. Todos estos archivos pueden ser puestos en el mismo directorio y es también posible poner la máquina virtual y el archivo *sources* en directorios separados donde todos tengan acceso a ellos. Haz lo que funcione mejor con tu estilo de trabajo y tu sistema operativo.

Comenzando. Para arrancar Pharo, haz lo que tu sistema operativo espera: arrastra el archivo *.image* sobre el ícono de la máquina virtual, o haz doble clic en el arhivo *.image*, o en la linea de comandos escribe el nombre de la

[1] Pharo está derivado de Squeak 3.9, y actualmente comparte la máquina virtual con Squeak.
[2] http://PharoByExample.org

Figure 1.2: Una imagen limpia de http://PharoByExample.org.

máquina virtual seguido de la ruta al archivo *.image*. (Si tienes multiples máquinas virtuales en tu máquina el sistema operativo puede no elegir la correcta; en ese caso es más seguro arrastrar y soltar la imagen sobre la máquina virtual, o usar la linea de comandos.)

Una vez que Pharo está ejecutando, deberías ver una única ventana, posiblemente conteniendo algunas ventanas de workspace abiertos (ver Figure 1.2) ¡y no es obvio cómo continuar! Puedes notar una barra de menú, pero principalmente Pharo hace uso de menúes contextuales.

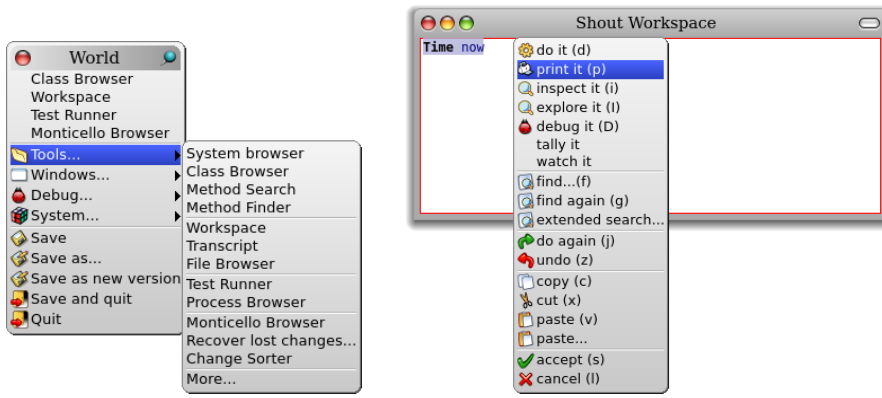
Ejecuta Pharo. Puedes cerrar cualquier workspace abierto haciendo click en el botón rojo en la esquina superior izquierda de la ventana del workspace.

Puedes minimizar ventanas (estas se moverán a la parte inferior de la pantalla) haciendo click en el botón naranja. Haciendo click en el botón verde la ventana ocupará la pantalla completa.

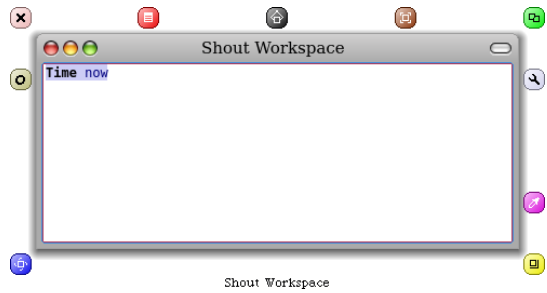
Primera interacción. Un buen lugar para comenzar es el menú world mostrado en Figure 1.3 (a).

Haz clic con el ratón en el fondo de la ventana principal para mostrar el menú world, elige Workspace *para crear un nuevo workspace.*

Smalltalk fué diseñado originalmente para computadoras con un ratón



(a) El menú world (b) El menú contextual



(c) The morphic halo

Figure 1.3: El menú world (lanzado haciendo click), un menú contextual (haciendo action-click), y morphic halo (haciendo meta-click).

de tres botones. Si tu ratón tiene menos de tres botones, tendrás que presionar teclas extras mientras haces click con el ratón para simular los botones faltantes. Un ratón de dos botones funciona bastante bien con Pharo, pero si tienes un ratón de un solo botón, deberías considerar seriamente comprar un ratón de dos botones con rueda de desplazamiento que permita hacer clic: hará que trabajar con Pharo sea mucho más placentero.

Pharo evita términos como "clic con el botón izquierdo" porque en diferentes computadoras, ratones, teclados y configuraciones personales significa que diferentes usuarios necesitarán presionar diferentes botones para lograr el mismo efecto. Originalmente Smalltalk introdujo colores para representar los diferentes botones del ratón.[3] Dado que muchos usuarios usarán teclas modificadoras (*control*, *ALT*, *meta* etc.) para lograr el mismo efecto, en su lugar usaremos los siguientes términos:

[3]Los colores de los botones eran *rojo*, *amarillo* y *azul*. Los autores de este libro nunca podieron recordar qué color refería a qué botón.

hacer click: este es el botón del ratón mas usado, y es equivalente a hacer click con un ratón de un solo botón sin ninguna tecla modificadora; haz click en la imagen para que aparezca el menú "World" (Figure 1.3 (a)).

hacer action-click: este es el segundo botón mas usado; es usado para que aparezca el menú contextual, esto es, un menú que ofrece diferentes conjuntos de acciones dependiendo en dónde está apuntando el ratón; ver Figure 1.3 (b). Si no tienes un ratón multi-botones, normalmente configurarás la tecla modificadora *control* para hacer action-click con el botón del ratón.

hacer meta-click: finalmente, puedes hacer meta-click en cualquier objeto mostrado en la imagen para activar el "morphic halo", un vector de manejadores que son usados para realizar operaciones en los objetos en pantalla, como por ejemplo rotarlos o cambiarles el tamaño; ver Figure 1.3 (c).[4] Si dejas el ratón sobre un manejador, un globo de ayuda te explicará su función. En Pharo, cómo hacer meta-click depende de tu sistema operativo: o bien debes mantener las teclas SHIFT + *ctrl* o bien SHIFT + *option* mientras haces clic.

Escribe Time now *en el workspace. Ahora haz action-click en area de trabajo. Selecciona* print it.

Recomendamos a las personas diestras configurar sus ratones para hacer click con el botón izquierdo, action-click con el botón derecho, y meta-click con la rueda de desplazamiento si está disponible. Si estás usando Macintosh sin un ratón con segundo botón, puedes simularlo manteniendo presionada la tecla ⌘ mientras haces click con el ratón. Sin embargo, si vas a usar Pharo a menudo, te recomendamos invertir en un ratón con al menos dos botones.

Puedes configurar tu ratón para trabajar de la manera que quieras uasndo las preferencias de tu sistema operativo y el controlador del ratón. Pharo tiene algunas preferencias para personalizar el ratón y las teclas meta en tu teclado. En el navegador de preferencias (System ... ▷ Preferences ... ▷ Preference Browser...), la categoría keyboard contiene una opción swapControlAndAltKeys que cambia las funciones action-click y meta-click. También hay opciones para duplicar las diferentes teclas de comando.

[4] Notar que manejadores morphic están inactivos por defecto en Pharo, pero puedes activarlos usando el Navegador de Preferencias, el cual veremos pronto.

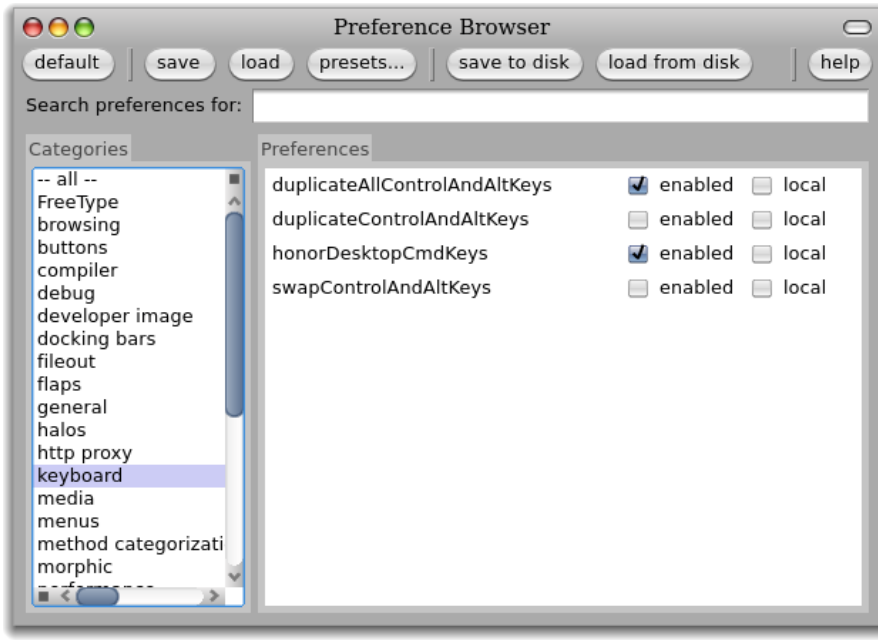


Figure 1.4: El navegador de preferencias.

1.2 El menú World

Haz click nuevamente en el fondo de la ventana principal de Pharo.

Verás el menú World nuevamente. La mayoría de los menúes de Pharo son modales; puedes dejarlos en la pantalla todo el tiempo que desees haciendo click en el ícono de la clavija en la esquina superior derecha. Hazlo.

El menú world te provee una forma simple para acceder a muchas de las herramientas que Pharo ofrece.

Hecha una mirada a los menúes World *y* Tools . . . . *(Figure 1.3 (a))*

Verás una lista de varias herramientas principales de Pharo, incluyendo el navegador y el workspace. Encontraremos la mayoría de ellos en los siguientes capítulos.

1.3 Enviando mensajes

Abre un workspace. Escribe el siguiente texto:

```
BouncingAtomsMorph new openInWorld
```

Ahora haz action-click. Debería aparecer un menú. Selecciona `do it (d)`. *(See Figure 1.5.)*

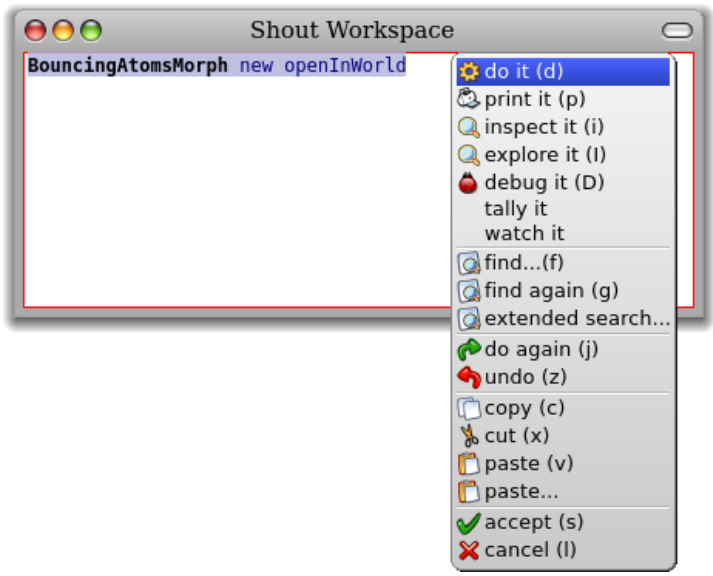


Figure 1.5: "Ejecutando" una expresión

Una ventana conteniendo un gran número de átomos deberían abrirse arriba a la izquierda en Pharo.

¡ Acabas de evaluar tu primera expresión Smalltalk! Haz enviado el mensaje `new` a la clase `BouncingAtomsMorph`, resultando en una nueva instancia de `BouncingAtomsMorph`, seguido del mensaje `openInWorld` a esta instancia. La clase `BouncingAtomsMorph` decidió que hacer con el mensaje `new`, esto es, buscó en sus *métodos* cómo manejar el mensaje `new` y reaccionó apropiadamante. De forma similar la instancia de `BouncingAtomsMorph` buscó en sus métodos cómo responder a `openInWorld` y tomó la acción apropiada.

Si hablas con personas que han usado Smalltalk por algún tiempo, rápidamente notarás que ellos generalmente no usan expresiones como "llamar a una operación" o "invocar un método", en su lugar dicen "enviar un mensaje". Esto refleja la idea de que los objetos son responsables por sus propias acciones. Nunca le *dices* a un objeto qué hacer — en su lugar políticamente le *pides* que haga algo enviándole un mensaje. El objeto, no tú, selecciona el método apropiado para responder a tu mensaje.

1.4 Guardando, finalizando y reiniciando una sesión de Pharo

Ahora haz click en la ventana de átomos y arrástrala dónde quieras. Ahora tienes la demostración "en tu mano". Suéltala haciendo click en cualquier parte.

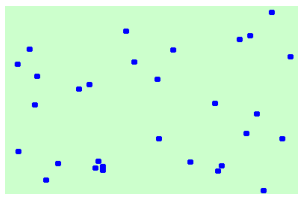


Figure 1.6: Un BouncingAtomsMorph.

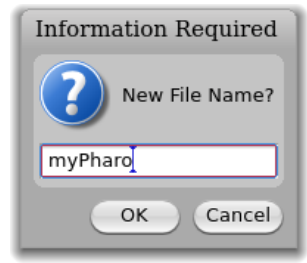


Figure 1.7: El cuadro de díalogo save as ... .

Selecciona World ▷ Save as ... *, ingresa el nombre "myPharo", y haz click en el botón* OK*. Ahora selecciona* World ▷ Save and quit*.*

Ahora si vas a la ubicación donde los archivos originales image y changes estaban, encontrarás dos nuevos archivos llamados"myPharo.image" y "myPharo.changes" que representan el estado de trabajo de la imagen de Pharo en el momento anterior que dijeras a Pharo Save and quit. Si lo deseas, puedes mover estos dos archivos a cualquier lugar que gustes en tu disco, pero si lo haces puedes (dependiendo de tu sistema operativo) necesitar tambien mover, copiar o enlazar a la máquina virtual y el archivo *sources*.

Inicia Pharo desde el archivo "myPharo.image" recientemente creado.

Ahora deberías encontrarte precisamente en el estado que estabas cuando terminaste Pharo. El BouncingAtomsMorph está allí nuevamente y los átomos continuan rebotando donde ellos estaban cuando tú terminaste.

Cuando inicias Pharo por pimera vez, la máquina virtual de Pharo carga la imagen que le sumistres. El archivo contiene una instantanea de un gran número de objetos, incluyendo una vasta cantidad de código preexistente y un gran número de herramientas de programación (los cuales son objetos). Trabajando con Pharo, enviarás mensajes a esos objetos, crearás nuevos objetos, y algunos de esos objetos morirán y su memoria será reclamada (*i.e.*, recolección de basura).

Cuando termines con Pharo, normalmente guardarás una instantanea que contiene todos tus objetos. Si guardas normalmente, sobreescribirás tu

viejo archivo de imagen con la nueva instantanea. Alternativamente, puedes guardar la imagen con otro nombre, como acabamos de hacer.

Junto al archivo *.image*, hay también un archivo *.changes*. Este archivo contiene una bitácora de todos los cambios del código fuente que haz hecho usando las herramientas. La mayor parte de las veces no necesitarás preocuparte por este archivo para nada. Como podemos ver, sin embargo, el archivo *.changes* puede ser muy útil para recuperación de errores, o rehacer cambios perdidos. ¡Más sobre esto luego!

La imagen con la que haz estado trabajando es un descendiente de la imagen original de Smalltalk-80 creada a finales de los años 70. ¡Algunos de esos objetos han estado allí por décadas!

Podrías pensar que la imagen es el mecanismo principal para almacenar proyectos de software, pero estarías equivocado. Como veremos pronto, hay muchas mejores herramientas para administar código y compartir software desarrollado por equipos. Las imagenes son muy útiles, pero deberías aprender a ser muy caballero acerca de crear y tirar imagenes, dado que herramientas como Monticello ofrecen una mejor forma de administrar versiones y compartir código entre desarrolladores.

Usando el ratón (y la tecla modificadora apropiada), haz meta-click en el BouncingAtomsMorph.[5]

Verás una colección de círculos coloridos que colectivamente son llamados el morphic halo de BouncingAtomsMorph. Cada círculo es llamado un *manejador*. Haz click en el manejador rosa conteniendo la cruz; el BouncingAtomsMorph debería haberse ido.

1.5 Workspaces y Transcripts

Cierra todas las ventanas abiertas. Abre un transcript y un workspace. (El transcript puede ser abierto desde World ▷ Tools ... *submenú.)*

Posiciona y cambia el tamaño de la ventana transcript y de la ventana workspace para que el workspace se superponga al transcript.

Puedes cambiar el tamaño de las ventanas arrstrando una de sus esquinas, o haciendo meta-click en la ventana para hacer aparecer el morphic halo, y arrastrando el manejador amarillo (abajo a la derecha).

Al mismo tiempo sólo una venta está activa; está en el frente y tiene sus bordes iluminados.

[5] Recuerda, quizás tengas que activar la opción halosEnabled en el Navegador de Preferencias.

El transcript es un objeto que a menudo es usado para emitir mensajes de sistema. Es un tipo de "consola de sistema".

Los workspaces son útiles para escribir porciones de código Smalltalk con las que te gustaría experimentar. Puedes tambien usar workspaces simplemente para escribir texto arbitrario que te gustaría recordar, como una lista de tareas o instrucciones para alguien que usará tu imagen. Los workspaces son a menudo usados para mantener documentación acerca de una imagen capturada, como es el caso de la imagen estandar que descargamos anteriormente (ver Figure 1.2).

Escribe el siguiente texto en el workspace:

```
Transcript show: 'hello world'; cr.
```

Intenta hacer doble-click en el workspace en varios puntos en el texto que acabas de escribir. Notarás como una palabra entera, una cadena de caracteres entera, o el texto completo es seleccionado, dependiendo si haces click en una palabra, al final de una cadena de caracteres, o al final de toda la expresión.

Selecciona el texto que acabas de escribir y haz action-click. Selecciona `do it (d)`.

Notarás como el texto "hello world" aparece en la ventana transcript (Figure 1.8). Hazlo nuevamente. (La `(d)` en el item de menú `do it (d)` dice que el atajo de teclado para *do it* es CMD–d. ¡Mas sobre esto en la siguiente sección!)

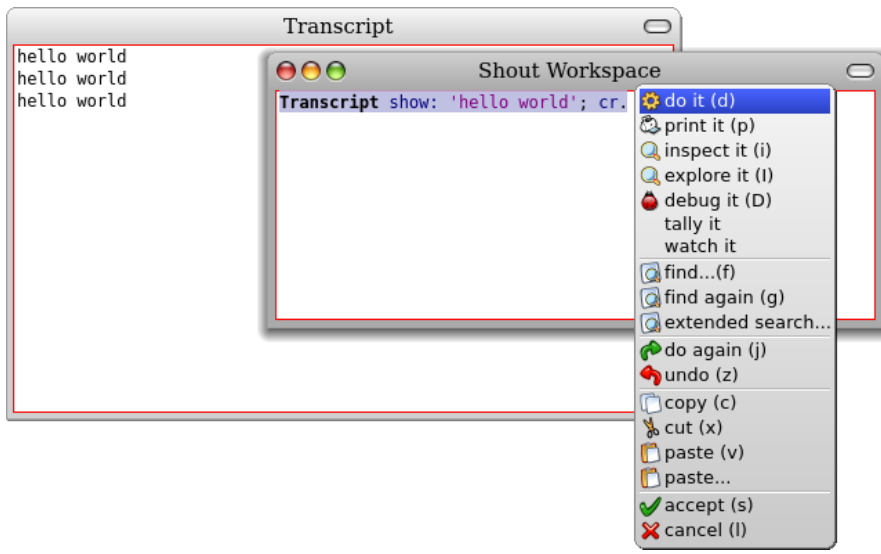


Figure 1.8: Superponiendo ventanas. El workspace está activo.

1.6 Atajos de teclado

Si quieres evaluar una expresión, no siempre tienes que hacer action-click. En su lugar, puedes usar atajos de teclado. Estos son las expresiones entre paréntesis en el menú. Dependiendo de la plataforma, puedes tener que presionar una de las teclas modificadoras (control, alt, command, o meta). (Indicaremos estas genéricamente como CMD−*tecla*.)

Evalua la expresión en el workspace nuevamente, pero usando el atajo de teclado: CMD−*d*.

Además de `do it`, habrás notado `print it`, `inspect it` y `explore it`. Hechemos una rápida mirada a cada una de estas.

Escribe `3 + 4` *en el workspace. Ahora* `do it` *con el atajo de teclado.*

¡No te sorprendas si no viste nada ocurrir! Lo que acabas de hacer es enviar el mensaje `+` con el argumento `4` al número `3`. Normalmente el resultado `7` habrá sido sido computado y retornado a ti, pero dado que el workspace sabía que hacer con la respuesta, simplemente se deshizo de ella. Si quieres ver el resultado, deberías elegir en su lugar `print it`. `print it` realmente compila la expresión, la ejecuta, envía el mensaje `printString` al resultado, y muestra la cadena de caracteres resultante.

Selecciona `3+4` *y selecciona* `print it` *(*CMD−*p).*

Esta vez vemos el resultado esperado (Figure 1.9).

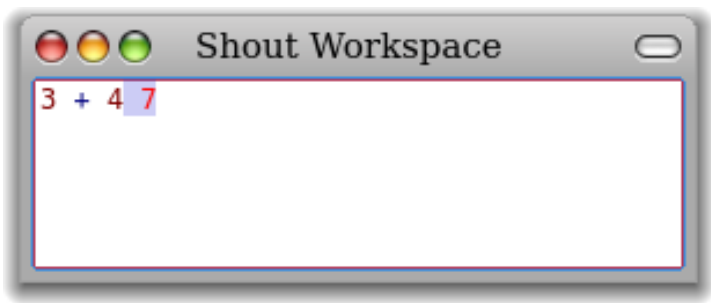


Figure 1.9: “print it” en lugar de “do it”.

```
3 + 4    ⟶    7
```

Usamos la notación ⟶ como una convención en este libro para indicar que una expresión particular en Pharo devuelve un resultado dado cuando usas `print it`.

Borra el texto iluminado "7" (Pharo debería haberlo seleccionado, entonces simplemente puedes presionar la tecla suprimir). Selecciona 3+4 *nuevamente y esta vez usa* inspect it (CMD−*i*).

Ahora deberías ver una nueva ventana, llamada *inspector*, con el encabezado SmallInteger: 7 (Figure 1.10). El inspector es una herramienta extremadamente útil que te permitirá navegar e interactuar con cualquier objeto en el sistema. El título nos dice que 7 es una instancia de la clase SmallInteger. El panel izquierdo nos permite navegar variables de instancia del objeto, los valores de los cuales son mostrados en el panel derecho. El panel inferior puede ser usado para escribir expresiones para enviar mensajes al objeto.

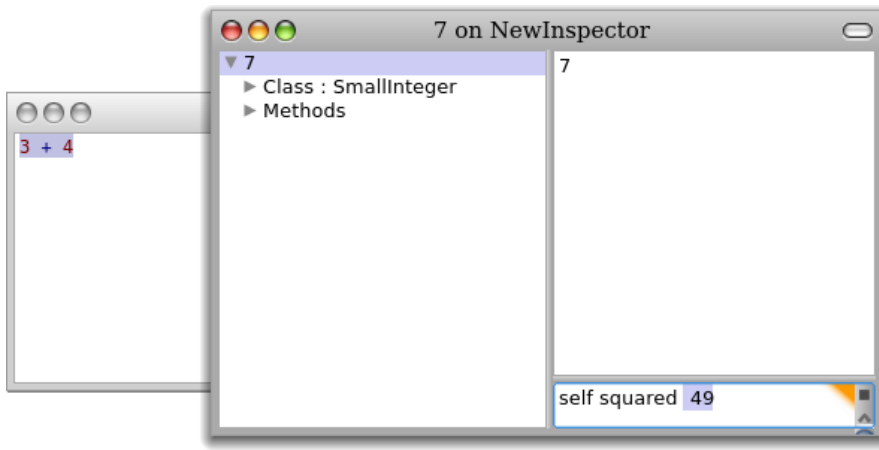


Figure 1.10: Inspeccionando un objeto.

Escribe self squared *en el panel inferior del inspector en* 7 *y usa* print it.

Cierra el inspector. Escribe la expresión Object *en un workspace y esta vez usa* explore it (CMD−*I, mayúscula de i*).

Esta vez deberías ver una ventana llamada Object conteniendo el texto ▷ root: Object. Haz clic en el triángulo para abrirla (Figure 1.11).

El explorador es similar al inspector, pero este ofrece una vista de árbol de un objeto complejo. En este caso el objeto que estamos mirando es la clase Object. Podemos ver directamente toda la información almacenada en la clase, y podemos facilmente navegar todas sus partes.

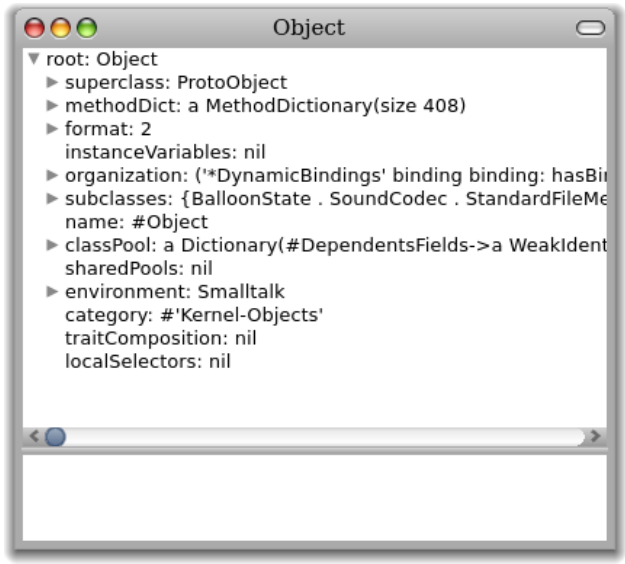


Figure 1.11: Explorando Object.

1.7 El navegador de clases

El navegador de clases[6] es una de las heramientas clave usada para programar. Como podemos ver, hay varios navegadores disponibles para Pharo, pero este es el navegador básico que buscarás en cualquier imagen.

Abre un navegador seleccionando World ▷ Class browser.[7]

Podemos ver un navegador en la Figure 1.12. La barra de título indica que estamos navegando la clase Object.

Cuando el navegador se abre por primera vez, todos los paneles están vacios menos el mas izquierdo. El primer panel lista todo los *paquetes* conocidos, los cuales contienen grupos de clases relacionadas.

Haz clic en el paquete Kernel.

Esto causa que el segundo panel muestre una lista de todas las clases en el paquete seleccionado.

[6]Confusamente, esto es referido varias veces como el "navegador del sistema" o el "navegador de código". Pharo usa la implementación OmniBrowser para el navegador, el cual puede ser también conocido como "OB" o el "navegador de paquetes". En este libro simplemente usaremos el término "navegador", o, en caso de ambiguedad, el "navegador de clases".

[7]Si el navegador que abres no luce como el mostrado en Figure 1.12, necesitas cambiar el navegador por defecto. Vea FAQ 5, p. 332.

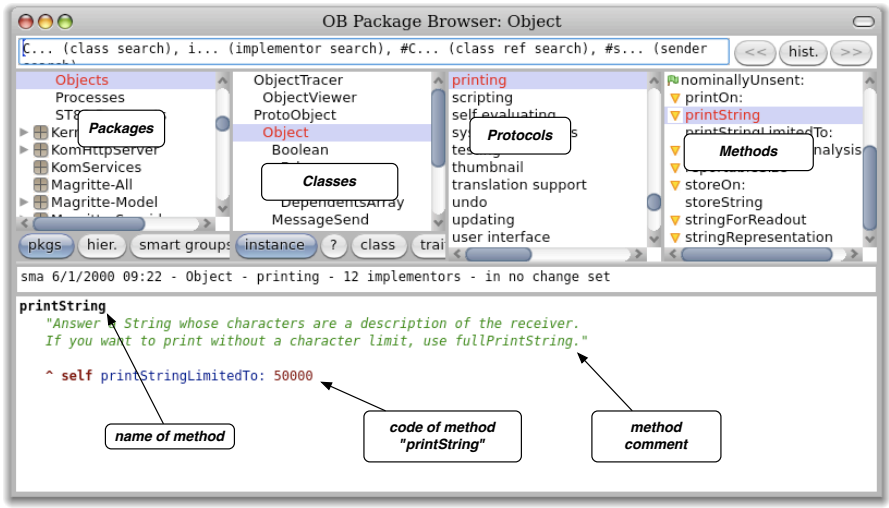


Figure 1.12: El navegador mostrando el método printString de la clase object.

Selecciona la clase Object.

Ahora los dos paneles restantes se completarán con texto. El tercer panel muestra los *protocolos* de la clase actualmente seleccionada. Estos son agrupaciones convenientes de métodos relacionados. Si ningún protocolo es seleccionado deberías ver todos los métodos en el cuarto panel.

Selecciona el protocolo printing.

Quizás necesites desplazarte hacia abajo para encontrarlo. Verás en el cuarto panel sólo métodos relacionados con impresión.

Selecciona el método printString.

Vemos en el panel inferior el código fuente del método printString, compartido por todos los objetos en el sistema (excepto por aquellos que los sobreescriben).

1.8 Encontrando clases

Hay varias formas de encontrar una clase en Pharo. La primera, como acabamos de ver, es conocer (o adivinar) en qué categoría está, y navegar hacia ella usando el navegador.

Una segunda forma es enviar el mensaje browse a la clase, pidiéndole que abra un navegador en ella misma. Supongamos que queremos navegar la

clase Boolean.

Escribe Boolean browse *dentro de un workspace y* do it.

Se abrirá un navegador con la clase Boolean (Figure 1.13). Hay también un atajo de teclado CMD−b (browse) que puedes usar en cualquier herramienta donde encuentres un nombre de clase; Selecciona el nombre y escribe CMD−b.

Usa el atajo de teclado para navegar la clase Boolean.

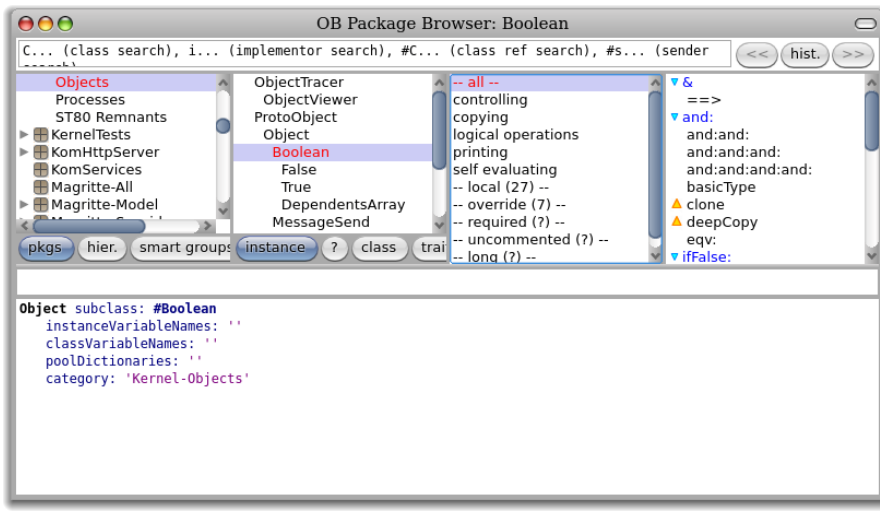


Figure 1.13: El navegador mostrando la definición de la clase Boolean.

Cuando la clase Boolean es seleccionada pero ningún protocolo o método es seleccionado, en lugar del código fuente de un método, vemos una *definición de clase* (Figure 1.13). Esto no es mas que un mensaje Smalltalk ordinario que es enviado a la clase padre, pidiéndolea crear una subclase. Aquí vemos que a la clase Object se le pide crear una subclase llamada Boolean sin variables de instancia, variables de clase o "pool dictionaries", y poner la clase Boolean en la categoría *Kernel-Objects*. Si haces click en el ? al pie del panel de clase, puedes ver los comentarios de la e n un panel dedicado (ver Figure 1.14).

A menudo, la forma mas rápida de encontrar una clase es buscarla por nombre. Por ejemplo, supongamos que estás buscando una clase que represente fechas y horas.

Pon el ratón en el panel de paquete del navegador y escribe CMD−*f, o selecciona* find class . . . (f) *haciendo action-click. Escribe "time" en el cuadro de diálogo y acéptalo.*

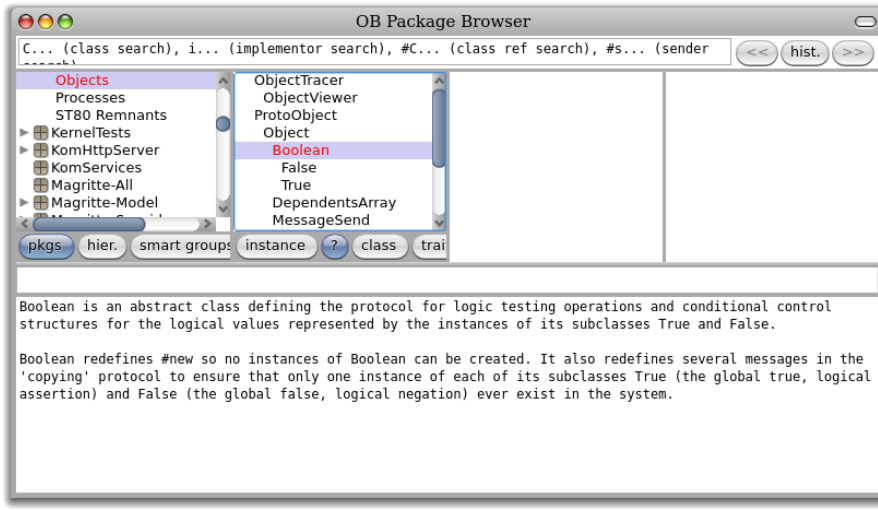


Figure 1.14: Los comentarios de la clase Boolean.

Se presentará una lista de clases cuyos nombre contienen "time" (ver Figure 1.15). Elige una, digamos, Time, y el navegador la mostrará, junto con comentario de clase que sugiere otras clases que pueden ser útiles. Si quieres navegar una de las otras, selecciona su nombre (en cualquier panel), y escribe CMD−b.

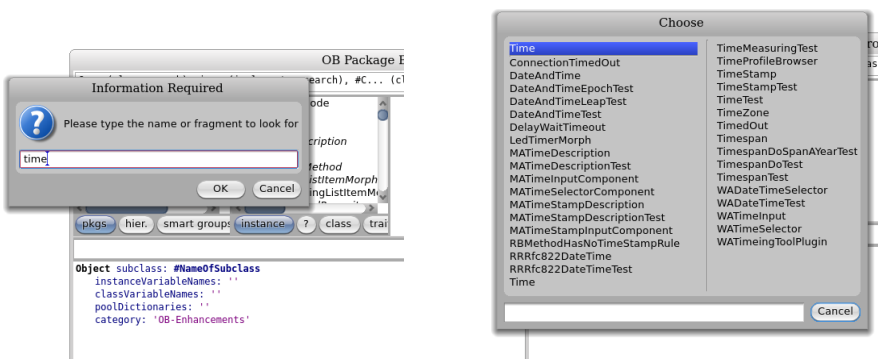


Figure 1.15: Buscando una clase por nombre.

Si escribes el nombre completo (respetando mayúsculas y minúsculas) de una clase en el cuadro de diálogo de búsqueda, el navegador irá directamente a esa clase sin mostrarle la lista de opciones.

1.9 Encontrando métodos

Algunas veces puedes adivinar el nombre de un método, o al menos parte del nombre de un método, más facilmente que el nombre de una clase. Por ejemplo, si estás interesado en la hora actual, quizás esperas que haya un método llamado "now", o conteniendo "now" como parte de su nombre. Pero ¿dónde podría estar? El *buscador de métodos* puede ayudarte.

Selecciona World ▷ Tools ... ▷ Method finder. *Escribe "now" en el panel superior izquierdo, y luego* accept *(o simplemente presiona la tecla* RETURN*).*

El buscador de métodos mostrará una lista de todos los nombres de los métodos que contienen la palabra "now". Para desplazar hasta now, mueve el cursor en la lista y escribe "n"; este truco funciona en todas las ventanas con desplazamiento. Selecciona "now" y el panel derecho mostrará las clases que definen un método con este nombre, como se muestra en Figure 1.16. Seleccionando cualquiera de ellas se abrirá un navegador en ella.

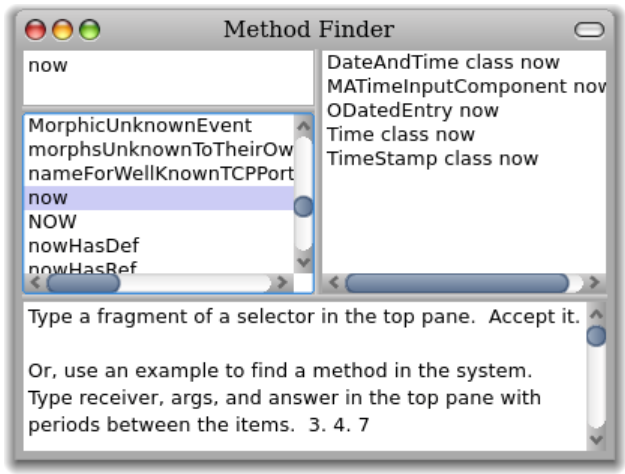


Figure 1.16: El buscador de métodos mostrando todas las clases que definen un método llamado now.

Otras veces quizás tienes idea que un método existe, pero no tendrás idea cuál es su nombre. ¡El buscador de métodos aún puede ayudar! Por ejemplo, supongamos que te gustaría encontrar un método que transforme una cadena de caracteres en mayúsculas, por ejemplo, traduciría 'eureka' en 'EUREKA'.

Escribe 'eureka' . 'EUREKA' *en el buscador de métodos y presiona la tecla* RETURN*, como se muestra en Figure 1.17.*

El buscador de métodos sugerirá un método que hace lo que quieres.[8]

Un asterico al comienzo de una linea en el panel derecho del buscador de métodos indica que este método es uno de los que fué realmente usado para obtener el resultado solicitado. El asterisco delante de String asUppercase nos permite saber que el método asUppercase definido en la clase String fue ejecutado y retornó el resultado buscado. Los métodos que no tiene un asterisco son los otros métodos que tiene el mismo nombre que los métodos que retornaron el resultado esperado. Character»asUppercase no fue ejecutado en nuestro ejemplo, porque 'eureka' no es un objeto Character.

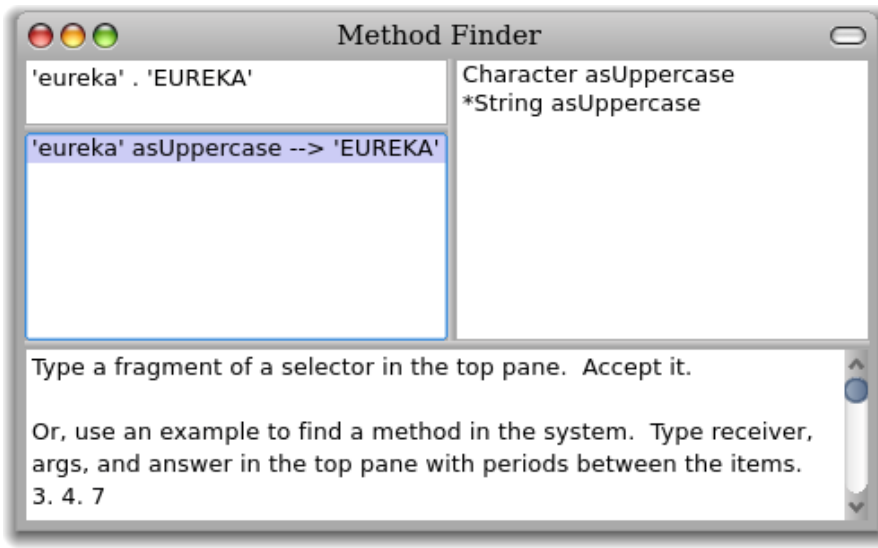


Figure 1.17: Encontrando un método mediante un ejemplo.

Puedes también usar el buscador de métodos con argumentos; por ejemplo, si estás buscando un método que encuentre el máximo factor común entre dos enteros, puedes probar 25. 35. 5 como ejemplo. Puedes darle al buscador de métodos múltiples ejemplos para reducir el espacio de búsqueda; el texto de ayuda en el panel inferior explica cómo.

[8]Si una ventana es desplegada con una advertencia acerca de un método deprecado, no tengas miedo — el buscador de métodos está simplemente probando todos los posibles candidatos, incluyendo métodos deprecados. Simplemente haz click en el botón Proceed.

1.10 Definiendo un nuevo método

El advenimiento de Test Driven Development[9] (TDD) ha cambiado la forma que escribimos código. La idea detrás de TDD es que escribimos una prueba que define el comportamiento deseado de nuestro código *antes* de escribir el propio código. Sólo entonces escribimos el código que satisface la prueba.

Supongamos que nuestra tarea es ecribir un método que "diga algo fuerte y con énfasis". ¿Qué podría significar esto exactamente? ¿Cuál sería un buen nombre para el método? ¿Cómo podemos estar seguros que los programadores quienes quizás tengan que mantener nuestro método en el futuro tendrán una descripción unívoca de lo que el método debería hacer? Podemos responder todas estas preguntas dando un ejemplo:

> Cuando enviamos el mensaje shout a la cadena de caracteres "Don't panic" el resultado debería ser "DON'T PANIC!".

Para hacer de este ejemplo algo que el sistema pueda ejecutar, lo ponemos dentro de un método de prueba:

Method 1.1: *Una prueba para un método gritar*

```
testShout
  self assert: ('Don''t panic' shout = 'DON''T PANIC!')
```

¿Cómo creamos un nuevo método en Pharo? Primero, tenemos que decidir a qué clase debería pertenecer el método. En este caso, el método shout que estamos probando irá en la clase String, por lo que la prueba correspondiente irá, por convención, en una clase llamada StringTest.

Abre un navegador en la clase StringTest, *y selecciona el protocolo apropiado para nuestro método, en este caso* tests - converting, *como se muestra en Figure 1.18. El texto seleccionado en el panel inferior es una plantilla que te recuerda cómo luce un método Smalltalk. Borralo y escribe el código del método method 1.1.*

Una vez que hayas escrito el texto en el navegador, notarás que el panel inferior se torna rojo. Esto es un recordatorio de que el panel contiene cambios sin guardar. Selecciona accept (s) haciendo action-click en el panel inferior, o simpelmente escribe CMD−s, para compilar y guardar el método.

Si esta es la primera vez que haz guardado código en tu imagen, probablemente te pedirá que ingreses tu nombre. Dado que mucha gente ha contribuido con código a la imagen, es importante mantener un seguimiento de

[9] Kent Beck, *Test Driven Development: By Example*. Addison-Wesley, 2003, ISBN 0–321–14653–0.

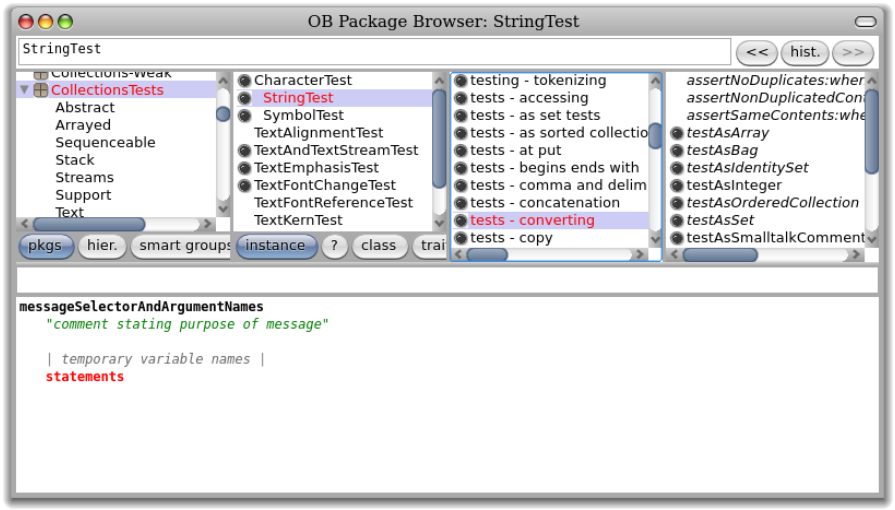


Figure 1.18: El nuevo método en la clase StringTest.

cualquiera que cree o modifique métodos. Simplemente ingresa tu nombre y apellido, sin espacios, o separados por un punto.

Debido a que no existe todavía ningún método llamado shout, el navegador te pedirá confirmar que este es el nombre que realmente quieres — y te sugerirá algunos otros nombres que podrías haber elegido (Figure 1.20). Esto puede ser bastante útil si haz cometido un error al escribir, realmente *queremos* decir shout, dado que es el método que estamos a punto de crear, por lo que tenemos que confirmar esto seleccionando la primera opción del menú de opciones, como se ve en Figure 1.20.

Ejecuta la prueba creada: abre el SUnit TestRunner *desde el menú* World.

Los paneles más a la izquierda son similares a los paneles superiores en el navegador. El panel izquierdo contiene una lista de categorías, pero estas están restringidas a aquellas categorías que contienen clases de prueba.

Selecciona CollectionsTests-Text *y el panel a la derecha mostrará todos las clases de prueba en la categoría, las cuales incluyen la clase* StringTest. *Los nombres de las clases ya están seleccionados, por lo que haz click en* Run Selected *para ejecutar todas estas pruebas.*

Deberías ver un mensaje como el mostrado en Figure 1.21, el cual indica que hubo un error en la ejecución de las pruebas. La lista de pruebas que dieron lugar a los errores se muestra en el panel inferior derecho; como puedes ver, StringTest»#testShout es el culpable. (Notar que StringTest>>#testShout es la manera en Smalltalk de identificar el método testShout de la

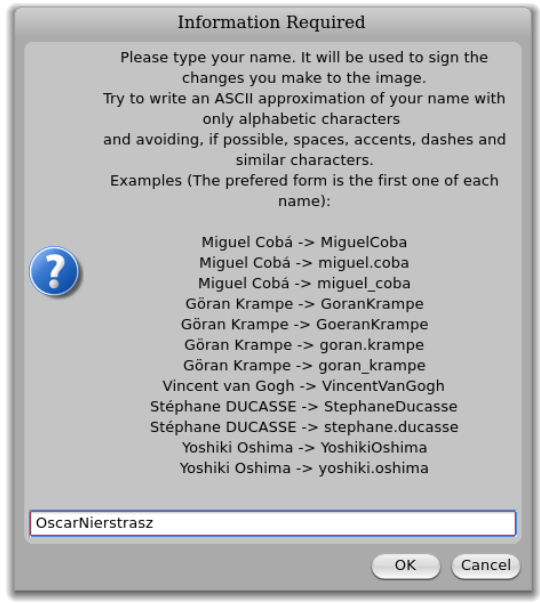


Figure 1.19: Ingresando tu nombre.

clase StringTest.) Si haces click en esa linea de texto, la prueba errónea se ejecutará de nuevo, esta vez de tal manera que se ve el error que ocurre: "MessageNotUnderstood: ByteString»shout".

La ventana que se abre con el mensaje de error es el depurador de Smalltalk (ver Figure 1.22). Miraremos el debugger y cómo usarlo en Chapter 6.

El error es, por supuesto, exactamente lo que esperabamos: ejecutar la prueba genera un error porque aún no hemos escrito un método que le diga a las cadenas de caracteres como shout. Sin embargo, es una buena práctica estar seguros que la prueba falla porque confirma que tenemos configurada la maquinaria de pruebas correctamente y que la nueva prueba está realmente siendo ejecutada. Una vez que haz visto el error, puedes dejar la prueba ejecutándose haciendo clic en Abandon, lo cual cerrará la ventana de depuración. Notar que a menudo con Smalltalk puedes definir el método no definido usando el botón Create, editar el método recientemente creado en el depurador, y ejecutar la prueba haciendo clic en Proceed.

¡Ahora definamos un método que hará que la prueba sea exitosa!

Selecciona la clase String *en el navegador, selecciona el protocolo* converting*, escribe el texto method 1.2 sobre la plantilla de creación de método, y selecciona* accept*. (Nota: para obtener ↑, escribe ^).*

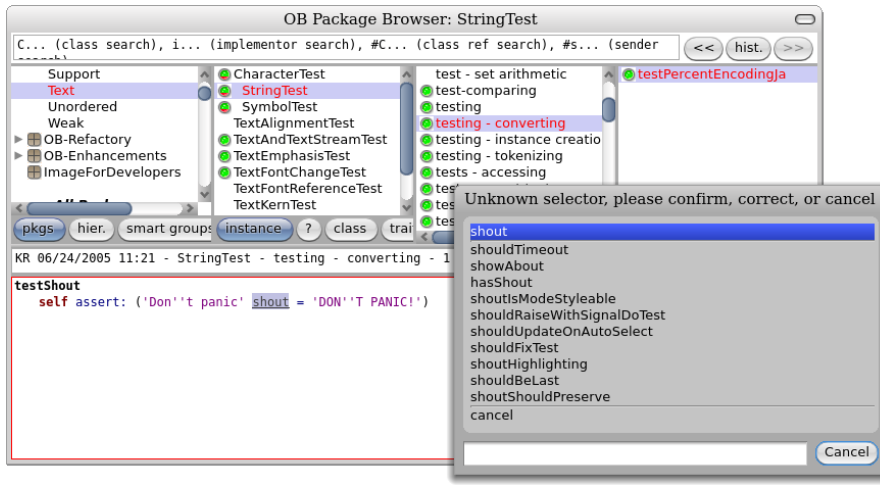


Figure 1.20: Aceptando el método testShout en la clase StringTest.

Method 1.2: *El método shout*

```
shout
  ↑ self asUppercase, '!'
```

La coma es la operación de concatenación de cadenas de caracteres, por lo que el cuerpo del método agrega un signo de exclamación a una versión en mayúsculas de lo que sea el objeto String al cual se le envió el mensaje shout. El ↑ dice a Pharo que la expresión que sigue es la respuesta a ser retornada desde el método, en este caso la nueva cadena de caracteres concatenada.

¿Este método funciona? Ejecutemos la prueba y veamos.

Haz clic en Run Selected *nuevamente en el test runner, y esta vez deberías ver una barra verde y un texto indicando que todas las pruebas se ejecutaron sin fallas ni errores.*

Cuando obtienes una barra verde[10], es una buena idea guardar tu trabajo y tomar un descanso. ¡Asi que hazlo ahora mismo!

1.11 Resumen del capítulo

Este capítulo te ha introducido en el ambiente de Pharo y te ha mostrado cómo usar la mayoría de las herramientas, como el navegador, el buscador de métodos, y el test runner. Haz visto también un poco de la sintaxis de Pharo, aunque puede que no la entiendas completamente aún.

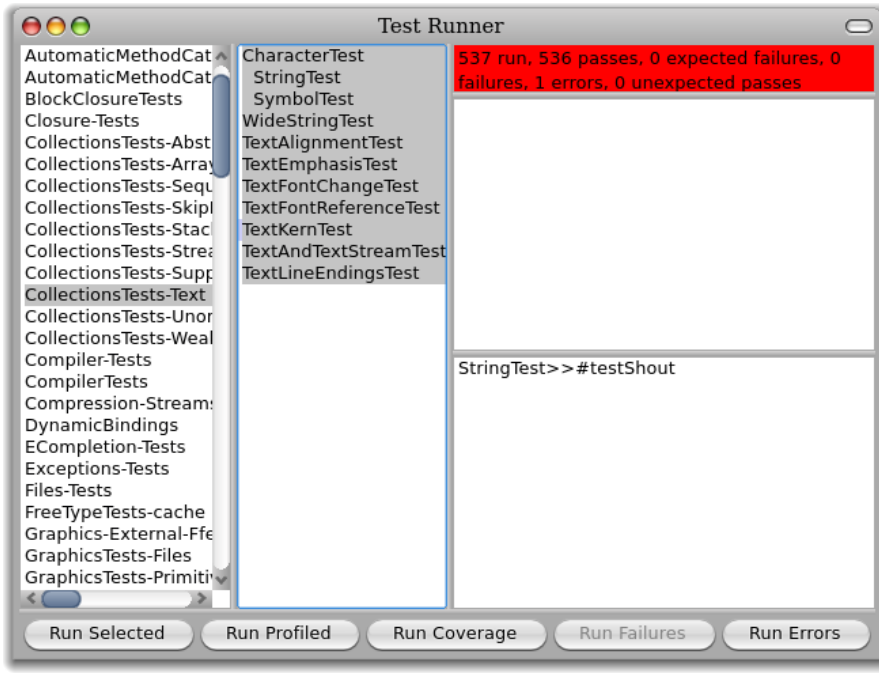


Figure 1.21: Ejecutando las pruebas de String.

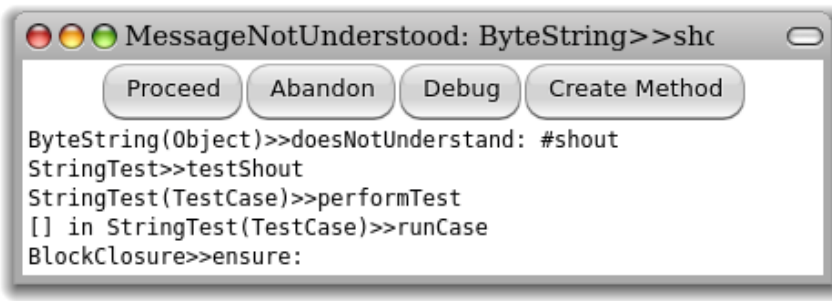


Figure 1.22: El (pre-)depurador.

- Un sistema Pharo ejecutando consiste en una *máquina virtual*, un archivo *sources*, y unos archivos *image* y *changes*. Sólo estos dos últimos cambian, grabando una instantanea del sistema ejecutando.

- Cuando recuperes una imagen de Pharo, te encontrarás en exactamente el mismo estado — con los mismos objetos ejecutando — que

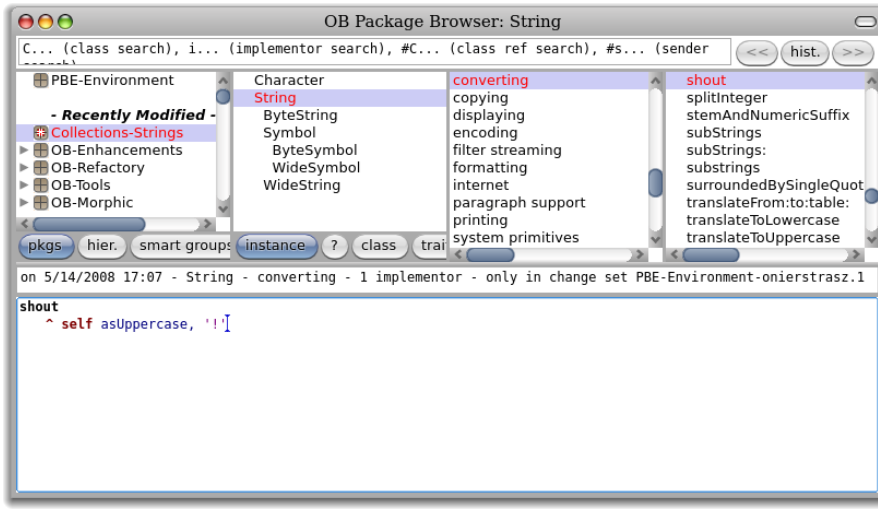


Figure 1.23: El método shout definido en la clase String.

tenías cuando guardaste por última vez la imagen.

- Pharo está diseñado para trabajar con un ratón de tres botones para hacer click, hacer action-click o hacer meta-click. Si no tienes un ratón de tres botones, puede usar las teclas modificadoras para obtener el mismo efecto.
- Puedes hacer click en el fondo de la ventana de Pharo para abrir el *menú World* y lanzar varias herramientas.
- Un *workspace* es una herramienta para escribir y evaluar porciones de código. También puedes usarlo para almacenar texto arbitrario.
- Puedes usar atajos de teclado en el texto del workspace, o en cualquier otra herramienta, para evaluar código. Los más importante de estos son do it (CMD–d), print it (CMD–p), inspect it (CMD–i), explore it (CMD–I) y browse it (CMD–b).
- El *navegador* es la herramienta principal para navegar código Pharo, y para desarrollar nuevo código.
- El *test runner* es una herramienta para ejecutar pruebas unitarias. También soporta Desarrollo Dirigido por Pruebas.

Chapter 2

Una primer aplicación

En este capítulo vamos a desarrollar un juego simple: Lights Out.[1] En el camino mostraremos la mayoría de las herramientas que los programadores de Pharo usan para construir y depurar sus programas, y mostraremos como se intercambian los programas con otros desarrolladores. Veremos el explorador, el inspector de objetos, el depurador y el paquete explorador Monticello. El desarrollo en Smalltalk es eficiente: encontrarás que pasas más tiempo escribiendo código y mucho menos manejando el proceso de desarrollo. Esto es en parte porque Smalltalk es muy simple, y en parte porque las herramientas que conforman el entorno de programación están muy bien integradas con el lenguaje.

2.1 El juego Lights Out

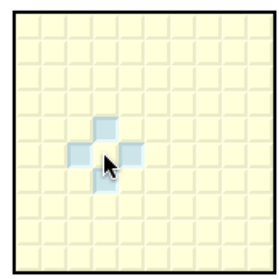


Figure 2.1: El tablero del juego Lights Out. El usuario ha hecho clic con el ratón como muestra el cursor.

Para mostrar como utilizar las herramientas de Pharo, construiremos un un juego sencillo llamado *Lights Out*. El tablero del juego se muestra en

[1] http://en.wikipedia.org/wiki/Lights_Out_(game)

la Figure 2.1; consiste en una grilla rectagular de *celdas* amarillas. Cuando haces clic sobre una celda con el ratón, las cuatro celdas adyacentes se tornan azules. Si haces Click otra vez, se volverán otras vez amarillas. El objetivo del juego es poner azules la mayor cantidad de celdas que sea posible.

En el Lights Out que se muesta en Figure 2.1 está compuesto por dos objetos: el tablero del juego en sí, y 100 objetos celda individuales. El código de Pharo para implementar el juego tendrá dos clases: una para el juego y otra para las celdas. Ahora veremos cómo definir estas clases usando las herramientas de programación de Pharo.

2.2 Creando un paquete nuevo

Ya hemos visto el Browser en Chapter 1, donde aprendimos como navegar clases y métodos, y vimos como definir nuevos métodos. Ahora veremos como crear paquetes, categorías y clases.

Abre el explorador y selecciona la herramienta package. Selecciona create package.[2]

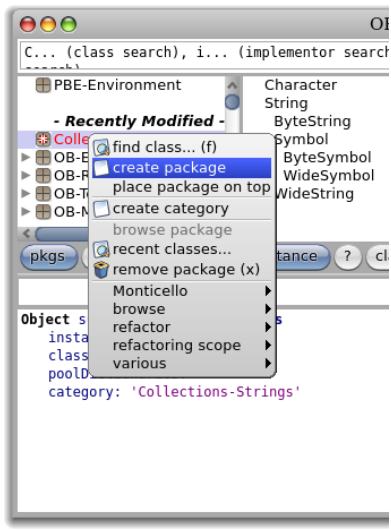


Figure 2.2: Agregando un paquete.

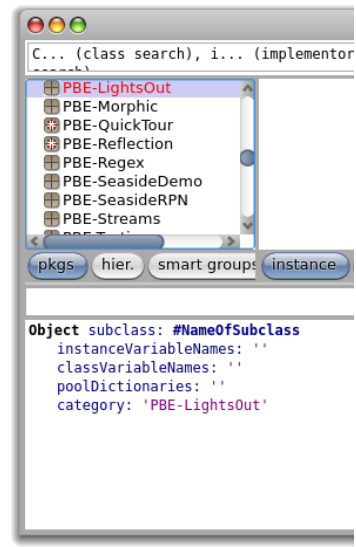


Figure 2.3: La plantilla de una clase.

Escribe el nombre del nuevo paquete (usaremos *PBE-LightsOut*) en el

[2]Estamos asumiendo que el explorador de paquetes está instalado como el explorador por defecto, lo que ocurre en casi todos los casos. Si el navagador que obtienes no se ve similar al de Figure 2.2, entonces tal vez tengas que cambiar el explorador por defecto. Mira FAQ 5, p. 332.

cuadro de diálogo y selecciona accept (o solo pulsa la tecla enter); el nuevo paquete se ha creado, y se posiciona alfabéticamente en la lista de paquetes.

2.3 Definiendo la clase LOCell

Por supuesto, todavía no hay clases en el nuevo paquete. Aún así, el panel principal de edición nos muestra una plantilla para hacer más sencilla la tarea de crear una nueva clase (ver Figure 2.3).

Esta plantilla nos muestra una expresión Smalltalk que envía un mensaje a la clase que se llama Object, solicitándole que genere una subclase llamada NameOfSubClass. La nueva clase no tendrá variables, y deberá pertenecer a la categoría *PBE-LightsOut*.

Sobre categorías y paquetes

Históricamente, se trata sobre *categorías*, no paquetes. Entonces podrías preguntarte, ¿cual es la diferencia? Una categoría es simplemente una colección de clases relacionanadas en una imágen Smalltalk. Un *paquete* es una colección de clases relacionadas junto a *métodos de extensión* que podrían estar bajo un control de versiones como Monticello. Por convención, los nombres de paquetes y categorías son siempre los mismos. En la mayoría de los casos no nos preocuparemos por las diferencias, pero seremos cuidadosos en la terminología de este libro para utilizar la denominación correcta, ya que hay puntos donde de la diferencia es crucial. Aprenderemos más sobre este tema cuando comencemos a trabajar con Monticello.

Creando una nueva clase

Solamente modificaremos la plantilla para crear la clase que realmente queremos.

 *Modifique la plantilla de creación de clase como sigue:*

- Reemplace Object por SimpleSwitchMorph.
- Reemplace NameOfSubClass por LOCell.
- Añada mouseAction a la lista de variables de instancia.

El resultado debería parecerse a class 2.1.

Class 2.1: *Definiendo la clase* LOCell

```
SimpleSwitchMorph subclass: #LOCell
  instanceVariableNames: 'mouseAction'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−LightsOut'
```

Esta nueva definición consiste de una expresión Smalltalk que envía un mensaje a la clase existente SimpleSwitchMorph, pidiéndole que cree una subclase llamada LOCell. (En realidad, como LOCell no existe aún, se le pasa *symbol* #LOCell como argumento, que indica el nombre de la clase a crear). También le indicamos que las instancias de la nueva clase deberían tener una instancia de variable de mouseAction, que usaremos para definir que acción debería tomar la celda si se hiciera clic sobre esta.

Hasta el momento no has creado nada. Nota que el borde del panel de plantilla de clase cambió a rojo (Figure 2.4). Esto significa que hay cambios *sin guardar.* Para enviar este mensaje, debes seleccionar `accept`.

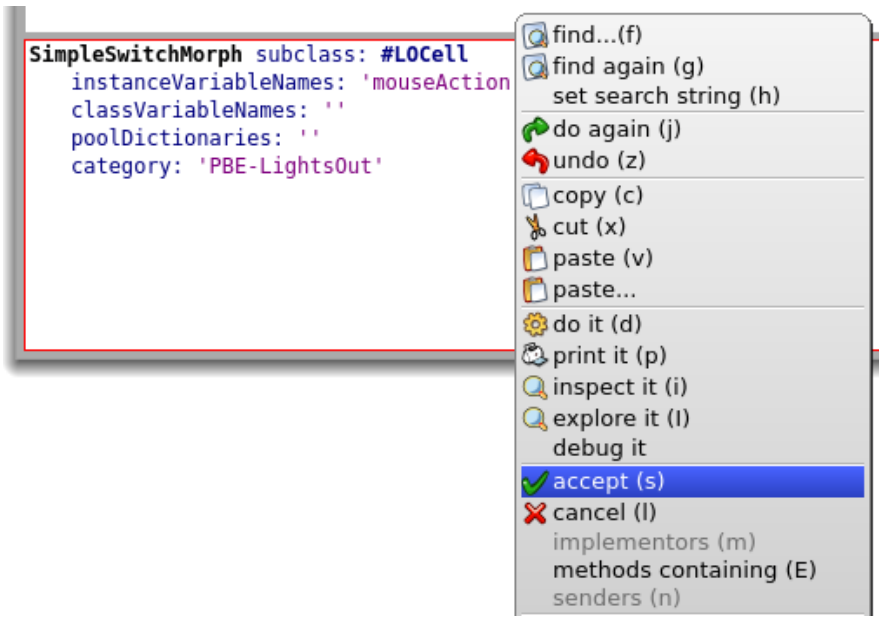


Figure 2.4: Template de creación de clase.

Aceptar la definición de la nueva clase.

Esto se puede hacer de dos formas: o bien action-click y selecciona `accept`, o bien usa el atajo CMD−s (para "guardar"). El mensaje será enviado a SimpleSwitchMorph, causando que la nueva clase sea compilada.

Una vez que la definición de la clase es aceptada, la clase será creada y aparecerá en el panel de clases del navegador (Figure 2.5). Ahora el panel de edición muestra la definición de la clase, y un pequeño panel debajo te recordará escribir unas pocas palabras describiendo el propósito de la clase. Esto se llama *comentario de la clase*. Es importante escribir un comentario que le de a otros programadores una visión de alto nivel del propósito de la clase. Los desarrolladores Smalltalk le dan un gran valor a la legibilidad de su código, y es poco común encontrar comentarios detallados dentro de los métodos: la filosofía de Smalltalk es que el código debe ser autoexplicativo. (Si no lo es, ¡deberías refactorizarlo hasta que lo sea!) Una clase n o debería contener una descripción detallada de la clase, pero unas pocas palabras que describan su propósito de manera general son vitales para que otros programadores sepan si vale la pena invertir tiempo en mirar la clase o no.

Escribe un comentario de clase para LOCell *y guárdalo; siempre se puede mejorar más tarde.*

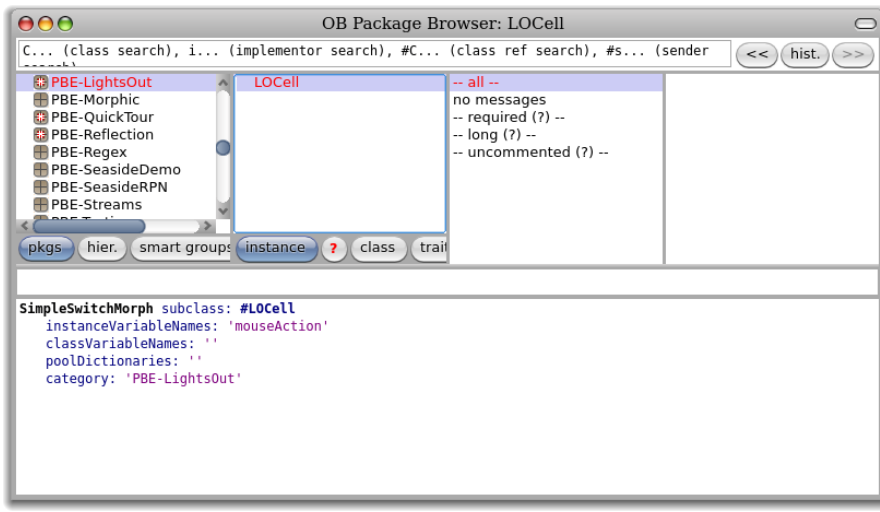


Figure 2.5: La nueva clase creada LOCell

2.4 Agregando métodos a una clase

Ahora agreguemos algunos métodos a nuestra clase.

Selecciona el protocolo --all-- *en el panel de protocolos.*

Verás un plantilla para la creación de métodos en el panel de edición. Selecciónalo y reemplázalo por el texto del method 2.2.

Method 2.2: *Initializing instances of* LOCell

```
1 initialize
2   super initialize.
3   self label: ''.
4   self borderWidth: 2.
5   bounds := 0@0 corner: 16@16.
6   offColor := Color paleYellow.
7   onColor := Color paleBlue darker.
8   self useSquareCorners.
9   self turnOff
```

Notar que los caracteres '' en la línea 3 son comillas simples sin nada en el medio, no una doble comilla. '' es una cadena de caracteres vacía.

 Aceptar *la definición de este método.*

¿Qué hace el código de arriba? No nos adentraremos en los detalles aquí (¡para eso es el resto del libro!), pero te daremos una breve introducción. Veámoslo línea por línea.

Notar que el método se llama initialize. ¡El nombre es muy significativo! Por convención, si una clase define un método llamado initialize, este será llamado inmediatamente después que el objeto sea creado. Entonces, cuando evaluamos LOCell new, el mensaje initialize será enviado automáticamente al objeto recién creado. Los métodos "initialize" se usan para establecer el estado de los objetos, normalmente para establecer las variables de instancia; esto es exactamente lo que estamos haciendo acá.

Lo primero que hace este método (línea 2) es ejecutar el método initialize de su superclase, SimpleSwitchMorph. El objetivo de esto es que cualquier estado heredado sea inicializado adecuadamente por el método initialize de la superclase. Siempre es buena idea inicializar estado heredado enviando el mensaje super initialize antes de hacer cualquier otra cosa; no sabemos exactamente que hará el método initialize de SimpleSwitchMorph, y no nos importa, pero seguramente inicializará algunas variables de instancia para que contengan valores razonables, por lo que es mejor invocarlo para no arriesgarse a que los objetos empiecen en un estado inválido.

El resto del método establece el estado del objeto. Enviar el mensaje self label: '', por ejemplo, establece la etiqueta de este objeto a la cadena de caracteres vacía.

La expresión 0@0 corner: 16@16 probablemente necesite un poco de explicación. 0@0 representa un objeto Point con las coordenadas x e y en 0. De hecho, 0@0 envía el mensaje @ al número 0 con parámetro 0.

El efecto será que el número 0 le pedirá a la clase Point que cree una nueva instancia con coordenadas (0,0). Ahora le enviamos a este nuevo punto el mensaje corner: 16@16, que hace que cree un Rectangle con esquinas 0@0 y 16

@16. Este nuevo rectángulo será asignado a la variable bounds, heredada de la superclase.

Nota que el orígen de la pantalla de Pharo es la *esquina izquierda*, y la coordenada *y* se incrementa hacia *abajo*.

El resto del método debería ser autoexplicativo. Parte de arte de escribir buen código Smalltalk es escoger los nombres adecuados para los métodos para que el código pueda ser leido como una especie de inglés macarrónico. Deberías poder imaginar al objeto hablandose a sí mismo y diciendo "Self use square corners!", "Self turn off".

2.5 Inspeccionando un objeto

Puedes probar el efecto del código que acabas de escribir creando un nuevo objeto LOCell y luego inspeccionándolo.

Abre un workspace. Tipea la expresión LOCell new *y* inspect it.

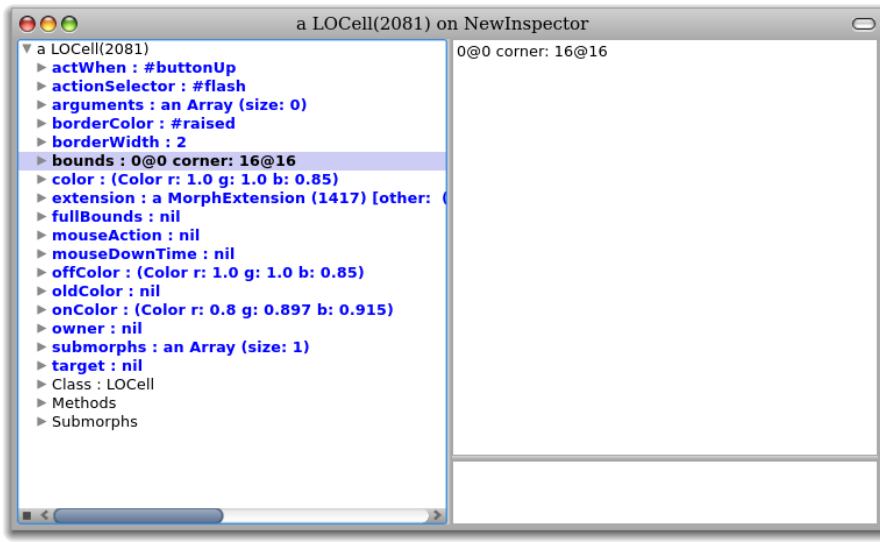


Figure 2.6: El inspector usado para examinar un objeto LOCell.

El panel izquierdo de inpector muestra una lista de variables de instancia; si seleccionas una (prueba bounds), el valor de la variable de instancia se muestra en el panel derecho.

El panel inferior del inspector es un mini-workspace. Es muy útil porque en este workspace la pseudo variable self está ligada al objeto seleccionado.

Selecciona LOCell en la raíz de la ventana del inspector. Tipea el texto `self openInWorld` *en el mini-workspace y* `do it`.

La celda debería aparecer cerca de la esquina superior izquierda de la pantalla, exactamente donde sus límites dicen que debería aparecer. meta-click en la celda para que aparezca el halo de Morphic . Mueve la celda con el asa marrón (esquina superior derecha) y redimensiónalo con el manejador amarilla (abajo a la derecha). Nota como los límites reportados por el inspector también cambian. (tal vez tengas que action-click `refresh` para ver los nuevos valores de los límites.)

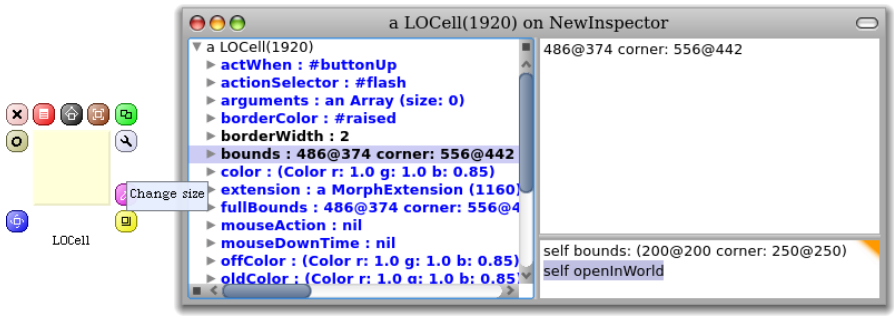


Figure 2.7: Redimensiona la celda.

Borra la celda haciendo clic en la x *en el asa rosa.*

2.6 Definiendo la clase LOGame

Ahora creemos la otra clase que necesitamos para el juego, que llamaremos LOGame.

Has visible la plantilla de definición de clases en la ventana principal del navegador.

Has esto haciendo clic en el nombre del paquete. Edita el código para que quede como se muestra a continuación, y has `accept`.

Class 2.3: *Defining the* LOGame *class*

```
BorderedMorph subclass: #LOGame
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−LightsOut'
```

Aquí creamos una subclase de BorderedMorph; Morph es la superclase de todas las formas gráficas de Pharo, y(¡sorpresa!) un BorderedMorph es un Morph con borde. Podríamos insertar los nombres de las variables de instancia entre comillas en la segunda línea pero, por ahora, dejemos esa lista vacía.

Ahora definamos un método initialize para LOGame.

Escribe lo siguiente en el navegador como un método para LOGame *e intenta* accept*:*

Method 2.4: *Initializing the game*

```
initialize
  | sampleCell width height n |
  super initialize.
  n := self cellsPerSide.
  sampleCell := LOCell new.
  width := sampleCell width.
  height := sampleCell height.
  self bounds: (5@5 extent: ((width*n) @(height*n)) + (2 * self borderWidth)).
  cells := Matrix new: n tabulate: [ :i :j | self newCellAt: i at: j ].
```

Pharo se quejará de que no conoce el significado de ciertos términos. Pharo dice que no conoce un mensaje cellsPerSide, y sugiere algunas correcciones, en caso de que haya sido un error de tipeo.

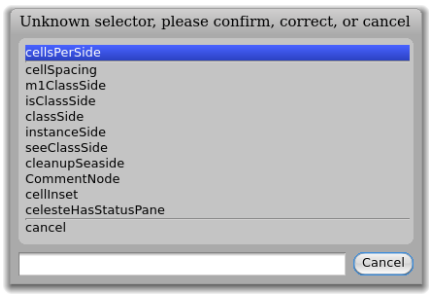


Figure 2.8: Pharo detectando un identificador desconocido.

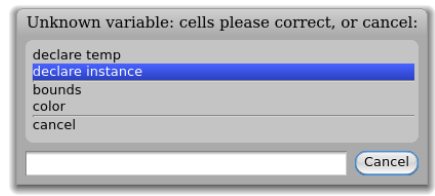


Figure 2.9: Declarando una nueva variable de instancia.

Pero cellsPerSide no es un error — es solo un método que no hemos definido aun — lo haremos en un momento.

Selecciona el primer item del menú, que confirma que quisimos escribir cellsPerSide.

A continuación, Pharo se quejará de que no conoce el significado de cells. Te ofrece un número de formas de arreglar esto.

 Elige declare instance *porque queremos que* cells *sea una variable de instancia.*

Finalmente, Pharo se quejará del mensaje newCellAt:at: en la última línea; esto tampoco es un error, asi que debes confirmar ese mensaje también.

Si miras ahora a la definición de la clase de nuevo (cosa que puedes hacer haciendo clic en el botón nstance), verás que el navegador la modificó para incluir la variable de instancia cells.

Miremos ahora al método initialize. La línea | sampleCell width height n | declara 4 variables temporales. Son llamadas temporales porque su ámbito y tiempo de vida son limitados a este método. Las variables temporales con nombres explicativos ayudan a hacer el código más legible. Smalltalk no tiene sintaxis especial para distinguir constantes y variables, y en realidad estas cuatro "variables" son en realidad constantes. Las líneas 4–7 definen estas constantes.

¿Qué tan grande debería ser nuestro tablero de juego? Lo suficientemente grande como para almacenar algun número entero de celdas, y lo suficientemente grande como para dibujar un borde alrededor de ellas. ¿Cuál sería el número correcto de celdas? ¿5? ¿10? ¿100? No sabemos aún, y si supiéramos, probablemente cambiaríamos de parecer luego. Así que delegamos la responsabilidad de saber ese número a otro método, que llamaremos cellsPerSide, y que escribiremos en un momento. Es porque estamos enviando el mensaje cellsPerSide antes de definirlo que Pharo nos pidió "confirmar, corregir o cancelar" cuando aceptamos el cuerpo del método initialize. No te desanimes por esto: es en realidad una buena práctica escribir en términos de métodos que no hemos definido aún. ¿Por que? No fué hasta que empezamos a escribir el método initialize que nos dimos cuenta que lo necesitábamos, y en ese momento, podemos darle un nombre significativo, y seguir adelante, sin interrumpir el flujo.

La cuarta línea usa este método: El código Smalltalk self cellsPerSide envía el mensaje cellsPerSide a self, es decir, a este mismo objeto. La respuesta, que será el número de celdas por lado del tablero de juego, es asignado a n.

Las siguientes tres líneas crean un nuevo objeto LOCell, y asignan su ancho y alto a las variables temporales adecuadas.

La línea 8 establece los límites del nuevo objeto. Sin preocuparte mucho por los detalles ahora, confía en que la expresión entre paréntesis crea un cuadrado con orígen (*i.e.*, esquina superior izquierda) en el punto (5,5) y su esquina inferior derecha lo suficientemente lejos como para dar espacio al número correcto de celdas.

La última línea establece la variable de instancia cells del objeto LOGame a una recién creada Matrix con el correcto número de filas y columnas. Hacemos esto enviando el mensaje new:tabulate: a la clase Matrix (las clases son objetos también, por lo que les podemos enviar mensajes). Sabemos que new:tabulate espera dos parámetros porque tiene dos símbolos : en su nombre.

Los parámetros van justo despues de los símbolos ':'. Si estás acostumbrado a lenguajes que ponen todos los argumentos juntos dentro de paréntesis, esto puede parecer raro al principio. ¡No entres en pánico, es solo sintaxis! Resulta que es una muy buena sintaxis porque el nombre de un método puede ser usado para explicar los roles de los argumentos. Por ejemplo, es bastante claro que `Matrix rows: 5 columns: 2` tiene 5 filas y 2 columnas, y no 2 filas y 5 columnas.

`Matrix new: n tabulate: [ :i :j | self newCellAt: i at: j ]` crea una nueva matriz de `n` $\times$`n` e inicializa sus elementos. El valor inicial de cada elemento dependerá de sus coordenadas. El (i,j)-ésimo elemento será inicializado al resultado de evaluar `self newCellAt: i at: j`.

2.7 Organizando métodos en protocolos

Antes de definir más métodos, demos una rápida mirada al tercer panel en la parte superior del navegador. De la misma manera que el primer panel del navegador nos permite categorizar clases en paquetes para que no nos sintamos abrumados por una larga lista de nombres de clases en el segundo panel, el tercer panel nos permite categorizar métodos para que no nos sintamos abrumados por una larga lista de nombres de métodos en el cuarto panel. Estas categorías de métodos se llaman "protocolos".

Si sólo hay unos pocos métodos en la clase, el nivel extra de jerarquía provisto por los protocolos no es realmente necesario. Esta es la razón por la que el navegador también nos ofrece el protocolo virtual *--all--*, que, como no te sorprenderá, contiene todos los métodos de la clase.

Si has seguido este ejemplo, el tercer panel bien podría contener el protocolo *as yet unclassified*.

 *Action-click en el panel de protocolo y selecciona* `various ▷ categorize automatically` *para arreglar esto, y mueve los métodos* `initialize` *a un nuevo protocolo llamado* `initialization`.

¿Como sabe Pharo que este es el protocolo correcto? Bueno, en general Pharo no puede saberlo, pero en este caso también hay un método `initialize` en una superclase, y Pharo asume que nuestro método `initialize` debería ir en la misma categoría que el que está sobreescribiendo.

Convención tipográfica. Los programadores Smalltalk usan con frecuencia la notación ">>" para identificar la clase a la que pertenece un método. Por ejemplo, el método `cellsPerSide` en la clase `LOGame` sería referenciado como `LOGame>>cellsPerSide`. Para indicar que esto *no es* sintaxis Smalltalk, usaremos en su lugar el símbolo especial », con lo que el método en el texto

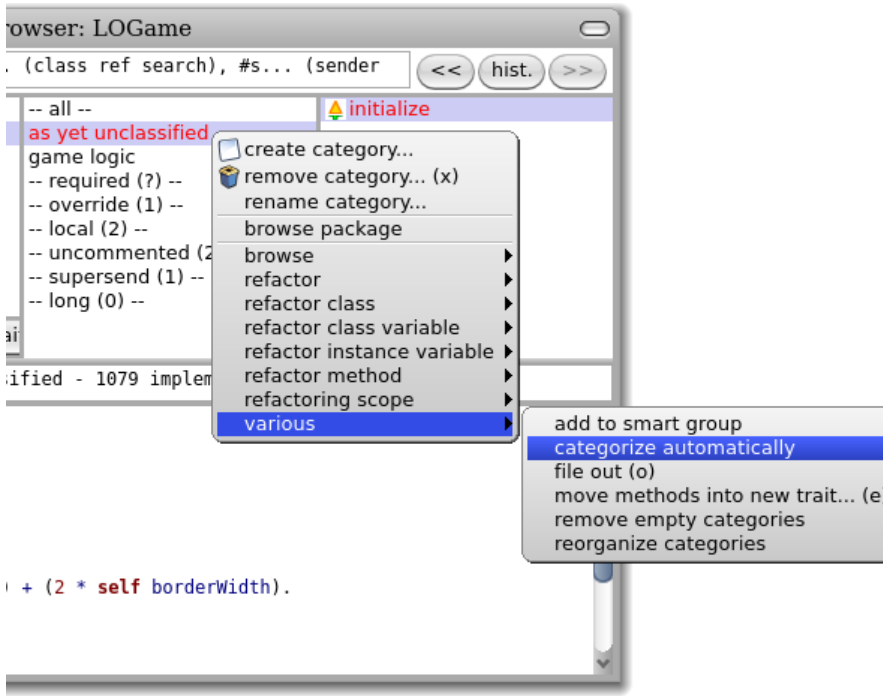


Figure 2.10: Catergoriza autimáticamente todos los métodos sin categorizar.

aparecerá como LOGame»cellsPerSide.

De ahora en adelante, cuando mostremos un método en este libro, escribiremos el nombre del método en esta forma. Por supuesto, cuando escribas el código en el navegador, no tendrás que escribir el nombre de la clase ni »; solo asegúrate de que la clase correcta está seleccionada en el panel de clases.

Ahora definamos los otros dos métodos que son usados por el método LOGame»initialize. Ambos pueden ir dentro del protocolo *initialization*.

Method 2.5: *Un método constante.*

```
LOGame»cellsPerSide
  "The number of cells along each side of the game"
  ↑10
```

Este método no podría ser más simple: responde con la constante 10. Una ventaja de representar constantes con métodos es que si el programa evoluciona de manera que la constante depende de otras características, el método puede ser cambiado para calcular el nuevo valor.

Method 2.6: *Un método de inicialización*

```
LOGame»newCellAt: i at: j
  "Create a cell for position (i,j) and add it to my on−screen
  representation at the appropriate screen position.  Answer the new cell"
  | c origin |
  c := LOCell new.
  origin := self innerBounds origin.
  self addMorph: c.
  c position: ((i − 1) * c width) @ ((j − 1) * c height) + origin.
  c mouseAction: [self toggleNeighboursOfCellAt: i at: j]
```

Agrega los métodos LOGame»cellsPerSide *y* LOGame»newCellAt:at:.

Confirma la pronunciación de los nuevos métodos toggleNeighboursOfCellAt:at: y mouseAction:.

Method 2.6 devuelve una nueva LOCell, especializada en la posición (i ,j) en la Matriz de celdas. La última línea define el mouseAction de la nueva celda como el *bloque* [self toggleNeighboursOfCellAt: i at: j]. En efecto, esto define el comportamiento callback cuando se hace clic con el ratón. El método correspondiente también tiene que ser definido.

Method 2.7: *The callback method*

```
LOGame»toggleNeighboursOfCellAt: i at: j
  (i > 1) ifTrue: [ (cells at: i − 1 at: j ) toggleState].
  (i < self cellsPerSide) ifTrue: [ (cells at: i + 1 at: j) toggleState].
  (j > 1) ifTrue: [ (cells at: i  at: j − 1) toggleState].
  (j < self cellsPerSide) ifTrue: [ (cells at: i at: j + 1) toggleState].
```

Method 2.7 cambia el estado de las cuatro celdas al norte, sur, este y oeste de una celda (i, j). La única complicación es que el tablero es finito, por lo que tenemos que asegurarnos que las celdas vecinas existen antes que cambiemos su estado.

Coloca este método en un nuevo protocolo llamado game logic. *(Action-click en el panel de protocolo para agregar un nuevo protocolo.)*

Para mover el método, puedes hacer clic en su nombre y arrastrarlo dentro del nuevo protocolo (Figure 2.11).

Para completar el juego Lights Out, necesitamos definir dos métodos más en la clase LOCell para manejar los eventos del ratón.

Method 2.8: *Un método setter típico*

```
LOCell»mouseAction: aBlock
  ↑ mouseAction := aBlock
```

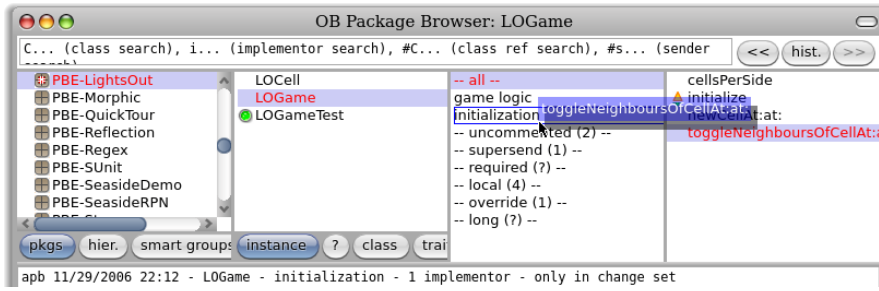


Figure 2.11: Arrastra un método a un protocolo.

Method 2.8 no hace nada más que establecer la variable mouseAction de la celda al argumento, y luego retorna el nuevo valor.

Cualquier método que *cambie* el valor de una variable de instancia de esta manera se llama *método setter*; un método que *retorna* el valor actual de una variable de instancia se llama *método getter*.

Si estás acostumbrado a los getters y setters en otros lenguajes, podrías esperar que estos métodos se llamen setmouseAction and getmouseAction. La convención Smalltalk es diferente. Un getter siempre tiene el mismo nombre que la variable que devuelve, y un setter siempre es llamado de manera similar, pero con un ":" al final, por lo que queda mouseAction and mouseAction:.

Colectivamente, los setters y getters son llamados métodos *de acceso*, y por convención deberían ser colocados en el protocolo *accessing*. En Smalltalk, *todas* las variables de instancia son privadas para el objeto que las posee, por lo que la única manera de leer o escribir esas variables para otro objeto en el lenguaje Smalltalk es a través de los métodos de acceso como este[3].

Ve a la clase LOCell, *define* LOCell»mouseAction: *y ponlo en el protocolo* accessing.

Finalmente, necesitamos definir un método mouseUp:; este será llamado automáticamente por el GUI framework si el botón del ratón es soltado cuando el ratón está sobre la celda en la pantalla.

Method 2.9: *Un manejador de eventos*

```
LOCell»mouseUp: anEvent
  mouseAction value
```

[3]En realidad, las variables de instancia también pueden ser accedidas desde las subclases.

Agrega el método LOCell»mouseUp: *y luego* categorize automatically *los métodos.*

Lo que hace este método es enviar el mensaje value al objeto almacenado en la variable de instancia mouseAction. Recuerda que en LOGame»newCellAt: i at: j asignamos el siguiente fragmento de código a mouseAction:

```
[self toggleNeighboursOfCellAt: i at: j ]
```

Enviar el mensaje value provoca que este fragmento de código sea evaluado, y consecuentemente el estado de las celdas cambiará.

2.8 Probemos nuestro código

Eso es: ¡el juego Lights Out está completo!

Si has seguido todos los pasos, deberías poder jugar el juego, que consiste en solo 2 clases y 7 métodos.

En un workspace, tipea LOGame new openInWorld *y* do it.

El juego se abrirá, y deberías poder hacer clic en las celdas y ver como funciona.

Bueno, demasiada teoría… Cuando haces clic en una celda, una ventana de *notificación* llamada ventana PreDebugWindow aparece con un mensaje de ¡error! Como se muestra en Figure 2.12, dice MessageNotUnderstood: LOGame» toggleState.

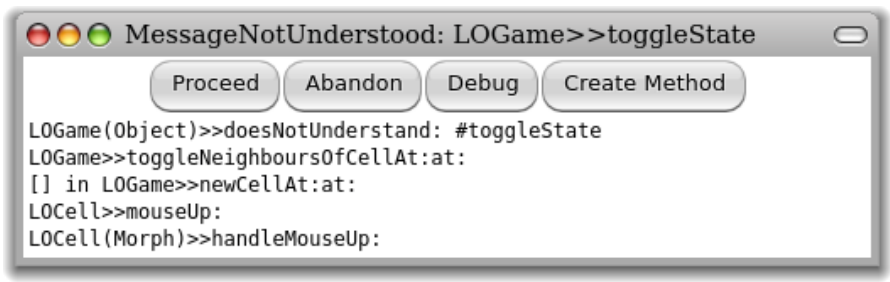


Figure 2.12: ¡Hay un bug en nuestro juego cuando se hace clic en nuestra celda!

¿Qué pasó? Para averiguarlo, usemos uno de las herramientas más poderosas de Smalltalk: el debugger.

Haz clic en el botón debug *en la ventana de notificación.*

El debugger aparecerá. En la parte superior de la ventana del debugger

puedes ver la pila de ejecución, mostrando todos los métodos activos; seleccionando alguno de ellos se mostrará, en el panel del medio, el código Smalltalk ejecutándose en ese método, con la parte que disparó el error resaltada.

Has clic en la línea etiquetada LOGame»toggleNeighboursOfCellAt:at: *(near the top).*

El debugger te mostrará el contexto de ejecución de este método donde ocurrió el error (Figure 2.13).

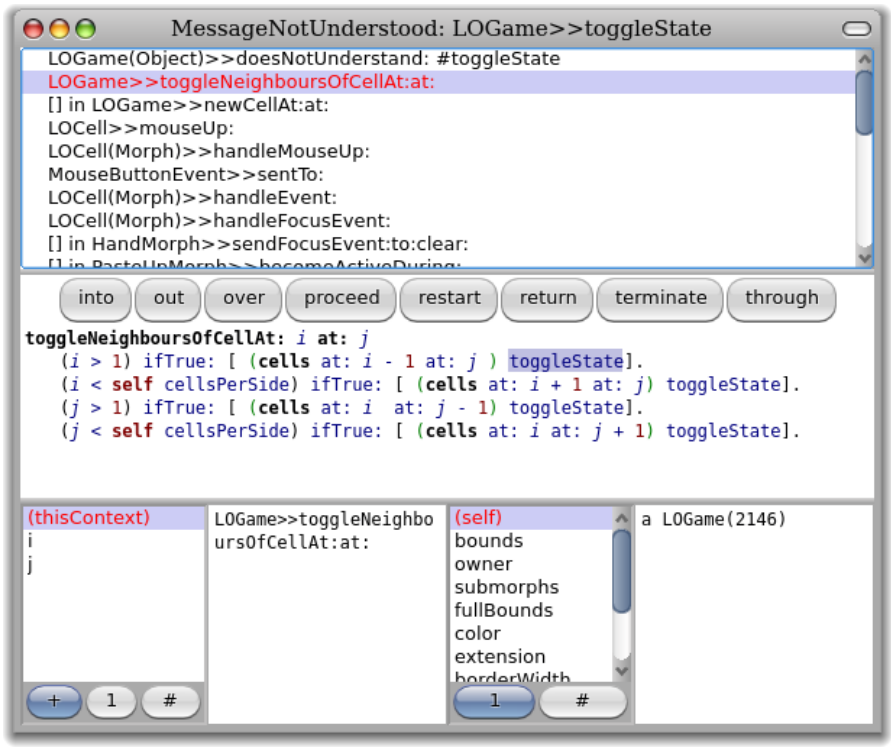


Figure 2.13: El debugger, con el método toggleNeighboursOfCell:at: seleccionado.

En la parte inferior del debugger hay dos pequeñas ventanas de inspección. A la izquierda, puedes inspeccionar el objeto receptor del mensaje que causó que el mensaje seleccionado se ejecute, asi que puedes mirar aquí para ver los valores de las variables de instancia. A la derecha puedes inspeccionar un objeto que representa el método que se está ejecutando, asi que puedes mirar aquí para ver los valores de los parámetros del método y las variables temporales.

Usando el debugger, puedes ejecutar código paso a paso, inspeccionar objetos en parámetros y variables locales, evaluar código como lo puedes hacer en el workspace, y, más sorprendente aún para los que están acostumbrados a otros debuggers, ¡cambiar el código mientras está siendo depurado! Algunos programadores Smalltalk programan en el debugger casi todo el tiempo, en lugar de usar el navegador. La ventaja de esto es que puedes ver el método que estás escribiendo como será ejecutado, con parámetros reales en el contexto de ejecución real.

En este caso podemos ver en la primera línea del panel superior que el mensaje toggleState fué enviado a una instancia de LOGame, cuando, claramente, debería haber sido una instancia de LOCell. El problema está seguramente con la inicialización de la matriz cells. Explorando el código de LOGame»initialize nos muestra que cells se llena con el valor de retorno de newCellAt:at:, pero cuand miramos ese método, ¡vemos que no hay sentencia de retorno! Por defecto, un método retorna self, que en el caso de newCellAt:at: es de hecho una instancia de LOGame.

Cierra la ventana del debugger. Agrega la expresión "↑ c" al final del método LOGame»newCellAt:at: *para que devuelva* c. *(Ver method 2.10.)*

Method 2.10: *Corrigiendo el bug.*

```
LOGame»newCellAt: i at: j
  "Create a cell for position (i,j) and add it to my on−screen
  representation at the appropriate screen position.  Answer the new cell"
  | c origin |
  c := LOCell new.
  origin := self innerBounds origin.
  self addMorph: c.
  c position: ((i − 1) * c width) @ ((j − 1) * c height) + origin.
  c mouseAction: [self toggleNeighboursOfCellAt: i at: j].
  ↑ c
```

Recuerda del Chapter 1 que la manera de hacer return de un valor de un método en Smalltalk es ↑, que obtienes tipeando ^.

A menudo puedes arreglar el código directamente en la ventaba del debugger y hacer clic en Proceed para continuar corriendo la aplicación. En nuestro caso, como el bug estaba en la inicialización de un objeto, y no en el método que falló, lo más fácil es cerrar la ventana de debug, destruir la intancia del juego ejecutándose (con el halo), y crear una nueva.

Do: LOGame new openInWorld *otra vez.*

Ahora el juego debería funcionar correctamente... o casi. Si hacemos clic, y luego movemos el ratón antes de soltar el botón, la celda sobre la que esta el ratón también cambiará. Resulta que esto es comportamiento

que heredamos de SimpleSwitchMorph. Podemos arreglar esto simplemente sobreescribiendo mouseMove: para que no haga nada:

Method 2.11: *Sobreescribiendo las acciones de movimiento del ratón.*

```
LOGame»mouseMove: anEvent
```

¡Finalmente terminamos!

2.9 Guardando y compartiendo código Samalltalk

Ahora que tienes el juego Lights Out funcionando, probablemente quieras guardarlo en algún lugar para poder compartirlo con tus amigos. Por supuesto, puedes guardar tu imágen de Pharo completa, y mostrar tu primer programa ejecutándola, pero tus amigos probablemente tengan su propio código en sus imágenes, y no quieran renunciar a eso para usar tu imágen. Lo que necesitas es una manera de sacar tu código fuente fuera de tu imágen Pharo para que otros programadores puedan introducirla en las suyas.

La manera más sencilla de hacer esto es haciendo *file out* a nuestro código. El menú action-click en el panel de Paquete te dará la opción various ▷ file out de todo el paquete *PBE-LightsOut*. El archivo resultante es más o menos humanamente legible, pero está realmente pensado para computadoras, no humanos. Puedes enviar este archivo via email a tus amigos, y ellos pueden archivarlo dentro de sus propias imágenes Pharo, usando el explorador de archivos.

Action-click en el paquete PBE-LightsOut *y has* various ▷ file out *de los contenidos.*

Ahora deberías encontrar un archivo llamado "PBE-LightsOut.st" en el mismo directorio del disco donde se encuentra tu imágen. Da una mirada a este archivo con un editor de texto.

*Abre una nueva imágen de Pharo y usa la herramienta File Browser (*Tools . . . ▷ File Browser*) para hacer* file in *al PBE-LightsOut.st fileout. Verifica que el juego ahora funciona en la nueva imágen.*

Paquetes Monticello

Por más que los fileouts son una manera conveniente de tomar una fotografía del código que has escrito, son definitivamente de "la vieja escuela". Así como muchos proyectos open-source hallan mucho más conveniente

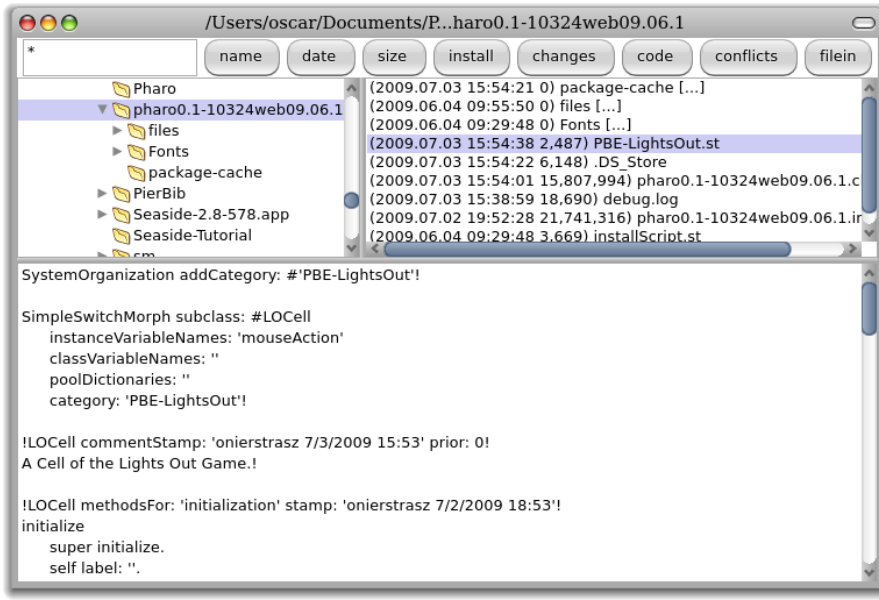


Figure 2.14: Filing in código fuente Pharo.

mantener su código en un repositorio usando CVS[4] o Subversion[5], los programadores Pharo encuentran más conveniente manejar su código usando paquetes Monticello. Estos paquetes se representan como archivos con nombres terminados en .mcz; son en realidad paquetes comprimidos en formato zip que contienen el código completo de tu paquete.

Usando el navegador de paquetes Monticello, puedes guardar paquetes en repositorios en varios tipos de servidores, incluyendo servidores FTP y HTTP; también puedes escribir paquetes en un repositorio en un sistema de archivos local. Una copia de tu paquete se cachea siempre en tu disco duro local, en el directorio *package-cache*. Monticello te permite guardar múltiples versiones de tu programa, mezclar versiones, volver a una versión vieja, y ver las diferencias entre las versiones.

De hecho, Monticello es un sistema de control de versiones distribuido; esto significa que permite a los desarrolladores guardar su trabajo en diferentes lugares, no en un solo repositorio como es el caso de CVS o Subversion.

También puedes enviar un archivo .mcz por email. El receptor tendrá que ponerlo en su directorio *package-cache*; Luego será capaz de usar Monticello para navegar y cargarlo.

[4] http://www.nongnu.org/cvs
[5] http://subversion.tigris.org

Abre el navegador Monticello desde el menú World.

En el panel de la derecha del navegador (ver Figure 2.15) hay una lista de repositorios Monticello, que incluirá todos los repositorios desde los cuales fué cargado código en la imágen que estás usando.

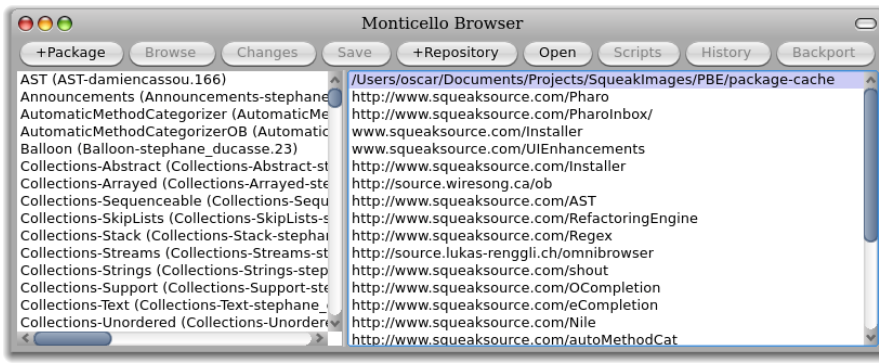


Figure 2.15: El navegador Monticello.

Al principio de la lista en el navegador Monticello hay un repositorio en un directorio local llamado *package cache*, que guarda en cache copias de los paquetes que has cargado o publicado en la red. Este cache local es realmente útil porque te permite guardar tu propio historial; también te permite trabajar en lugares donde no tienes acceso a internet, o cuando la conexión es tan lenta que no quieres guardar en un repositorio remoto con mucha frecuencia.

Guardando y cargando código con Monticello.

A la izquierda del navegador Monticello hay una lista de paquetes que tienen una versión cargada en la imágen; los paquetes que han sido modificados desde que han sido cargados están marcados con un asterisco. (Estos algunas veces son nombrados como paquetes sucios.) Si seleccionas un paquete, la lista de repositorios se restringe a aquellos repositorios que contienen una copia del paquete seleccionado.

Agrega el paquete PBE−LightsOut *a tu navegador Monticello usando el botón* +Package *y escribe* PBE−LightsOut.

SqueakSource : un SourceForge para Pharo.

Pensamos que la mejor manera de guardar tu código y compartirlo es creando una cuenta para tu proyecto en un servidor SqueakSource. Squeak-

Source es como SourceForge[6]: es un front-end web para un servidor Monticello HTTP que te permite manejar tus proyectos. Hay un servidor SqueakSource público en http://www.squeaksource.com, y hay una copia del código en este libro en http://www.squeaksource.com/PharoByExample.html. Puedes mirar este proyecto con un navegador web, pero es más productivo hacerlo desde dentro de Pharo, usando el navegador Monticello, que te permite manejar tus paquetes.

 *Abre http://www.squeaksource.com en un navegador web. Crea una cuenta y luego crea ("registra") un proyecto para el juego Lights Out.*

SqueakSource te mostrará la información que deberías usar para agregar un repositorio usando Monticello.

Una vez que tu proyecto haya sido creado en SqueakSource, tienes que indicarle a tu sistema Pharo que lo use.

 *Con el paquete* PBE−LightsOut *seleccionado, haz clic en el botón* +Repository *en el navegador Monticello.*

Verás una lista con los diferentes tipos de repositorios que están disponibles; para agregar un reposotorio SqueakSource selecciona HTTP. Podrás proveer la información necesaria sobre el servidor. Debes copiar la plantilla mostrada para identificar tu proyecto SqueakSource, pegarlo en Monticello y proveer tus iniciales y contraseña:

```
MCHttpRepository
   location: 'http://www.squeaksource.com/YourProject'
   user: 'yourInitials'
   password: 'yourPassword'
```

Si provees iniciales y contraseña vacías, también puedes cargar el proyecto, pero no podrás actualizarlo:

```
MCHttpRepository
   location: 'http://www.squeaksource.com/YourProject'
   user: ''
   password: ''
```

Una vez que hayas aceptado esta plantilla, tu nuevo repositorio debería ser listado en el lado derecho del navegador Monticello.

 *Haz clic en el bontón* Save *para guardar una primera versión de tu juego Lights Out en SqueakSource.*

Para cargar el paquete en tu imágen, primero debes seleccionar una versión particular. Puedes hacer esto en el navegador del reposotorio, que

[6] http://sourceforge.net

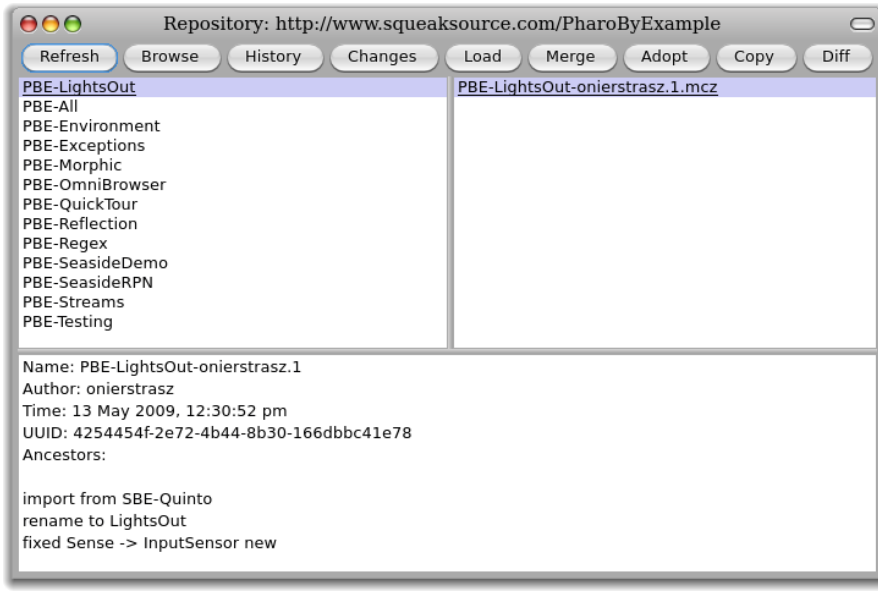


Figure 2.16: Navegando en un reposotorio Monticello

puedes abrir usando el botón Open o el menú action-click. Una vez que hayas seleccionado una versión, podrás cargarla en tu imágen.

Abre el reposotorio PBE−LightsOut *que acabas de guardar.*

Monticello tiene muchas más capacidades, que serán abordadas con más profundidad en Chapter 6. También puedes mirar la documentación en línea de Monticello en http://www.wiresong.ca/Monticello/.

2.10 Resúmen del capítulo

En este capítulo has visto como crear categorías, clases y métodos. Has visto como usar el navegador, el inspector, el depurador y el navegador Monticello.

- Las categorías son grupos de clases relacionadas entre sí.
- Una nueva clase se crea enviando un mensaje a su superclase.
- Los protocolos son grupos de métodos relacionados entre sí.
- Un nuevo método se crea o modifica editando su definición en el navegador y *aceptando* los cambios.

- El inspector ofrece una GUI simple y multi-propósito para inspeccionar e interactuar con objetos arbitrarios-
- El navegador detecta el uso de métodos y variables no declarados, y ofrece posibles correcciones.
- El método initialize se ejecuta automáticamente después de que un objeto es creado en Pharo. Puedes poner cualquier código de inicialización ahí.
- El depurador provee una GUI de alto nivel para inspeccionar y modificar el estado de un programa en tiempo de ejecución.
- Puedes compartir código fuente haciendo *file out* de una categoría.
- Una mejor manera de compartir código es usando Monticello para manejar un reposotorio externo, por ejemplo definido como un proyecto SqueakSource.

Chapter 3

Sintaxis en dos palabras

Pharo, como la mayoría de los dialectos de Smalltalk modernos, adopta una sintaxis muy similar a la de Smalltalk-80. La sintaxis está diseñada para que el texto del programa pueda leerse como una especie de inglés macarrónico:

```
(Smalltalk includes: Class) ifTrue: [ Transcript show: Class superclass ]
```

La sintaxis de Pharo es mínima. Esencialmente, sólo hay sintaxis para el *envío de mensajes* (es decir, expresiones). Las expresiones se construyen a partir de un número muy pequeño de elementos primitivos. Hay sólo 6 palabras clave y no existe sintaxis para las estructuras de control o la declaración de nuevas clases. En su lugar, casi todo se consigue enviando mensajes a objetos. Por ejemplo, en lugar de una estructura de control if-then-else, Smalltalk envía mensajes como ifTrue: a objetos Boolean. Las nuevas (sub)clases se crean enviando un mensaje a sus superclases.

3.1 Elementos sintácticos

Las expresiones se componen de las siguientes piezas: (i) seis palabras reservas o *pseudo-variables*: self, super, nil, true, false y thisContext; (ii) expresiones constantes para *objetos literales* incluyendo números, caracteres, cadenas de caracteres, símbolos y *arrays*; (iii) declaraciones de variables; (iv) asignaciones; (v) bloques de cierre y (vi) mensajes.

Podemos ver ejemplos de los distintos elementos sintácticos en la Table 3.1.

Variables locales puntoInicial es un nombre de variable, o identificador. Por convenio, los identificadores están formados por palabras en "camel-

Sintaxis	Lo que representa
puntoInicial	un nombre de variable
Transcript	un nombre de variable global
self	una pseudo-variable
1	entero decimal
2r101	entero binario
1.5	número en coma flotante
2.4e7	notación exponencial
$a	el carácter 'a'
'Hola'	la cadena de caracteres "Hola"
#Hola	el símbolo #Hola
#(1 2 3)	un array literal
{1. 2. 1+2}	un array dinámico
"un comentario"	un comentario
| x y |	declaración de las variables x e y
x := 1	asignar 1 a x
[x + y]	un bloque que se evalúa como x+y
<primitive: 1>	primitiva o anotación de la máquina virtual
3 factorial	mensaje unario
3+4	mensaje binario
2 raisedTo: 6 modulo: 10	mensaje de palabra clave
↑ true	devuelve el valor true
Transcript show: 'hola'. Transcript cr	separator de expresiones (.)
Transcript show: 'hola'; cr	mensaje en cascada (;)

Table 3.1: Sintaxis de Pharo en dos palabras

Case" (es decir, cada palabra excepto la primera comienza con una letra mayúscula). La primera letra de una variable de instancia, método, argumento de un bloque o variable temporal debe ser minúscula. Esto indica al lector que la variable tiene un alcance privado.

Variables compartidas Los identificadores que comienzan con letras mayúsculas son variables globales, variables de clase, diccionarios comunes (pool dictionaries) o nombres de clase. Transcript es una variable global, una instancia de la clase TranscriptStream.

El receptor self es una palabra clave que se refiere al objeto en el que el método actual se está ejecutando. Lo llamamos "el receptor" porque normalmente este objeto habrá recibido el mensaje que causó que el método se ejecutara. Decimos que self es una "pseudo-variable" ya

que no podemos asignarla.

Enteros Además de los enteros decimales normales como 42, Pharo también proporciona una notación científica. 2r101 es 101 en base 2 (es decir, binario), que es igual al número decimal 5.

Números en coma flotante Pueden especificarse en potencias de base diez: 2.4e7 es 2.4×10^7.

Caracteres Un signo dólar introduce un carácter literal: $a es el literal para 'a'. Pueden obtenerse instancias de caracteres no imprimibles enviando los mensajes adecuados a la clase Character, como Character space y Character tab.

Cadenas de caracteres Las comillas simples se utilizan para definir una c adena literal. Si quieres una cadena que contenga una comilla, simplemente se usan dos comillas, como en 'entre "comillas" simples'.

Símbolos se parecen a las cadenas en que contienen una secuencia de caracteres. Sin embargo, al contrario que en una cadena, se garantiza que un símbolo literal es único globalmente. Sólo existe un objeto símbolo #Hola, pero puede haber varios objetos String con el valor 'Hola'.

Arrays en tiempo de compilación Se definen mediante #( ) rodeando literales separados por espacios. Dentro de los paréntesis todo debe ser constante en tiempo de compilación. Por ejemplo, #(27 (true false) abc) es un array literal de tres elementos: el entero 27, el array en tiempo de compilación conteniendo los dos booleanos y el símbolo #abc (observa que esto es lo mismo que #(27 #(true false) #abc)).

Arrays en tiempo de ejecución Las llaves { } definen un array (dinámico) en tiempo de ejecución. Sus elementos son expresiones separadas por puntos. Por lo tanto, { 1. 2. 1+2 } define un array con los elementos 1, 2 y el resultado de evaluar 1+2 (la notación de llaves es específica de los dialectos de Smalltalk Pharo y Squeak. En otros Smalltalks tienes que construir explícitamente los arrays dinámicos).

Comentarios Se encierran entre comillas dobles. *"Hola"* es un comentario, no una cadena de caracteres, y es ignorado por el compilador de Pharo. Los comentarios pueden ocupar varias líneas.

Definiciones de variables locales Las barras verticales || encierran la declaración de una o más variables locales en un método (y también en un bloque).

Asignación := asigna un objeto a una variable.

Bloques Los corchetes [] definen un bloque, conocido también como clausura de bloque o clausura léxica, que es un objeto de primer orden representando una función. Como veremos, los bloques pueden tener argumentos y variables locales.

Primitivas <primitive: ...> indica la invocación de una primitiva de máquina virtual. (<primitive: 1> es la primitiva de máquina virtual para SmallInteger»+). Cualquier código que siga a la primitiva es ejecutado sólo si la primitiva falla. La misma sintaxis se utiliza también para las anotaciones de los métodos.

Mensajes unarios consisten en una sola palabra (como factorial) enviada a un receptor (como 3).

Mensajes binarios son operadores (como +) enviados a un receptor y que tienen un único argumento. En 3+4, el receptor es 3 y el argumento es 4.

Mensajes de palabra clave consisten en varias palabras clave (como raisedTo:modulo:), acabando cada una con dos puntos y recibiendo un único argumento. En la expresión 2 raisedTo: 6 modulo: 10, el *selector del mensaje* raisedTo:modulo: recibe los dos argumentos 6 y 10, cada uno siguiente los dos puntos. Enviamos el mensaje al receptor 2.

Retorno de método ↑ se usa para *retornar* un valor desde un método (debes escribir ^ para obtener el caracter ↑).

Secuencias de sentencias Un punto (.) es el *separador* de *sentencias*. Poner un punto entre dos expresiones las convierte en sentencias independientes.

Cascadas Los puntos y coma pueden usarse para enviar una *cascada* de mensajes a un único receptor. En Transcript show: 'hola'; cr primero enviamos el mensaje de palabra clave show: 'hola' al receptor Transcript y entonces enviamos el mensaje unario cr al mismo receptor.

Las clases Number, Character, String y Boolean se describen con más detalle en Chapter 8.

3.2 Pseudo-variables

En Smalltalk existen 6 palabras reservadas o *pseudo-variables*: nil, true, false, self , super y thisContext. Se llaman pseudo-variables porque están predefinidas y no pueden ser asignadas. true, false y nil son constantes, mientras que los valores de self, super y thisContext varían dinámicamente cuando el código es ejecutado.

true y false son las únicas instancias de las clases Boolean True y False. Véase Chapter 8 para más detalles.

self siempre se refiere al receptor del método ejecutándose actualmente.

super también se refiere al receptor del mensaje actual, pero cuando envías un mensaje a super, la búsqueda del método cambia, de modo que comienza desde la superclase de la clase que contiene el método que usa super. Para más detalles véase Chapter 5.

nil es el objeto indefinido. Es la única instancia de la clase UndefinedObject. Las variables de instancia, variables de clase y variables locales se inicializan a nil.

thisContext es una pseudo-variable que representa el marco superior de la pila de ejecución. En otras palabras, representa el MethodContext o BlockClosure ejecutándose actualmente. Normalmente, thisContext no es de interés para la mayoría de los programadores, pero es imprescindible para implementar herramientas de desarrollo como el depurador y también se utiliza para implementar el manejo de excepciones y las continuaciones.

3.3 Envíos de mensajes

Existen tres tipos de mensajes en Pharo.

1. Los mensajes *unarios* no reciben ningún argumento. `1 factorial` envía el mensaje factorial al objeto 1.
2. Los mensajes *binarios* reciben exactamente un argumento. `1 + 2` envía el mensaje + con el argumento 2 al objeto 1.
3. Los mensajes *de palabra clave* reciben un número arbitrario de argumentos. `2 raisedTo: 6 modulo: 10` envía el mensaje que consiste en el selector del mensaje raisedTo:modulo: y los argumentos 6 y 10 al objeto 2.

Los selectores de los mensajes unarios consisten en caracteres alfanuméricos y comienzan con una letra minúscula.

Los selectores de los mensajes binarios consisten en uno o más de los siguientes caracteres:

```
+ - / \ * ~ < > = @ % | & ? ,
```

Los selectores de mensajes de palabra clave consisten en una serie de palabras clave alfanuméricas que comiezan con una letra minúscula y acaban con dos puntos.

Los mensajes unarios tienen la mayor precedencia, después los mensajes binarios y por último, los mensajes de palabra clave, por lo tanto:

```
2 raisedTo: 1 + 3 factorial  ⟶  128
```

(Primero enviamos factorial a 3, después enviamos + 6 a 1 y por último, enviamos raisedTo: 7 a 2.) Recuerda que usamos la notación *expresión* ⟶ *resultado* para mostrar el resultado de la evaluación de una expresión.

Aparte de la precedencia, la evaluación se estrictamente de izquierda a derecha, con lo que:

```
1 + 2 * 3  ⟶  9
```

no 7. Se deben usar paréntesis para cambiar el orden de la evaluación:

```
1 + (2 * 3)  ⟶  7
```

Los envíos de mensajes pueden componerse por puntos y puntos y coma. Una secuencia de expresiones separadas por puntos causa que cada expresión sea evaluada como una *sentencia* detrás de otra.

```
Transcript cr.
Transcript show: 'hola mundo'.
Transcript cr
```

Esto enviará cr al objeto Transcript, después le enviará show: 'hello world' y por último, otro cr.

Cuando una serie de mensajes se envía al *mismo* receptor, puede expreserse más brevemente como una *cascada*. El receptor se especifica sólo una vez y la secuencia de mensajes se separa por puntos y coma:

```
Transcript cr;
  show: 'hola mundo';
  cr
```

Esto tiene exactamente el mismo efecto que el ejemplo anterior.

3.4 Sintaxis de los métodos

Mientras que las expresiones se pueden evaluar en cualquier sitio en Pharo (por ejemplo, en un espacio de trabajo, en un depurador o en un navegador), los métodos se definen normalmente en una ventana de navegador o en el depurador (los métodos también pueden ser introducidos desde un medio externo, pero esta no es la forma habitual de programar en Pharo).

Los programas se desarrollan escribiendo método a método en el contexto de una clase dada (una clase se define enviando un mensaje a una clase

existente, pidiéndole crear una clase, por lo que no es necesaria una sintaxis especial para la definición de clases).

Este es el método lineCount en la clase String (el convenio habitual es referirse a los métodos como NombreDeClase»nombreDeMetodo, por lo que llamamos a este método String»lineCount).

Method 3.1: *Line count*

```
String»lineCount
  "Answer the number of lines represented by the receiver,
  where every cr adds one line."
  | cr count |
  cr := Character cr.
  count := 1 min: self size.
  self do:
    [:c | c == cr ifTrue: [count := count + 1]].
  ↑ count
```

Sintácticamente un método consiste en:

1. El patrón del metódo, que contiene el nombre (es decir, lineCount) y los argumentos (ninguno en este ejemplo);
2. comentarios (pueden aparecer en cualquier sitio, pero el convenio es colocar uno al comienzo que explique lo que hace el método);
3. declaraciones de las variables locales (es decir, cr y count); y
4. cualquier número de expresiones separadas por puntos; aquí hay cuatro.

La evaluación de cualquier expresión precedida por un ↑ (escrito como ^) causará que el método salga en ese punto, devolviendo el valor de la expresión. Un método que termina sin devolver explícitamente ninguna expresión, devolverá implícitamente self.

Los argumentos y variables locales deberían comenzar siempre con letras minúsculas. Los nombres que comienzan con letras mayúsculas se supone que son variables globales. Los nombres de clases, como Character, por ejemplo, son simplemente variables globales que se refieren al objeto que representa esa clase.

3.5 Sintaxis de los bloques

Los bloques proporcionan un mecanismo para diferir la evaluación de expresiones. Un bloque es básicamente una función anónima. Un bloque se

evalúa enviándole el mensaje value. El bloque responde con el valor de la última expresión de su cuerpo, si no existe un valor de retorno explícito (con ↑), en cuyo caso no responde ningún valor.

```
[ 1 + 2 ] value    ⟶    3
```

Los bloques pueden recibir parámetros, cada uno de los cuales se declara con dos puntos precediéndole. Una barra vertical separa la declaración de parámetros del cuerpo del bloque. Para evaluar un cuerpo con un parámetro, debes enviar el mensaje value: con un argumento. A un bloque de dos parámetros se le debe enviar value:value:, y así, hasta los 4 argumentos.

```
[ :x | 1 + x ] value: 2    ⟶    3
[ :x :y | x + y ] value: 1 value: 2    ⟶    3
```

Si tienes un bloque con más de cuatro parámetros, debes usar valueWithArguments: y pasar los argumentos en un array (un bloque con un número grande de parámetros es con frecuencia señal de un problema de diseño).

Los bloques pueden también declarar variables locales, que son repodeadas por barras verticales, tal como las declaraciones de variables locales en un método. Las variables locales son declaradas después de los argumentos:

```
[ :x :y | | z | z := x+ y. z ] value: 1 value: 2    ⟶    3
```

Los bloques son en realidad *cierres* léxicos, ya que pueden referirse a las variables del entorno circundante. El siguiente bloque se refiere a la varaible x del entorno que lo contiene:

```
| x |
x := 1.
[ :y | x + y ] value: 2    ⟶    3
```

Los bloques son instancias de la clase BlockClosure. Esto significa que son objetos, por lo que pueden ser asignados a variables y pasados como argumentos como cualquier otro objeto.

3.6 Condicionales y bucles en dos palabras

Smalltalk no ofrece una sintaxis especial para construcciones de control. En su lugar, típicamente se expresan mediante el envío de mensajes a booleanos, números y colecciones, con bloques como argumentos.

Los condicionales se expresan enviando uno de los mensajes ifTrue:, ifFalse: o ifTrue:ifFalse: al resultado de una expresión booleana. Véase Chapter 8 para más sobre booleanos.

```
(17 * 13 > 220)
  ifTrue: [ 'mayor' ]
  ifFalse: [ 'menor' ]    ⟶    'mayor'
```

Los bucles son típicamente expresados mediante el envío de mensajes a bloques, enteros o colecciones. Puesto que la condición de salida para un bucle puede ser evaluada repetidamente, debería ser un bloque en lugar de un valor booleano. Este es un ejemplo de un bucle muy procedimental:

```
n := 1.
[ n < 1000 ] whileTrue: [ n := n*2 ].
n    ⟶    1024
```

whileFalse: invierte la condición de salida.

```
n := 1.
[ n > 1000 ] whileFalse: [ n := n*2 ].
n    ⟶    1024
```

timesRepeat: ofrece una forma sencilla de implementar un iteración fija:

```
n := 1.
10 timesRepeat: [ n := n*2 ].
n    ⟶    1024
```

También podemos enviar el mensaje to:do: a un número que entonces actúa como el valor inicial del contador de un bucle. Los dos argumentos son el límite superior y un bloque que toma el valor actual del contador del bucle como su argumento:

```
result := String new.
1 to: 10 do: [:n | result := result, n printString, ' '].
result    ⟶    '1 2 3 4 5 6 7 8 9 10 '
```

Iteradores de orden superior. Las colecciones comprenden un gran número de clases diferentes, muchas de las cuales soportan el mismo protocolo. Los mensajes más importantes para iterar sobre las colecciones incluyen do:, collect:, select:, reject:, detect: and inject:into:. Estos mensajes definen iteradores de orden superior que nos permiten escribir código muy compacto.

Un Interval es una colección que nos permite iterar sobre una secuencia de números desde el punto de inicio hasta el final. 1 to: 10 representa el intervalo desde 1 hasta 10. Ya que es una colección, podemos enviarle el mensaje do:. El argumento es un bloque que es evaluado por cada elemento de la colección.

```
result := String new.
(1 to: 10) do: [:n | result := result, n printString, ' '].
result    ⟶    '1 2 3 4 5 6 7 8 9 10 '
```

collect: construye una nueva colección del mismo tamaño, transformando cada elemento.

```
(1 to: 10) collect: [ :each | each * each ]    ⟶    #(1 4 9 16 25 36 49 64 81 100)
```

select: y reject: construyen nuevas colecciones, cada consistiendo en un subconjunto de los elementos que satisfacen (o no) la condición del bloque booleano. detect: devuelve el primer elemento que satisface la condición. No olvides que las cadenas son también colecciones, por lo que puedes iterar sobre todos los caracteres.

```
'hello there' select: [ :char | char isVowel ]    ⟶    'eoee'
'hello there' reject: [ :char | char isVowel ]    ⟶    'hll thr'
'hello there' detect: [ :char | char isVowel ]    ⟶    $e
```

Finalmente, deberías tener presente que las colecciones también soportan un operador como el *fold* del estilo funcional en el método inject:into: method . Este te permite generar un resultado acumulado usando una expresión que comienza con un valor semilla y que inyecta cada elemento de la colección. Sumas y productos son ejemplo típicos.

```
(1 to: 10) inject: 0 into: [ :sum :each | sum + each ]    ⟶    55
```

Esto es equivalente a 0+1+2+3+4+5+6+7+8+9+10.

Puedes encontrar más sobre colecciones en Chapter 9.

3.7 Primitivas y pragmas

En Smalltalk todo es un objeto y todo ocurre mediante el envío de mensajes. No obstante, en ciertos puntos tocamos el fondo. Algunos objetos sólo pueden funcionar invocando las primitivas de la máquina virtual.

Por ejemplo, todo lo siguiente es implementado como primitivas: reserva de memoria (new, new:), manipulación de bits (bitAnd:, bitOr:, bitShift:), aritmética entera y decimal (+, −, <, >, *, / , =, ==...), y el acceso a los arrays (at:, at:put:).

Las primitivas se invocan con la sintaxis <primitive: unNumero>. Un método que invoque tal primitiva también puede incluir código Smalltalk, que será evaluado *sólo* si la primitiva falla.

A continuación vemos le código para SmallInteger»+. Si la primitiva falla, la expresión super + unNumero será evaluada y devuelta.

Method 3.2: *Un método primitivo*

```
+ aNumber
  "Primitive. Add the receiver to the argument and answer with the result
  if it is a SmallInteger. Fail if the argument or the result is not a
  SmallInteger  Essential  No Lookup. See Object documentation whatIsAPrimitive."

  <primitive: 1>
  ↑ super + aNumber
```

En Pharo, la sintaxis del corchete angular también se utiliza para las anotaciones a los métodos, llamadas pragmas.

3.8 Resumen del capítulo

- Pharo (sólo tiene) seis identificadores reservados también llamados *pseudo-variables*: true, false, nil, self, super y thisContext.
- Hay cinco tipos de objetos literales: números (5, 2.5, 1.9e15, 2r111), caracteres ($a), cadenas ('hola'), símbolos (#hola) y arrays (#('hola' #hola))
- Las cadenas están delimitadas por comillas simples, los comentarios por comillas dobles. Para tener una cita dentro de una cadena, usa doble comilla.
- A diferencia de las cadenas, se garantiza que los símbolos son únicos globalmente.
- Usa #(...) para definir un array literal. Usa { ... } para definir un array dinámico. Observa que #(1 + 2) size ⟶ 3, pero que { 1 + 2 } size ⟶ 1
- Hay tres tipos de mensajes: *unarios* (*e.g.*, 1 asString, Array new), *binarios* (*e.g.*, 3 + 4, 'hola' , 'mundo') y *de palabra clave* (*e.g.*, 'hola' at: 2 put: $o)
- Un mensaje *en cascada* envía una secuencia de mensajes al mismo destino , separados por puntos y coma: OrderedCollection new add: #calvin; add: #hobbes; size ⟶ 2
- Las variables locales se declara con barras verticales. Usa := para la asignación. |x| x:=1
- Las expresiones consisten en envíos de mensajes, cascadas y asignaciones, posiblemente agrupados con paréntesis. Las *sentencias* son expresiones separadas por puntos.

- Los bloques de cierre son expresiones encerradas en corchetes. Los bloques pueden tener argumentos y pueden contener variables temporales. Las expresiones en el bloque no se evalúan hasta que envías al bloque un mensaje value... con el número de argumentos correcto.
  `[:x | x + 2] value: 4` $\longrightarrow$ `6.`

- No hay una sintaxis dedicada para las construcciones de control, sólo mensajes que condicionalmente evalúan bloques.
  `(Smalltalk includes: Class) ifTrue: [ Transcript show: Class superclass ]`

Chapter 4

Comprendiendo la sintaxis de los mensajes

Aunque la sintaxis de los mensajes de Smalltalk es extremadamente simple, no es convencional y puede llevar algún tiempo sentirse familiarizado con ella. Este capítulo ofrece alguna guía para ayudarte a estar aclimatado a esta sintaxis especial de envío de mensajes. Si ya te sientes cómodo con la sintaxis, puedes elegir saltarte este capítulo, o volver al mismo mas tarde.

4.1 Identificando mensajes

En Smalltalk, excepto para los elementos sintácticos listados en Chapter 3 (: = ↑ . ; # () {} [: |]), todo es un envío de mensaje. Como en C++, puedes definir operadores como + para tus propias clases, pero todos los operadores tienen la misma precedencia. Además, no puedes cambiar la aridad de un método. "−" es siempre un mensaje binario; no hay ninguna manera de tener un " −" unario con una sobrecarga diferente.

En Smalltalk el orden en el cual los mensajes se envían es determinado por el tipo de mensaje. Hay solo tres tipos de mensajes: mensajes *unarios*, mensajes *binarios*, y mensajes *keyword*. Los mensajes unarios siempre se envían primero, luego los mensajes binarios y finalmente los mensajes keyword. Como en la mayoría de los lenguajes, los paréntesis pueden ser utilizados para cambiar el orden de evaluación. Estas reglas hacen el código Smalltalk lo mas fácil de leer posible. Y en la mayor parte del tiempo no tienes que estar pensando en las reglas.

Como la mayor parte del cómputo en Smalltalk se realiza a través del pasaje de mensajes, identificar correctamente los mismos es crucial. La siguiente terminología va a ayudarnos:

- Un mensaje se compone con un *selector* de mensaje y opcionalmente con argumentos de mensaje.
- Un mensaje es enviado a un *receptor*.
- La combinación de un mensaje y su receptor es denominada *envío de mensaje* como se muestra en Figure 4.1.

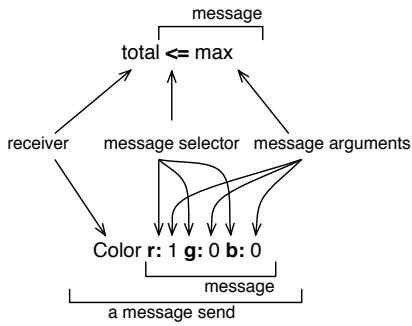


Figure 4.1: Dos mensajes compuestos por un receptor, un selector de método, y un conjunto de argumentos.

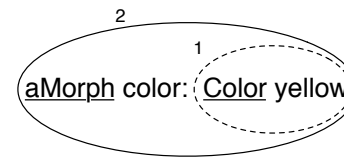


Figure 4.2: aMorph color: Color yellow está compuesto por dos envíos de mensaje: Color yellow y aMorph color: Color yellow.

> Un mensaje siempre es enviado a un receptor, el cual puede ser un simple literal, un bloque, una variable o el resultado de evaluar otro mensaje.

Para ayudarte a identificar el receptor de un mensaje, te lo subrayaremos. También vamos a rodear cada envío de mensaje con una elipse con el número de envío de mensaje, comenzando por el primero que será enviado, para ayudarte a ver el orden en el cual los mensajes son enviados.

La Figure 4.2 presenta dos envíos de mensajes, Color yellow y aMorph color: Color yellow, por lo tanto hay dos elipses. El envío de mensaje Color yellow es ejecutado primero y en consecuencia su elipse es numerada con 1. Hay dos receptores: aMorph el cual recibe el mensaje color: ... y Color el cual recibe el mensaje yellow. Ambos receptores están subrayados.

Un receptor puede ser el primer elemento de un mensaje, como 100 en el envío de mensaje 100 + 200 o Color en el envío de mensaje Color yellow. De cualquier manera, un receptor puede ser tambíen el resultado de otro envío de mensaje. Por ejemplo, en el mensaje Pen new go: 100, el receptor del mensaje go: 100 es el objeto devuelto por el envío del mensaje Pen new. En todos

Envío de mensaje	Tipo de mensaje	Resultado
`Color yellow`	unario	Crea un color.
`aPen go: 100.`	keyword	El lápiz receptor se mueve 100 píxeles adelante.
`100 + 20`	binario	El número 100 recibe el mensaje + con el número 20.
`Browser open`	unario	Abre un nuevo navegador.
`Pen new go: 100`	unario y keyword	Se crea un lápiz y se lo mueve 100 píxeles.
`aPen go: 100 + 20`	keyword y binario	El lápiz receptor se mueve 120 píxeles adelante.

Table 4.1: Ejemplos de Envíos de Mensajes y sus Tipos

los casos, un mensaje es enviado a un objeto llamado *receptor* el cual puede ser el resultado de otro envío de mensaje.

La Table 4.1 muestra varios ejemplos de envíos de mensajes. Deberías notar que no todos los envíos de mensaje tienen argumentos. Los mensajes unarios como `open` no tienen argumentos. Los mensajes keyword simples y los mensajes binarios como `go: 100` y `+ 20` tienen un argumento. También hay mensajes simples y compuestos. `Color yellow` y `100 + 20` son simples: un mensaje es enviado a un objeto, mientras que el envío de mensaje `aPen go: 100 + 20` está compuesto por dos mensajes: `+ 20` es enviado a `100` y `go:` es enviado a `aPen` con el argumento que haya resultado del primer mensaje. Un receptor puede ser una expresión (como una asignación, un envío de mensaje o un literal) que devuelva un objeto. En `Pen new go: 100`, el mensaje `go: 100` es enviado al objeto que resulta de la ejecución del envío del mensaje `Pen new`.

4.2 Tres tipos de mensajes

Smalltalk define unas pocas reglas simples para determinar el orden en el cual los mensajes son enviados. Estas reglas están basadas en la distinción entre 3 tipos diferentes de mensajes:

- Los *Mensajes Unarios* son mensajes que se envían a un objeto sin más información. Por ejemplo en `3 factorial`, `factorial` es un mensaje unario.
- Los *Mensajes Binarios* son mensajes que consisten de operadores (frecuentemente aritméticos). Son binarios porque siempre involucran sólo dos objetos: el receptor y el objeto argumento. Por ejemplo en `10 + 20`, `+` es un mensaje binario enviado al receptor `10` con el argumento

20.

- Los *Mensajes Keyword* son mensajes que consisten de una o mas palabras clave, cada una terminada por dos puntos (:) y tomando un argumento. Por ejemplo en anArray at: 1 put: 10, la palabra clave at: toma el argumento 1 y la palabra clave put: toma el argumento 10.

Mensajes Unarios

Los mensajes unarios son mensajes que no requieren ningún argumento. Ellos siguen el modelo sintáctico: receptor nombreDeMensaje. El selector se construye simplemente con una sucesión de caracteres que no contengan dos puntos (:) (*e.g.*, factorial, open, class).

```
89 sin          ⟶   0.860069405812453
3 sqrt          ⟶   1.732050807568877
Float pi        ⟶   3.141592653589793
'blop' size     ⟶   4
true not        ⟶   false
Object class    ⟶   Object class  "The class of Object is Object class (!)"
```

> Los mensajes unarios son mensajes que no requieren argumentos. Siguen el modelo sintáctico:
> receptor **selector**

Mensajes Binarios

Los mensajes binarios son mensajes que requieren exactamente un argumento *y* cuyo selector consiste en una sequencia de uno o mas caracteres del conjunto: +, −, *, /, &, =, >, |, <, ∼, y @. Nota que −− no está permitido por razones de análisis sintáctico.

```
100@100          ⟶   100@100  "crea un objeto Point"
3 + 4            ⟶   7
10 − 1           ⟶   9
4 <= 3           ⟶   false
(4/3) * 3 = 4    ⟶   true  "la igualdad es un simple mensaje binario, las fracciones
      son exactas"
(3/4) == (3/4)   ⟶   false  "dos fracciones iguales no son el mismo objeto"
```

> Los mensajes binarios son mensajes que requieren exactamente un argumento *y* cuyo selector está compuesto por una secuencia de los caracteres +, −, *, /, &, =, >, |, <, ∼, y @. −− no está permitido. Siguen el modelo sintáctico:
> receptor **selector** argumento

Mensajes Keyword

Los mensajes keyword son mensajes que requieren uno o más argumentos y cuyo selector consiste en una o más palabras clave cada una terminada en dos puntos (:). Los mensajes keyword siguen el modelo sintáctico: receptor **selectorPalabraUno:** argUno **palabraDos:** argDos.

Cada palabra clave toma un argumento. Entonces r:g:b: es un método con tres argumentos, playFileNamed: y at: son métodos con un argumento, y at:put: es un método con dos argumentos. Para crear una instancia de la clase Color se puede utilizar el método r:g:b: como en Color r: 1 g: 0 b: 0, que crea el color rojo. Nota que los dos puntos son parte del selector.

> En Java o C++, la invocación al método de Smalltalk Color r: 1 g: 0 b: 0 sería escrita Color.rgb(1, 0, 0).

```
1 to: 10                    ⟶   (1 to: 10)  "crea un intervalo"
Color r: 1 g: 0 b: 0        ⟶   Color red  "crea un nuevo color"
12 between: 8 and: 15       ⟶   true

nums := Array newFrom: (1 to: 5).
nums at: 1 put: 6.
nums    ⟶    #(6 2 3 4 5)
```

> Los mensajes keyword son mensajes que requieren uno o más argumentos. Su selector consiste en una o más palabras clave cada una terminada con dos puntos (:). Siguen el modelo sintáctico:
> receptor **selectorPalabraUno:** argUno **palabraDos:** argDos

4.3 Composición de mensajes

Los tres tipos de mensajes tienen diferente precedencia, lo cual permite que sean compuestos de manera elegante.

1. Los mensajes unarios son siempre enviados primero, luego los mensajes binarios y finalmente los mensajes keyword.
2. Los mensajes entre paréntesis son enviados antes que cualquier otro tipo de mensaje.
3. Los mensajes del mismo tipo son evaluados de izquierda a derecha.

Estas reglas llevan a un orden de lectura muy natural. Ahora bien, si deseas asegurarte que tus mensajes sean enviados en el orden que deseas puedes agregar paréntesis como se muestra en la Figure 4.3. En esta figura, el mensaje yellow es un mensaje unario y el mensaje color: un mensaje keyword, por lo tanto el envío de Color yellow se realiza primero. De todos modos, dado que los envíos de mensajes entre paréntesis se realizan primero, colocando paréntesis (innecesarios) alrededor de Color yellow simplemente enfatiza la precedencia de dicho mensaje. El resto de la sección ilustrará cada uno de estos puntos.

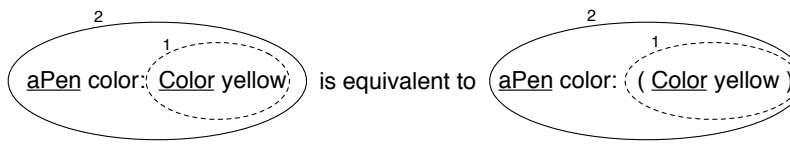


Figure 4.3: Los mensajes unarios se envían primero, así que se envía Color yellow. Esto devuelve un objeto color, el cual es pasado como argumento del mensaje aPen color:.

Unarios > Binarios > Keyword

Los mensaje unarios se envían primero, a continuación los mensajes binarios, y finalmente los mensajes keyword. También se dice que los mensajes unarios tienen mayor prioridad que los otros tipos de mensaje.

> **Regla Uno.** Los mensaje unarios se envían primero, luego los mensajes binarios, y finalmente los mensajes keyword.
>
> Unarios > Binarios > Keyword

Como muestran estos ejemplos, las reglas de sintaxis de Smalltalk generalmente aseguran que los envíos de mensajes puedan ser leídos de manera natural:

```
1000 factorial / 999 factorial  ⟶  1000
2 raisedTo: 1 + 3 factorial     ⟶  128
```

Desafortunadamente estas reglas son algo simplistas para envíos de mensajes aritméticos, entonces es necesario introducir paréntesis cuando se desea imponer cierta prioridad sobre los operadores binarios:

```
1 + 2 * 3      ⟶   9
1 + (2 * 3)    ⟶   7
```

El siguiente ejemplo, el cual es un poco mas complejo (!), ofrece una buena ilustración de que aún las expresiones complicadas en Smalltalk pueden ser leídas de manera natural:

```
[:aClass | aClass methodDict keys select: [:aMethod | (aClass >> aMethod) isAbstract ]]
      value: Boolean    ⟶    an IdentitySet(#or: #| #and: #& #ifTrue: #ifTrue:ifFalse:
      #ifFalse: #not #ifFalse:ifTrue:)
```

Aquí queremos saber cuáles métodos de la clase Boolean son abstractos[1]. Le pedimos a alguna clase pasada como argumento, aClass, las claves de su diccionario de métodos y seleccionamos aquellas claves asociadas a métodos que son abstractos, utilizando dichas claves para solicitar a la clase el objeto método correspondiente y preguntarle a éste si es abstracto o no. A continuación ligamos al argumento aClass el valor concreto Boolean. Solamente necesitamos paréntesis para enviar el mensaje binario >>, el cual solicita un método a una clase, antes de enviar el mensaje unario isAbstract a dicho método. El resultado nos muestra qué métodos deben ser implementados por las subclases concretas de Boolean: True y False.

Ejemplo. En el mensaje aPen color: Color yellow, encontramos el mensaje *unario* yellow enviado a la clase Color y el mensaje *keyword* color: enviado a aPen. Los mensajes unarios se envían primero, en consecuencia Color yellow es enviado al comienzo de la evaluación (1). El envío realizado devuelve un objeto color, el cual es pasado como argumento del mensaje aPen color: aColor (2) como se muestra en el example 4.1. La Figure 4.3 muestra gráficamente cómo son enviados los mensajes.

Example 4.1: *Descomponiendo la evaluación de* aPen color: Color yellow

```
      aPen color: Color yellow
(1)               Color yellow      "mensaje unario enviado primero"
                  ⟶   aColor
(2)   aPen color: aColor            "mensaje keyword enviado luego"
```

Ejemplo. En el mensaje aPen go: 100 + 20, encontramos el mensaje *binary* + 20 y el mensaje *keyword* go:. Los mensajes binarios se envían antes que los

[1] De hecho, también podríamos haber escrito la expresión equivalente pero más simple: Boolean methodDict select: #isAbstract thenCollect: #selector

mensajes keyword así que 100 + 20 es enviado primero (1): el mensaje + 20 es enviado al objeto 100 retornando el número 120. Entonces el mensaje aPen go: 120 es enviado con 120 como argumento (2). El example 4.2 muestra como es ejecutado el todo el envío de mensajes.

Example 4.2: *Descomponiendo* aPen go: 100 + 20

```
    aPen go: 100 + 20
(1)         100 + 20        "mensaje binario enviado primero"
                ⟶    120
(2) aPen go: 120            "luego el mensaje keyword"
```

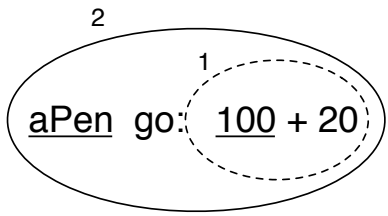


Figure 4.4: Los mensajes binarios se envían antes que los mensajes keyword.

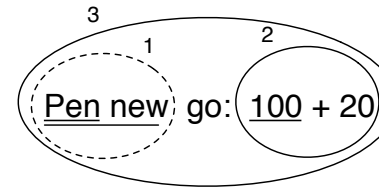


Figure 4.5: Descomponiendo Pen new go: 100 + 20

Ejemplo. Como ejercicio te proponemos descomponer la evaluación del mensaje Pen new go: 100 + 20 el cual está compuesto por un mensaje unario, un mensaje keyword y un mensaje binario (ver Figure 4.5).

Los paréntesis primero

> **Regla Dos.** Los mensajes entre paréntesis se envían antes que los otros mensajes.
>
> (Mensaje) > Unarios > Binarios > Keyword

```
1.5 tan rounded asString = (((1.5 tan) rounded) asString)   ⟶   true   "no se
    necesitan par\'entesis"
3 + 4 factorial   ⟶   27   "(no es 5040)"
(3 + 4) factorial   ⟶   5040
```

Aquí necesitamos los paréntesis para forzar el envío de lowMajorScaleOn: antes del envío de play.

```
(FMSound lowMajorScaleOn: FMSound clarinet) play
"(1) enviar el mensaje clarinet a la clase FMSound para crear un sonido de clarinete.
```

```
(2) enviar este sonido a FMSound como argumento del mensaje keyword
      lowMajorScaleOn:.
(3) reproducir el sonido resultante."
```

Ejemplo. El mensaje (65@325 extent: 134 @ 100) center retorna el centro de un rectángulo cuya esquina superior izquierda es $(65, 325)$ y cuyo tamaño es 134×100. El example 4.3 muestra cómo el mensaje es descompuesto y enviado. Primero es enviado el mensaje entre paréntesis: contiene los mensajes binarios 65@325 y 134@100 que son enviados primero y retornan objetos punto, y el mensaje keyword extent: que es enviado a continuación y retorna un rectángulo. Finalmente es enviado al rectángulo el mensaje unario center y entonces un nuevo punto es retornado por este. Evaluar el mensaje sin paréntesis llevaría a un error porque el objeto 100 no comprende el mensaje center.

Example 4.3: *Ejemplo de Paréntesis.*

```
    (65 @ 325 extent: 134 @ 100) center
(1) 65@325                          "binario"
      ⟶   aPoint
(2) 134@100                         "binario"
      ⟶   anotherPoint
(3) aPoint extent: anotherPoint     "keyword"
      ⟶   aRectangle
(4) aRectangle center               "unario"
      ⟶   132@375
```

De izquierda a derecha

Ahora sabemos cómo son manejados los mensajes de distintos tipos o prioridades. La pregunta final que resta por ser respondida es cómo son enviados los mensajes con la misma prioridad. Son enviados de izquierda a derecha. Debes notar que ya viste este comportamiento en example 4.3 donde los dos mensajes (@) de creación de puntos fueron enviados primero.

> **Regla Tres.** Cuando los mensajes son del mismo tipo, el orden de evaluación es de izquierda a derecha.

Ejemplo. En los envíos de los mensajes Pen new down todos los mensajes son unarios, así que el primero de la izquierda, Pen new, se envía primero. Este mensaje retorna un nuevo objeto lápiz al cual se le envía el segundo mensaje down, como se muestra en Figure 4.6.

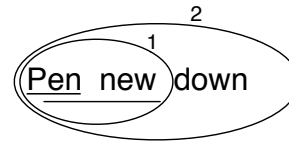
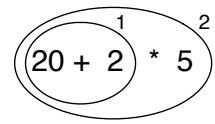


Figure 4.6: Decomponiendo Pen new down



Inconsistencias aritméticas

Las reglas de composición de mensajes son simples pero producen inconsistencias en la ejecución de envíos de mensajes aritméticos expresados en mensajes binarios. Aquí veremos algunas situaciones comunes en las cuales el agregado de paréntesis es necesario para subsanar dichas inconsistencias.

```
3 + 4 * 5        ⟶   35   "(no resulta 23)  Los mensajes binarios se env\'ian de
      izquierda a derecha"
3 + (4 * 5)      ⟶   23
1 + 1/3          ⟶   (2/3)   "no resulta 4/3"
1 + (1/3)        ⟶   (4/3)
1/3 + 2/3        ⟶   (7/9)   "no resulta 1"
(1/3) + (2/3)    ⟶   1
```

Ejemplo. En los envíos de mensajes 20 + 2 * 5, están presenta solamente los mensajes binarios + y *. En Smalltalk no hay prioridades específicas para las operaciones + y *. Son sólo mensajes binarios, entonces * no tiene prioridad por sobre +. Aquí el mesaje mas a la izquierda + es enviado primero (1) y luego se envía al resultado el mensaje * como se muestra en example 4.4.

Example 4.4: *Descomponiendo* 20 + 2 * 5

```
"Debido a que no hay prioridades entre los mensajes binarios, el mensaje de la
      izquierda + es evaluado primero, aun cuando para las reglas de la aritm\'etica el *
      deber\'ia ser enviado antes."

    20 + 2 * 5
(1) 20 + 2   ⟶   22
(2) 22     * 5   ⟶   110
```

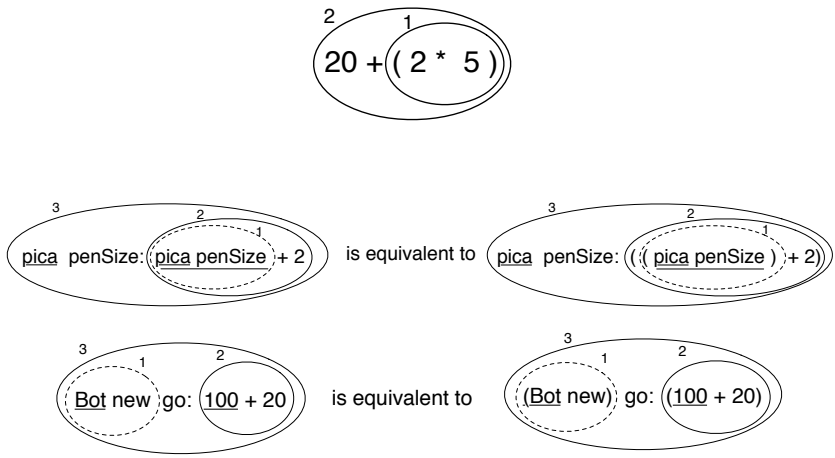


Figure 4.7: Mensajes equivalentes utilizando paréntesis.

Como se muestra en example 4.4 el resultado de este envío de mensajes no es 30 sino 110. Este resultado es quizás inesperado pero es directamente consistente con las reglas utilizadas para enviar mensajes. De alguna manera este es un precio a pagar por la simplicidad de modelo de Smalltalk. Para obtener el resultado correcto, deberíamos utilizar paréntesis. Los mensajes se evalúan primero cuando son rodeados con paréntesis. Entonces el envío de los mensajes 20 + (2 * 5) retornará el resultado correcto como se muestra en example 4.5.

Example 4.5: *Descomponiendo* 20 + (2 * 5)

```
"Los mensajes rodeados con par\'entesis son evaluados primero, en consecuencia * se
      env\'ia antes que + lo cual produce el comportamiento correcto."

   20 + (2 * 5)
(1)      (2 * 5)   ⟶   10
(2) 20 + 10        ⟶   30
```

> En Smalltalk, los operadores aritméticos como + y * no poseen diferente prioridad. + y * son simples mensajes binarios, de lo cual se sigue que * no posee prioridad por sobre +. Se deben utilizar los paréntesis para obtener los resultados deseados.

Se debe notar que la primera regla que determina que los mensajes unarios sean enviados antes que los binarios y que los keyword evita la necesidad

Precedencias implícita y su equivalentes con paréntesis explícitos	
aPen color: Color yellow	aPen color: (Color y
aPen go: 100 + 20	aPen go: (100 + 20)
aPen penSize: aPen penSize + 2	aPen penSize: ((aP
2 factorial + 4	(2 factorial) + 4

Figure 4.8: Envíos de mensajes y equivalencias usando paréntesis.

de utilizar paréntesis explícitos a su alrededor. La Table 4.8 muestra envíos de mensajes escritos siguiendo estas reglas y a su lado los envíos de mensajes equivalentes si las reglas no existieran. Ambos envíos de mensajes resultan en el mismo efecto o retornan el mismo valor.

4.4 Sugerencias para identificar los mensajes keyword

Frecuentemente los principiantes tienen problemas para comprender cuándo es necesario utilizar paréntesis. Veamos cómo son reconocidos los mensajes keyword por el compilador.

Paréntesis o no?

Los caracteres [,], (y) delimitan diferentes áreas. En de cada una de dichas áreas, un mensaje keyword es la secuencia más larga de palabras terminadas por : que no sea cortada por los caracteres . o ;. Cuando los caracteres [,], (y) rodean un conjunto de palabras terminadas con dos puntos, estas palabras participan en el mensaje keyword *local* al área definida.

En este ejemplo, hay dos mensajes keyword diferentes: rotatedBy:magnify:smoothing: y at:put:.

```
aDict
  at: (rotatingForm
        rotateBy: angle
        magnify: 2
        smoothing: 1)
  put: 3
```

> Los caracteres [,], (y) delimitan diferentes áreas. Dentro de cada una de dichas áreas, un mensaje keyword está compuesto por la secuencia más larga de palabras terminadas por : que no sea cortada por los caracteres ., o ;. Cuando los caracteres [,], (y) rodean algunas palabras terminadas por dos puntos, estas palabras participan en el mensaje keyword local al área definida.

Consejos. Si tienes problemas con estas reglas de precedencia, puedes comenzar simplemente poniendo paréntesis cada vez que quieras distinguir dos mensajes que tengan la misma precedencia.

El siguiente trozo de código no requiere paréntesis porque el envío de mensaje x isNil es unario, por lo tanto es enviado antes que el mensaje keyword ifTrue:.

```
(x isNil)
  ifTrue: [...]
```

El siguiente trozo de código requiere paréntesis porque los mensajes includes: y ifTrue: son ambos mensajes keyword.

```
ord := OrderedCollection new.
(ord includes: $a)
  ifTrue: [...]
```

Sin los paréntesis el mensaje desconocido includes:ifTrue: sería enviado a la colección!

Cuándo usar [ ] o ( )

Puedes tener problemas para comprender cuándo utilizar corchetes en lugar de paréntesis. El principio básico es el que indica que debes utilizar [ ] cuando no sepas cuántas veces, potencialmente cero, una expresión deberá ser evaluada. La notación [*expresión*] creará un cierre de bloque (*i.e.*, un objeto) de *expresión*, el cual podrá ser evaluado cualquier número de veces (posiblemente cero), dependiendo del contexto. Aquí se debe notar que dicha expresión puede ser tanto un envío de mensaje, como una variable, una asignación o un bloque.

Por lo tanto las ramas condicionales de ifTrue: o ifTrue:ifFalse: requieren bloques. Siguiendo el mismo principio tanto el receptor como el argumento de un mensaje whileTrue: requieren el uso de corchetes puesto que no sabemos cuántas veces deberán ser evaluados ni uno ni el otro.

Los paréntesis, por otro lado, solo afectan el orden del envío de mensajes. Entonces en (*expresión*), la *expresión* será *siempre* evaluada exactamente una vez.

```
[ x isReady ] whileTrue: [ y doSomething ]   "tanto el receptor como el argumento deben
    ser bloques"
4 timesRepeat: [ Beeper beep ]               "el argumento es evaluado mas de una vez,
    entonces debe ser un bloque"
(x isReady) ifTrue: [ y doSomething ]        "el receptor es evaluado una vez, entonces
    no es un bloque"
```

4.5 Secuencias de expresiones

Las expresiones (*i.e.*, envíos de mensajes, asignaciones...) separadas por puntos son evaluadas secuencialmente. Nota que no hay un punto entre una definición de variable y la siguiente expresión. El valor de una secuencia es el valor de la última expresión. Los valores retornados por todas las expresiones exceptuando la última son ignorados. Debes notar también que el punto es un separador y no un terminador. En consecuencia un punto final es solo opcional.

```
| box |
box := 20@30 corner: 60@90.
box containsPoint: 40@50    ⟶    true
```

4.6 Mensajes en cascada

Smalltalk ofrece una manera de enviar múltiples mensajes al mismo receptor utilizando un punto y coma (;). Esto es llamado *cascada* en la jerga de Smalltalk.

Expression Msg1 ; Msg2

```
Transcript show: 'Pharo is '.
Transcript show: 'fun '.
Transcript cr.
```

es equivalente a:

```
Transcript
  show: 'Pharo is';
  show: 'fun ';
  cr
```

Debes notar que el objeto que recibe la cascada puede también ser el resultado de un envío de mensaje en sí mismo. De hecho el receptor de todos los mensajes de una cascada es el receptor del primer mensaje de la misma. En

el siguiente ejemplo, el primer mensaje de la cascada es setX:setY: dado que está seguido por una cascada. El receptor del mensaje en cascada setX:setY: es el objeto punto que acaba de ser creado como resultado de la evaluación de Point new, y *no* Point. El mensaje subsiguiente isZero es enviado al mismo receptor.

```
Point new setX: 25 setY: 35; isZero  ⟶  false
```

4.7 Resumen del capítulo

- Un mensaje es siempre enviado a un objeto llamado el *receptor* que puede ser el resultado de otro mensaje enviado.
- Los mensajes unarios son mensajes que no requiren argumento.
  Son de la forma receptor **selector**.
- Los mensajes binarios son mensajes que involucran dos objetos, el receptor y otro objeto *y* cuyo selector está compuesto de uno o más de caracteres de la siguiente lista: +, −, *, /, |, &, =, >, <, ~, y @. Son de la forma: receptor **selector** argumento
- Los mensajes keyword son mensajes que involucran a más de un objeto y que contienen al menos un caracter dos puntos (:).
  Son de la forma: receptor **selectorPalabraUno:** argumentoUno **palabraDos:** argumentoDos
- **Regla Uno.** Los mensajes unarios son enviados primero, luego los binarios y finalmente los keyword.
- **Regla Dos.** Los mensajes en paréntesis son enviados antes que cualquier otro.
- **Regla Tres.** Cuando los mensajes enviados son del mismo tipo, el orden de evaluación de de izquierda a derecha.
- En Smalltalk, tradicional, la aritmética de operadores como + y * tienen la misma prioridad. + y * son solo mensaje binarios, por lo tanto * no tienen prioridad sobre +. Debes usar paréntesis para obtener un resultado diferente.

Part II

Developing in Pharo

Chapter 5

El modelo de objetos de Smalltalk

El modelo de programación de Smalltalk es simple y uniforme: todo es un objeto, y los objetos se comunican entre si únicamente mediante el envío de mensajes.

Sin embargo, esta simplicidad y uniformidad puedes ser una fuente de dificultad para aquellos programadores que están acostumbrados a otros lengujes. En este capítulo, presentamos los conceptos claves del modelo de objetos de Smalltalk; en particular discutimos las concecuencias de representar clases como objetos.

5.1 Las reglas del modelo

El modelo de objetos de Smalltalk está basado en un conjunto de reglas simples que son aplicadas *de manera uniforme*. Dichas reglas son las siguientes:

Rule 1. Todo es un objeto.

Rule 2. Todo objeto es instancia de una clase.

Rule 3. Toda clase tiene una superclase.

Rule 4. Todo ocurre mediante el envío de mensajes.

Rule 5. El Method lookup sigue la cadena de herencia.

Veamos cada una de estas reglas en detalle.

5.2 Todo es un Objeto

La frase "todo es un objeto" es altamente contagiosa. Después de poco tiempo trabajando con Smalltalk, te sorprenderás de cómo esta regla simplifica todo lo que haces. Los números enteros, por ejemplo, son objetos verdaderos, por lo tanto puedes enviarles mensajes, tal como lo haces con cualquier otro objeto.

```
3 + 4            ⟶   7   ``env\'ia '+ 4' to 3, yielding 7"
20 factorial     ⟶   2432902008176640000   "send factorial, yielding a big number"
```

La representación de `20 factorial` es ciertamente diferente de la representación de `7`, pero dado que ambos dos son objetos, ninguno de los dos códigos — ni siquiera la implementación de `factorial` — necesita saber acerca de esto.

Quizás, la consecuencia fundamental de esta regla es lo siguiente:

> Las clases son objetos también.

Además, las clases no son objetos de segundo orden: realmente son objetos de primer orden a los cuales les puedes enviar mensajes, inspeccionar, etc. Esto significa que Pharo es un sistema verdaderamente reflectivo, lo que da un gran poder de expresión a los desarrolladores.

Más produndo en la implementación, hay tres tipos diferentes de objetos. Existen (1) objetos ordinarios con variables de instancia que son pasadas por referencia, existen (2) *small integers* que son pasados por valor, y existen objetos indexables como arrays que contienen una porción continua de memoria. La belleza de Smalltalk es que normalmente no tienes que preocuparte por las diferencias de estos tres tipos de objetos.

5.3 Todo objeto es instancia de una clase

Todo objeto tiene una clase; puedes averiguar cuál enviándole el mensaje `class`.

```
1 class              ⟶   SmallInteger
20 factorial class   ⟶   LargePositiveInteger
'hello' class        ⟶   ByteString
#(1 2 3) class       ⟶   Array
(4@5) class          ⟶   Point
Object new class     ⟶   Object
```

Una clase define la *estructura* de sus instancias mediante variables de instancia, y el *comportamiento* mediante métodos. Cada método tiene un nombre, llamado su *selector*, que es único dentro de su clase.

Dado que *las clases son objetos*, y que *todo objeto es instancia de una clase*, entonces se deduce que las clases también tienen que ser instancias de clases. Una clase cuyas instancias son clases, se llama *metaclase*. Siempre que creas una clase, el sistema crea automáticamente la metaclase. La metaclase define la estructura y el comportamiento de la clase que es su instancia. El 99% de las veces no necesitarás pensar en metaclases, y posiblemente, las ignores. (Tendremos una mirada más de cerca a metaclases en Chapter 13.)

Variables de instancia

Las variables de instancia en Smalltalk son privadas a la instancia en si misma. Esto es un contraste con Java y C++, que permiten que las variables de instancia (también conocidas como "campos", "atributos" o incluso "colaboradores internos") sean accedidas por otras instancias de la misma clase. Decimos que el *límite del encapsulamiento* de los objetos en Java y C++ es la clase, mientras que en Smalltalk es la instancia.

En Smalltalk dos instancias de la misma clase no pueden acceder a las variables de instancia de la otra a menos que la clase defina "métodos de acceso".

No existe sintaxis alguna del lenguaje que provea acceso directo a las variables de instancia de cualquier otro objeto. (En realidad, un mecanismo llamado reflexión sí provee una forma de pedirle a otro objeto el valor de sus variables de instancia; la meta-programación es usada para escribir herramientas como el inspector de objetos, cuyo único propósito es mirar internamente otros objetos.)

Las variables de instancia pueden ser accedidas por nombre desde cualquier método de instancia de la clase que las define y también desde los métodos definidos en sus subclases. Esto significa que las variables de instancia de Smalltalk son similares a las variables *protegidas* de C++ y Java. No obstante, preferimos decir que son privadas, pues en Smalltalk es considerado un mal estilo acceder directamente a las variables de instancia desde una subclase.

Ejemplo

El método `Point»dist:` (method 5.1) computa la distancia entre el receptor y otro punto. Las variabes de instancia `x` e `y` del receptor son accedidas directamente por el cuerpo de método. Sin embargo, las variables de instancia del otro punto deben ser accedidas enviándole los mensajes `x` e `y`.

Method 5.1: *la distancia entre dos puntos*

```
Point»dist: aPoint
   "Answer the distance between aPoint and the receiver."
   | dx dy |
   dx := aPoint x − x.
   dy :=  aPoint y − y.
   ↑ ((dx * dx) + (dy * dy)) sqrt
```

```
1@1 dist: 4@5    ⟶    5.0
```

La razón clave para preferir encapsulamiento basado en instancias sobre encapsulamiento basado en clases, es que permite que diferentes implementaciones de la misma abstracción puedan coexistir.

Por ejemplo, el método point»dist: no necesita saber o preocuparse si el argumento aPoint es una instancia de la misma clase que el receptor o no. El objeto del argumento podría estar representado en coordinadas polares, o como un registro en una base de datos, o en otra computadora en un sistema distribuido; mientras pueda responder a los mensajes x e y, el código de method 5.1 va a seguir funcionando correctamente.

Métodos

Todos los métodos son públicos.[1] Los métodos son agrupados en protocolos que indican su intención. Algunos nombres de protocolos comunes han sido establecidos por convención, por ejemplo, *accessing* para todos los métodos de acceso, e *initialization* para crear un estado concistente inicial del objeto. El protocolo *private* se usa a veces para agrupar métodos que no deberían ser vistos desde afuera del objeto. Sin embargo, nada previene que le puedas enviar un mensaje que está implementado como "privado".

Los métodos pueden acceder a todas las variables de instancia del objeto. Algunos programadores Smalltalk prefieren acceder a las variables de instancia únicamente mediante métodos de acceso. Dicha práctica tiene cierto valor, pero también impacta en la interfaz de las clases, y pero aún, expone estado privado al mundo.

Del lado de instancia y del lado de clase

Dado que las clases son objetos, también tienen sus propias variables de instancia y sus propios métodos. Llamamos a ellos, *variables de instancia de clase* y *métodos de clase*, pero en realidad no son diferentes a las variables de

[1] Bueno, en realidad, casi todos. En Pharo, los métodos cuyos selectores empiezan con el prefijo pvt son privados: Un mensaje pvt puede ser enviado *solamente* a self. No obstante, los métodos pvt no son muy usados.

instancia y métodos ordinarios: las variables de instancia de clase son sólo variables de instancia definidas por una metaclase, y los métodos de clase son simplemente métodos definidos por una metaclase.

Una clase y su metaclase son dos clases separadas, a pesar de que la primera es una instancia de la segunda. No obstante, esto es en gran medida irrelevante para el programador: debe concentrarse en definir el comportamiento de sus objetos y en las clases que los crean.

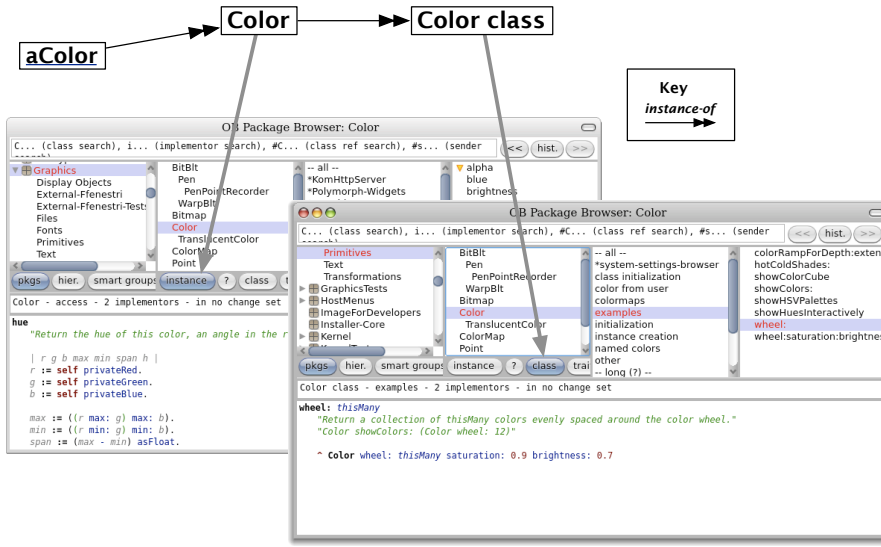


Figure 5.1: Navegando una clase y su metaclase.

Por esta razón, el navegador te ayuda a navegar ambas, la clase y su metaclase, como si fueran una sola cosa con dos pestañas: "del lado de instancia" y "del lado de clase", como se muestra en Figure 5.1.

Haciendo clic en el botón instance navegamos la clase Color, *i.e.*, puedes navegar los métodos que son ejecutados cuando se le envían mensajes a una instancia de Color, como por ejemplo el color azul. Presionando el botón class navegamos la clase Color class, *i.e.*, se pueden ver los métodos que serán ejecutados cuando se le envien mensajes a la clase Color propiamente dicha. Por ejemplo, Color blue envía el mesaje blue a la clase Color. Por lo tanto, encontrarás el método blue definido en el lado de clase de Color, no del lado de instancia.

```
aColor := Color blue.          "Class side method blue"
aColor        ⟶   Color blue
aColor red    ⟶   0.0      "Instance side accessor method red"
aColor blue   ⟶   1.0      "Instance side accessor method blue"
```

Una clase se define completando el esqueleto propuesto en el lado de instancia. Cuando se acepta esta plantilla, el sistema crea no sólo la clase definida, pero también la metaclase asociada.

Puede navegar la metaclase haciendo clic en el botón class. La única parte de la plantilla de creación de la metaclase que tiene sentido editar directamente es la lista de nombres de variables de instancia.

Una vez que una clase fue creada, haciendo clic en el botón instance podemos editar y navegar los métodos que serán poseídos por instancias de esa clase (y de sus subclases). Por ejemplo, podemos ver en la Figure 5.1 que el método hue está definido en el lado de instancia de la clase Color. Por el contrario, el bóton class permite navegar y editar la metaclase (en este caso Color class).

Métodos de clase

Los métodos de clase pueden ser muy útiles; puedes navegar Color class para ver unos buenos ejemplos. Verás que hay dos tipos de métodos definidos en una clase: aquellos que crean instancias de la clase, como Color class» blue y aquellos que llevan a cabo una función de utilidad, como Color class »showColorCube. Esto es lo típico, aunque ocasionalmente encontrarás métodos de clase usados para otra cosa.

Es conveniente poner los métodos de utilidad del lado de clase porque pueden ser ejecutados sin tener que crear ningún objeto primero. De hecho, la mayoría de ellos tendrán un comentario cuyo objetivo es hacer más fácil su ejecución.

Verás el efecto de ejecutar este método. (Seleccione World ▷ restore display (r) para deshacer los cambios.)

Para aquellos familiarizados con Java y C++, los métodos de clase parecieran ser similares a los métodos estáticos. Sin embargo, la uniformidad de Smalltalk implica que de alguna forma son distintos: mientras que los métodos estáticos de Java son realmente simples procedimientos resueltos estáticamente, los métodos de clase de Smalltalk son métodos despachados dinamicamente. Esto significa que la herencia, sobre-escritura y mensajes enviados con super funcionan en los métdos de clase en Smalltalk, mientras que no funcionan con los métodos estáticos de Java.

Variables de instancia de clase

Con las variables de instancia ordinarias, todas las instancias de una clase tienen el mismo conjunto de nombres de variables, y las instancias de sus subclases heredan esos nombres; no obstante, cada instancia tiene su propio conjunto privado de valores. La historia es exactamente la misma con las variables de instancia de clase: cada clase tiene sus propias variables de instancia de clase privadas. Una subclase heredará esas variables de instancia de clase, *pero tendrá sus propias copias de dichas variables*. Así cómo los objetos no comparten sus variables de instancia, tampoco las clases ni sus subclases comparten las variables de instancia de clase.

Una variable de instancia de clase llamada contador se podría usar por ejemplo, para llevar la cuenta de cuantas instancias se crean de una clase dada. De todos modos, cada subclase tendrá su propia variable count, por lo tanto, las instancias de las subclases serían contadas por separado.

Ejemplo: las variables de instancia de clase no son compartidas con sus subclases. Supongamos que definimos las classes Dog (perro) y Hyena (hiena), donde Hyena hereda la variable de instancia de clase contador de Dog.

Class 5.2: *Perros y hienas*

```
Object subclass: #Dog
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE−CIV'

Dog class
   instanceVariableNames: 'count'

Dog subclass: #Hyena
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE−CIV'
```

Ahora supongamos que creamos métodos de clase en Dog para inicializar su variable count a 0.

Method 5.3: *Manteniendo la cuenta de perros nuevos*

```
Dog class»initialize
   super initialize.
   count := 0.

Dog class»new
   count := count +1.
```

```
  ↑ super new

Dog class»count
  ↑ count
```

Ahora cuando creamos un nuevo perro su contador es incrementado, y lo mismo para cada hiena, pero son contados por separado:

```
Dog initialize.
Hyena initialize.
Dog count      ⟶   0
Hyena count    ⟶   0
Dog new.
Dog count      ⟶   1
Dog new.
Dog count      ⟶   2
Hyena new.
Hyena count    ⟶   1
```

Note también que las variables de instancia de clase son privadas a la clase extactamente de la misma forma en que las variables de instancia son privadas a la instancia. Dado que las clases y sus instancias son diferentes objetos, existen las siguientes inmediatas consecuencias:

> Una clase no tiene acceso a las variables de instancia de sus propias instancias.

> Una instancia de una clase no tiene acceso a las variables de instancia de clase de su propia clase.

Por esta razón, los métodos de inicialización de instancia deber ser definidos siempre en el lado de instancia del navegador —el lado de clase del navegador no tiene acceso a las variables de instancia ¡por lo tanto no las puede inicializar!

De manera similar, las instancias puede acceder a las variables de instancia de clase solamente de forma indirecta, enviando un mensaje de acceso a sus clases.

Java no tiene nada equivalente a las variables de instancia de clase. Las variables estáticas de Java y C++ son más parecidas a las variables de clase de Smalltalk, que discutiremos en la Section 5.7: todas las subclases y todas sus instancias comparten la misma variable estática.

Ejemplo: Definiendo un Singleton. El patrón Singleton[2] provee un típico ejemplo del uso de variables de instancia de clase y de métodos de clase. Imagine que quisiéramos implementar la clase WebServer y usar el patrón Singleton para asegurarnos que tenga una única instancia.

haciendo clic en el botón instance del navegador, definimos la clase WebServer como sigue (class 5.4).

Class 5.4: *Una clase singleton*

```
Object subclass: #WebServer
   instanceVariableNames: 'sessions'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Web'
```

Luego, haciendo clic en el botón class, agregamos la variable de instancia uniqueInstance del lado de clase

Class 5.5: *El lado de clase de la clase singleton*

```
WebServer class
   instanceVariableNames: 'uniqueInstance'
```

La consequencia de esto es que la clase WebServer ahora tiene otra variable de instancia, además de las variables de instancia que hereda, como superclass y methodDict.

Ahora podemos definir un método de clase llamado uniqueInstance como se muestra en el method 5.6. Este método primero chequea si uniqueInstance ha sido inicializada o no. Si no lo fue, el método crea una instancia y la asigna a la variable de instancia de clase uniqueInstance. Finalmente el valor de uniqueInstance es retornado. Como uniqueInstance es una variable de instancia de clase, éste método puede acceder a ella directamente.

Method 5.6: *uniqueInstance (en el lado de clase)*

```
WebServer class»uniqueInstance
   uniqueInstance ifNil: [uniqueInstance := self new].
   ↑ uniqueInstance
```

La primera vez que WebServer uniqueInstance es ejecutado, una instancia de la clase WebServer será creada y asignada a la variable uniqueInstance. La próxima vez, la instancia creada previamente será retornada, en lugar de crear una nueva.

Note que el código de creación de instancias adentro del condicional en el method 5.6 está escrito como self new y no como WebServer new. ¿Cuál

[2]Sherman R. Alpert, Kyle Brown and Bobby Woolf, *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998, ISBN 0–201–18462–1.

es la diferencia? Dado que el método uniqueInstance está definido en la clase WebServer, podría pensarse que ambas son iguales. De hecho, hasta que alguien cree una subclase de WebServer, son iguales. Pero supongamos que ReliableWebServer es una subclase de WebServer, y hereda el método uniqueInstance. Esperaríamos claramente que ReliableWebServer uniqueInstance responda un ReliableWebServer. Usando self nos aseguramos de que así sea, pues será ligado con la correspondiente clase. Note también que WebServer y ReliableWebServer van a tener, cada uno, sus propias variables de instancia de clase llamadas uniqueInstance. Esas dos variables van a tener obviamente distintos valores.

5.4 Toda clase tiene una superclase

Cada clase en Smalltalk hereda su comportamiento y la descripción de su estructura de una única *superclase*. Esto significa que Smalltalk tiene herencia simple.

```
SmallInteger superclass  ⟶  Integer
Integer superclass       ⟶  Number
Number superclass        ⟶  Magnitude
Magnitude superclass     ⟶  Object
Object superclass        ⟶  ProtoObject
ProtoObject superclass   ⟶  nil
```

Tradicionalmente la raiz de la jerarquía de herencia en Smalltalk es la clase Object (dado que todo es un objeto). En Pharo, la raiz es en realidad una clase llamada ProtoObject, aunque normalmente no prestarás atención a esta clase. ProtoObject encapsula el conjunto de mensajes mínimos que todos los objetos *deben* tener. Sin embargo, la mayoría de las clases heredan de Object, quien define varios mensajes adicionales que casi todos los objetos deberían entender y responder. A menos que tengas una muy buena razón para hacer lo contrario, cuando crees clases para aplicaciones deberían normalmente heredar de Object, o de una de sus subclases.

 *Una nueva clase es creada normalmente enviando el mensaje* subclass: instanceVariableNames: ... *a una clase existente. Hay también algunos otros métodos para crear clases. Puedes observar el protocolo* Kernel-Classes ▷ Class ▷ subclass creation *para ver cuales son.*

Aunque Pharo no provee múltiple herencia, sí soporta una mecanismo llamado *trait*s para compartir comportamiento entre clases no relacionadas. Traits son colecciones de métodos que pueden ser reutilizados por múltiples cases que no están relacionadas por la herencia. Usando traits uno puede compartir código entre diferentes clases sin duplicación de código.

Métodos abstractos y clases abstractas

Una clase abstracta es una clase que existe para ser subclasificada, más que instanciada. Una clase abstracta es usualmente incompleta, en el sentido de que no define todos los métodos que usa. Todos esos métodos que le faltan — aquellos que otros métodos asumen, pero que no están definidos en si mismos — son llamados métodos abstractos.

Smalltalk no tiene una sintaxis dedicada para especificar que un método o clase es abstracta. Por convención, el cuerpo de un método abstracto consiste en la expresión self subclassResponsibility. Este es un conocido "Marker Method", e indica que las subclases tienen la responsabilidad de definir una versión concreta del método. Los métodos self subclassResponsibility deberían ser siempre sobreescritos, y por lo tanto nunca deberían ser ejecutados. Si te olvidas de sobreescribir alguno, y es ejecutado, una excepción será lanzada.

Una clase es considerada abstracta si al menos uno de sus métodos es abstracto. En realidad, nada te previene de crear una instancia de una clase abstracta: todo va a andar bien hasta que un método abstracto sea invocado.

Ejemplo: la clase Magnitude.

Magnitude es una clase abstracta que nos ayuda a definir objetos que pueden ser comparados con otros. Las subclases de Magnitude deberían implementar los métodos <, = and hash. Usando esos mensajes Magnitude define otros métodos como >, >=, <=, max:, min: between:and: y otros para comparar grafos de objetos. El método < es abstracto y definido como se muestra en method 5.7.

Method 5.7: Magnitude»<

```
Magnitude»< aMagnitude
    "Answer whether the receiver is less than the argument."
    ↑self subclassResponsibility
```

Por el contrario, el método >= es concreto; está definido en términos de <:

Method 5.8: Magnitude»>=

```
>= aMagnitude
    "Answer whether the receiver is greater than or equal to the argument."
    ↑(self < aMagnitude) not
```

Lo mismo sucede con los otros métodos de comparación.

Character es una subclase de Magnitude; sobreescribe el método subclassResponsibility de < con su propia versión de < (mirar method 5.9). Character también define los métodos = y hash; hereda de Magnitude los métodos >=, <=, ~= entre otros.

Method 5.9: Character»<

```
Character»< aCharacter
    "Answer true if the receiver's value < aCharacter's value."
    ↑self asciiValue < aCharacter asciiValue
```

Traits

Un *trait* es una colección de métodos que pueden ser incluidos en el comportamiento de una clase sin la necesidad de la herencia. Esto permite facilmente que las clases tengan una única superclase, pero que también puedan compartir métodos con otras clases con las que no están relacionadas.

Para definir un nuevo trait, simplemente remplaza la plantilla de creación de subclases, enviando un mensaje a la clase Trait.

Class 5.10: *Defining a new trait*

```
Trait named: #TAuthor
    uses: { }
    category: 'PBE−LightsOut'
```

Acá definimos el trait TAuthor en la categoría *PBE-LightsOut*. Este trait no *usa* ningún otro trait existente. En general siempre podemos especificar la expresión de composición de un trait, de otros traits para usar como parte del argumnento de palabra clave uses:.

Los traits pueden contener métodos pero no variables de instancia. Supongamos que queremos agregar un método author a varias clases, independientemente de la herencia. Podríamos hacerlo de la siguiente manera:

Method 5.11: *An author method*

```
TAuthor»author
    "Returns author initials"
    ↑ 'on'    "oscar nierstrasz"
```

Ahora podemos usar este trait en una clase que ya tiene su propia superclase, por ejemplo en la clase LOGame que definimos en Chapter 2. Simplemente modificamos la plantilla de creación de clases de LOGame para incluir la palabra clave uses: que especifica que TAuthor debería ser usado.

Class 5.12: *Using a trait*

```
BorderedMorph subclass: #LOGame
    uses: TAuthor
    instanceVariableNames: 'cells'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE−LightsOut'
```

Como es de esperarse, si ahora instanciamos un LOGame, dicho objeto reponderá al mensaje author de forma correcta.

```
LOGame new author  ⟶  'on'
```

Las expresiones de composición de Trait pueden combinar múltiples traits usando el operador +. En caso de conflictos (*i.e.*, múltiples traits definen métodos con el mismo nombre), estos pueden resolverse removiendo explícitamente estos métodos (con −), o redefiniendo or by redefining estos métodos en la clase o trait que estes definiendo. También es posible darle un *alias* a los métodos (con @), proporcionandoles así un nuevo nombre.

Los Traits son usados en el núcleo del sistema. Un buen ejemplo es la clase Behavior.

Class 5.13: Behavior *defined using traits*

```
Object subclass: #Behavior
  uses: TPureBehavior @ {#basicAddTraitSelector:withMethod:->
    #addTraitSelector:withMethod:}
  instanceVariableNames: 'superclass methodDict format'
  classVariableNames: 'ObsoleteSubclasses'
  poolDictionaries: ''
  category: 'Kernel−Classes'
```

Aquí vemos que al método addTraitSelector:withMethod: definido en el trait TPureBehavior se le ha dado el alias basicAddTraitSelector:withMethod:. Actualmente se esta agregando a los navegadores el soporte para traits.

5.5 Todo ocurre mediante el envío de mensajes

Esta regla captura la escencia de programar en Smalltalk.

En la programación procedural, la elección de que pieza del código se ejecutará cuando un procedimiento se llama es hecha por quien llama. El que llama elige el procedimiento o función se ejecutará *de forma estática*, por nombre.

En la programación orientada a objetos, nosotros *no* "llamamos métodos": nosotros "enviamos mensajes." La elección de terminología es significativa. Cada objeto tiene sus propias responsabilidades. No le *decimos* a un objeto qué hacer apicandole algún procedimiento. En su lugar, le *pedimos* a un objeto que haga algo por nosotros enviandole un mensaje. El mensaje *no* es una pieza de código: no es mas que un nombre y una lista de argumentos. El receptor luego decide como responder seleccionando su propio *método* para hacer lo que le pedimos.

Dado que objetos diferentes pueden terner métodos diferentes para responder al mismo mensaje, el método debe ser escojido *dinámicamente*, cuando el mensaje es recibido.

```
3 + 4          ⟶  7      "enviar el mensaje + con argumento 4 al integer  3"
(1@2) + 4      ⟶  5@6    "enviar el mensaje + con argumento 4 al point (1@2)"
```

Como consecuencia, podemos enviarle el *mismo mensaje* a diferentes objetos, cada uno de los cuales puede tener *su propio método* para responder al mensaje. No le decimos al SmallInteger 3 o al Point 1@2 como responder al mensaje + 4. Cada uno tiene su propio método para +, y responde a + 4 como corresponde.

Una de las consecuencias del modelo de envío de mensajes de Smalltalk's es que fomenta un estilo en el cual los objetos tienden a tener métodos muy pequeños y delegan tareas a otros objetos, en lugar de implementar métodos procedurales enormes que asumen demasiada responsabilidad. Joseph Pelrine expresa este principio de la siguiente manera:

> No haga nada que pueda recaer en alguien mas.

Muchos lenguajes orientados a proporcionan operaciones para objetos tanto estaticas como dinámicas; en Smalltalk hay únicamente envíos de mensajes dinámicos. En lugar de proporcionar operaciones de clase estáticas, por ejemplo, las clases son objetos y sencillamente les mandamos mensajes.

Casi todo en Smalltalk sucede enviando mensajes. En algún punto la acción debe llevarse a cabo:

- Las *declaraciones de variable* no son envios de mensajes. De hecho, las declaraciones de variable n i siquiera son ejecutables. Declarar una variable solo provoca que se asigne espacio para una referencia al objeto.
- Las*asignaciones* no son envios de mensajes. Una asignación a una variable provoca que ese nombre de variable sea vinculada en el contexto de esa definición.
- Los *Returns* no son envios de mensajes. Un return simplemente hace que el resultado computado sea devuelto al emisor.
- Los *primitivos* no son envios de mensajes. Están implementados en la máquina virtual.

Salvo estas pocas excepciones, casi todo lo demás realmente sucede mediante el envío de mensajes. En particular, dado que no hay "campos públicos"

en Smalltalk, la unica manera de actualizar una variable de instancia de otro objeto es enviandole un mensaje solicitando que actualize su propio campo. Por supuesto, proporcionar métodos getter y setter para todas las variables de instancia de un objeto no es una buena practica de la orientación a objetos. Joseph Pelrine también establece esta muy bien:

> No deje que nadie juegue con sus datos.

5.6 El Method lookup sigue la cadena de herencia

¿Que sucede exactamente cuando un objeto recibe un mensaje?

El proceso es bastante sencillo: La clase del receptor busca el método a usar para encargarse del mensaje. Si la clase no posee un método, le pregunta a su superclase, y asi sucesivamente, subiendo en la cadena de herencia. Cuando el método es encontrado, los argumentos se vinculan con los parámetros del método, y la maquina virtual lo ejecuta.

En esencia es tan simple como esto Sin embargo hay algunas cuestiones que necesitan ser respondidas con cuidado:

- *¿Qué sucede cuándo un método no devuelve explicitamente un valor?*
- *¿Qué sucede cuándo una clase reimplementa un metodo de superclase?*
- *¿Cuál es la diferencia entre enviar mensajes a* self *y* super*?*
- *¿Qué sucede cuando no se encuentra ningún método?*

Las reglas que aqui presentamos para el method lookup son conceptuales: las implementaciones de la maquina virtual usan todo tipo de trucos y optimizaciones para acelerar el method lookup. Ese es su trabajo, pero usted nunca debería detectar que estan haciendo algo diferente a nuestras reglas.

Primero echemos un vistazo a la estrategia básica de busqueda, y luego consideremos estas cuestiones.

El Method lookup

Suponga que creamos una instancia de EllipseMorph.

```
anEllipse := EllipseMorph new.
```

Si ahora le enviamos a este objeto el mensaje defaultColor, obtenemos el resultado Color yellow:

```
anEllipse defaultColor  ⟶  Color yellow
```

La clase EllipseMorph implementa defaultColor, asi que el método indicado es encontrado inmediatamente.

Method 5.14: *A locally implemented method*

```
EllipseMorph»defaultColor
   "responder el color predeterminado / estilo de relleno a el receptor"
   ↑ Color yellow
```

En cambio, si enviamos el mensaje openInWorld a anEllipse, el método no es encontrado de inmediato, dado que la clase EllipseMorph no implementa openInWorld. La busqueda por lo tamto continua en la superclase, BorderedMorph, y asu sucesivamente, hasta que un método openInWorld sea encontrado en la clase Morph (ver Figure 5.2).

Method 5.15: *An inherited method*

```
Morph»openInWorld
   "a\~nadir este morph al mundo."

   self openInWorld: self currentWorld
```

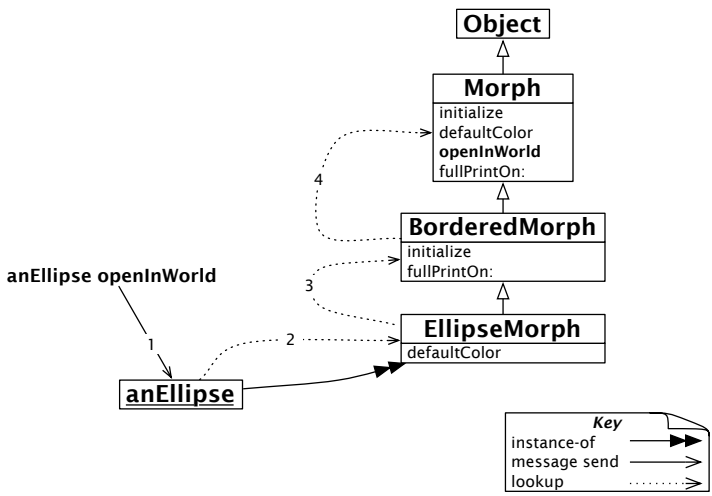


Figure 5.2: El Method lookup sigue la cadena de herencia.

Devolviendo self

Note que EllipseMorph»defaultColor (method 5.14) explicitamente devuelve Color yellow mientras que Morph»openInWorld (method 5.15) no parece devolver nada.

De hecho un método *siempre* responde un mensaje con un valor — que es, por supuesto, un objeto.

La respuesta puede estar definida por el concepto de ↑ en el método, pero si la ejecución alcanza el final del método sin ejecutar un ↑, el método aún devuelve un valor: devuelve el objeto que recibió el mensaje. Comúnmente decimos que el método "responde a self", porque en Smalltalk la pseudo-variable self representa al receptor del mensaje, como lo hace this en Java.

Esto sugiere que method 5.15 es equivalente a method 5.16:

Method 5.16: *Explicitly returning self*

```
Morph»openInWorld
   "A\~nade este morph al mundo."

   self openInWorld: self currentWorld
   ↑ self      "No hagas esto a menos que sea tu intenci\'on"
```

¿Por qué no es bueno escribir ↑ self explicitamente? Bueno, cuando devuelves algo explicitamente, estas comunicando que devuelves algo de interés para el emisor. Cuando expresamente devuelves self, estas diciendo que esperas que el emisor utilice el valor devuelto. Aquí este no es el caso, así que es mejor no retornar explicitamente self.

Se trata de un lenguaje común en Smalltalk, al cual Kent Beck denomina "Valor de retorno interesante"[3]:

> Devuelve un valor solo cuando es tu intención que el emisor use el valor.

Sobreescribiendo una extensión

Si miramos nuevamente la herencia de clase de EllipseMorph en Figure 5.2, observamos que las clases Morph y EllipseMorph implementan defaultColor. De hecho, si abrimos un nuevo morph (Morph new openInWorld) observamos que obtenemos un morph azul, mientras que una elipse será por defecto de color amarillo.

[3] Kent Beck, *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

Decimos que EllipseMorph *sobreescribe* el método defaultColor que hereda de Morph. El método heredado ya no existe desde el punto de vista de anEllipse.

A veces no queremos sobreescribir métodos heredados, sino más bien *extenderlos* con alguna funcionalidad nueva, es decir, nos gustaría ser capaces de invocar el método sobreescrito *en adición a* la nueva funcionalidad que estamos definiendo en la subclase. En Smalltalk, como en muchos lenguajes orientados a objetos que soportan herencia simple, esto se puede hacer enviando mensajes a super.

La aplicación mas importante de este mecanismo está en el método initialize. Siempre que una nueva instancia de una clase se inicializa, es fundamental también inicializar las variables de instancia heredadas. Sin embargo, el conocimiento de cómo hacer esto ya está capturado en los métodos initialize de cada una de las superclases de la cadena de herencia. ¡La subclase nada tiene que hacer tratando de inicializar las variables de instancia heredadas!

Por tanto es una buena práctica siempre que implementamos un método initialize enviar super initialize antes de realizar cualquier ulterior initialization:

Method 5.17: *Super initialize*

```
BorderedMorph»initialize
    "initialize the state of the receiver"
    super initialize.
    self borderInitialize
```

> Un método initialize siempre debería comenzar enviando super initialize.

Envío de mensajes a self y super

Necesitamos que los envíos de mensajes a super sends compongan el comportamiento heredado que de otra manera sería sobreescrito. Sin embargo la forma habitual de escribir métodos, ya sea heredados o no, es por medio de envíos a self .

¿Cómo es que los envíos de mensajes a self difieren de los envíos a super? Así como self, super representa al receptor del mensaje. Lo único que cambia es el method lookup. En lugar de que la busqueda del method lookup empiece en la clase del receptor, comienza en la superclase de la clase del método donde el envío a super toma lugar.

¡Note que super *no* es la superclase! Es un error común y natural el pensar esto. También es un error pensar que la búsqueda del method lookup

comienza en la superclase del receptor. Veremos con el siguiente ejemplo con precisión cómo funciona esto.

Considere el mensaje constructorString, que podemos enviarle a cualquier morph:

```
anEllipse constructorString  ⟶  '((EllipseMorph newBounds: (0@0 corner: 50@40)
    color: Color yellow) setBorderWidth: 1 borderColor: Color black)'
```

El valor de retorno es una cadena de caracteres que puede ser evaluada para recrear el morph.

¿Cómo es obtenido exactamente este resultado a través de una combinacion de envío de mensajes a self y a super? Primero, anEllipse constructorString provocara que el método constructorString sea encontrado en la clase Morph, como se muestra en Figure 5.3.

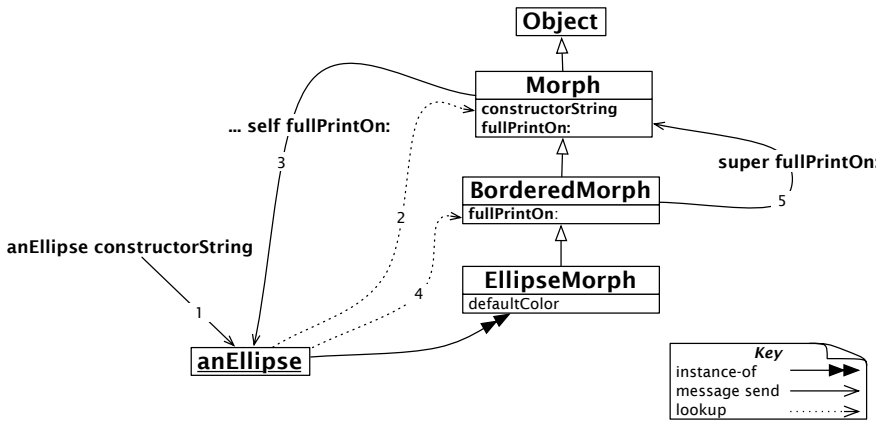


Figure 5.3: self and super sends

Method 5.18: *A* self *send*

```
Morph»constructorString
  ↑ String streamContents: [:s | self printConstructorOn: s indent: 0].
```

El método Morph»constructorString lleva a cabo un envío a self de printConstructorOn:indent:. Este mensaje también se busca subiendo en la cadena de herencia, comenzando en la clase EllipseMorph, y siendo encontrado en la clase Morph. Este método, a su vez hace un envío a self de printConstructorOn:indent:nodeDict:, que hace un envío a self de fullPrintOn:. Una vez más, fullPrintOn: se busca subiendo en la cadena de herencia comenzando por la clase EllipseMorph, y fullPrintOn: es encontrado en BorderedMorph (véa Figure 5.3 otra vez). Lo que es fundamental tener en cuenta es que el envío a self

provoca que el method lookup comience nuevamente en la clase del receptor, es decir, la clase de anEllipse.

> Un envío a self dispara un method lookup *dinámico* comenzando en la clase del receptor.

Method 5.19: *Combining* super *and* self *sends*

```
BorderedMorph»fullPrintOn: aStream
   aStream nextPutAll: '('.
   super fullPrintOn: aStream.
   aStream nextPutAll: ') setBorderWidth: '; print: borderWidth;
      nextPutAll: ' borderColor: ' , (self colorString: borderColor)
```

At this point, BorderedMorph»fullPrintOn: does a super send to extend the fullPrintOn: behaviour it inherits from its superclass. Because this is a super send, the lookup now starts in the superclass of the class where the super send occurs, namely in Morph. We then immediately find and evaluate Morph »fullPrintOn:.

Note that the super lookup did not start in the superclass of the receiver. This would have caused lookup to start from BorderedMorph, resulting in an infinite loop!

> A super send triggers a *static* method lookup starting in the superclass of the class of the method performing the super send.

Si piensas cuidadosamente acerca de los envíos a super y Figure 5.3, te darás cuenta que super bindings are static: lo único que importa es la clase en la cual se encuentra el texto del envío a super. By contrast, the meaning of self is dynamic: it always represents the receiver of the currently executing message. This means that *all* messages sent to self are looked-up by starting in the receiver's class.

Mensaje no comprendido

¿Qué sucede si el método que estamos buscando no se encuentra?

Supongamos que enviamos el mensaje foo a nuestra elipse. En primer lugar el method lookup normal subirá por la cadena de herencia todo el camino hasta Object (o más bien ProtoObject) en busca de este método. Cuando este método no se encuentra, la ind máquina virtual hará que el objeto envie self doesNotUnderstand: #foo. (See Figure 5.4.)

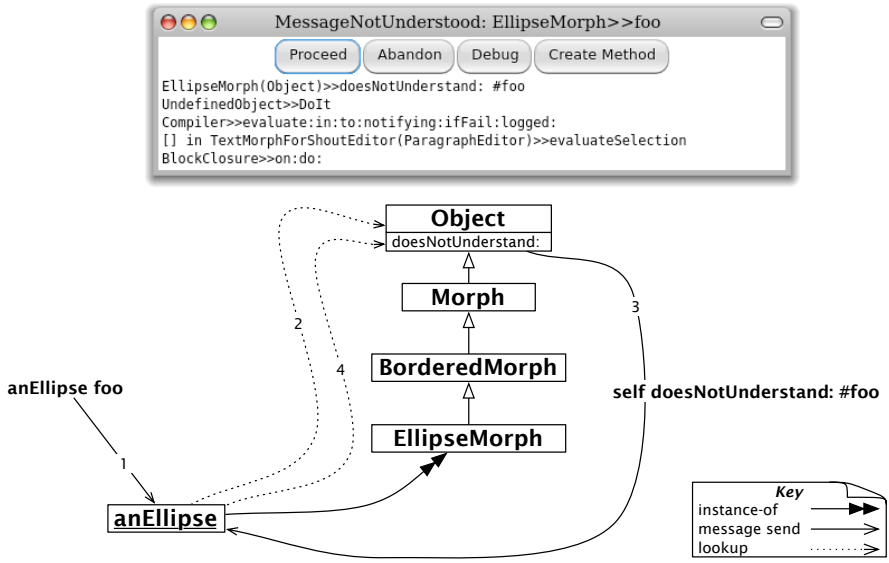


Figure 5.4: El mensaje foo no se comprende

Ahora, este es un envio de mensaje dinámico perfectamente ordinario, por lo que la búsqueda se inicia de nuevo desde la clase EllipseMorph , pero esta vez buscando el método doesNotUnderstand:. Como resultado, Object implementa doesNotUnderstand:. Este método creará un nuevo objeto MessageNotUnderstood que sea capaz de iniciar un depurador en el contexto de ejecución actual.

¿Por qué tomar este camino complicado de manejar como un error evidente? Bueno, esto ofrece a los desarrolladores una manera fácil de interceptar estos errores y tomar las medidas alternativas. Uno podría fácilmente reemplazar el método doesNotUnderstand: en cualquier subclase de Object y proporcionar una forma diferente de manejar el error.

De hecho, esta puede ser una manera fácil de implementar delegación automática de mensajes de un objeto a otro. Un objeto Delegador simplemente podría delegar todos los mensajes que no entiende a otro objeto cuya responsabilidad es manejarlos ¡o arrojar un error mismo!

5.7 Variables compartidas

Ahora echaremos un vistazo a un aspecto de Smalltalk que no queda cubierto tan facilmente por nuestras cinco reglas: las variables compartidas.

Smalltalk proporciona tres tipos de variables compartidas: (1) variables *globalmente* compartidas; (2) variables compartidas entre instancias y clases (*variables de clase*), y (3) variables compartidas entre un grupo de clases (*pool variables*). Los nombres de todas esas variables compartidas comienzan con una letra mayuscula, para advertirnos que realmente estan compratidas entre múltiples objetos.

Variables globales

En Pharo, todas las variables globales estan almacenadas en un namespace llamado Smalltalk, el cual está implementado como una instancia de la clase SystemDictionary. Las variables globales son accesibles en todas partes. Cada clase es nombrada por una variable global; además, unas pocas variables globales son usadas para nombrar objetos especiales u objetos comunmente útiles.

La variable Transcript nombra una instancia de TranscriptStream, un flujo de datos que escribe en una ventana que se desplaza. El siguiente código muestra cierta información y luego avanza a la siguiente linea en el Transcript .

```
Transcript show: 'Pharo is fun and powerful' ; cr
```

Antes de ejecutar do it, abre un transcript seleccionando World ▷ Tools ... ▷ Transcript.

HINT *Escribir en el Transcript es lento, en especial cuando la ventana de transcript está abierta. Asi que, si experimentas algo de lentitud y estas escribiendo en el Transcript, considera cerrarlo.*

Otras variables globales útiles

- Smalltalk es la instancia de SystemDictionary que define todas las variables globales — incluyendo al mismo Smalltalk. Las claves para este diccionario son los símbolos que dan nombre a los objetos globales en el codigo Smalltalk. Asi que, por ejemplo,

  ```
  Smalltalk at: #Boolean    ⟶    Boolean
  ```

  Dado que Smalltalk en si mismo es una variable global,

  ```
  Smalltalk at: #Smalltalk    ⟶    a SystemDictionary(lots of globals)}
  ```

  y

  ```
  (Smalltalk at: #Smalltalk) == Smalltalk    ⟶    true
  ```

- Sensor es una instancia de EventSensor, y representa la entrada de información a Pharo. Por ejemplo, el Sensor keyboard responde con el siguiente input de un caracter en el teclado, y Sensor leftShiftDown responde true si la tecla shift izquierda está siendo oprimida, mientras que Sensor mousePoint responde un Point indicando la locación actual del ratón.

- World es una instancia de PasteUpMorph que representa la pantalla. World bounds responde un rectangulo que define todo el espacio de la pantalla; todos los Morphs en la pantalla son submorphs de World.

- ActiveHand es la instancia actual de HandMorph, la representacion gráfica del cursor. Los submorphs de ActiveHand contienen cualquier cosa que esté siendo arrastrada por el ratón.

- Undeclared es otro diccionario — contiene todas las variables no declaradas. Si escribes un método que hace referencia a una variable no declarada, el navegador normalmente te sugerirá declararla, por ejemplo, como una variable global o como una variable de instancia de la clase. Sin embargo, si mas tarde borras la declaración, el código referenciará a una variable no declarada.

  ¡Examinar a Undeclared puede a veces ayudar a explicar este comportamiento extraño!

- SystemOrganization es una instancia de SystemOrganizer: registra la organización de las clases en paquetes. Mas precisamente, clasifica los *nombres* de clases, por lo que

```
SystemOrganization categoryOfElement: #Magnitude  ⟶  #'Kernel−Numbers'
```

El procedimiento actual consiste en limitar estrictamente el uso de las variables globales; en general es mejor utilizar las variables de instancia de la clase o las variables de clase, y proveer métodos de clase para acceder a ellas. En realidad, si Pharo hoy fuera a ser implementado desde cero, la mayoria de las variables globales que no son clases serían reemplazadas por singletons.

La manera habitual de definir una variable global es mediante el do it en una asignacion a un identificador en mayusculas pero no declarado. The parser will then offer to declare the global for you. If you want to define a global programmatically, just execute Smalltalk at: #AGlobalName put: nil. To remove it, execute Smalltalk removeKey: #AGlobalName.

Variables de clase

En ocasiones nos es necesario ,compartir información entre todas las instancias de una clase y la clase en si. Esto es posible utilizando *variables de clase*.

El termino variable de clase indica que el tiempo de vida de la variable es el mismo que el de la clase. Sin embargo, lo que este termino no expresa es que estas variables son compartidas entre todas las instancias de una clase, así como también por la clase en sí, como se muestra en Figure 5.5. En efecto, un mejor nombre hubiera sido *variables compartidas*, ya que este expresa más claramente su rol, y además advierte del peligro de utilizarlas, en especial si son modificadas.

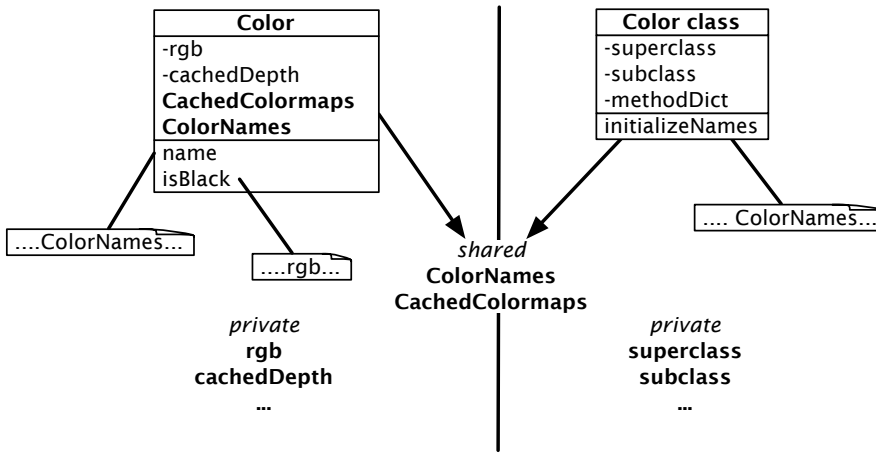


Figure 5.5: M'etodos de instancia y de clase accediendo diferentes variables.

En Figure 5.5 podemos ver que rgb y cachedDepth son variables de instancia de Color, y por tanto, solo accesibles a instancias de Color. También vemos que superclass, subclass, methodDict, etc. son variables de instancia de clase, *i.e.*, variables de instancia solo accesibles por la clase Color.

Pero también podemos ver algo nuevo: ColorNames y CachedColormaps son *variables de clase* definidas por Color. La capitalización de estas variables nos da una pista acerca de que son compartidas. De hecho, no solo pueden todas las instancias de Color acceder estas variables compartidas, sino también la clase Color en sí, *y cualquiera de sus subclases*. Tanto los métods de instancia como de clase pueden acceder estas variables compartidas.

Una variable de clase es definida en la plantilla de definición de la clase. Por ejemplo, la clase Color define un gran numero de variables de clase para acelerar la creación de colores; su definición se muestra debajo. A class variable is declared in the class definition template. For example, the class Color defines a large number of class variables to speed up color creation; its definition is shown below (class 5.20).

Class 5.20: *Color and its class variables*

```
Object subclass: #Color
    instanceVariableNames: 'rgb cachedDepth cachedBitPattern'
    classVariableNames: 'Black Blue BlueShift Brown CachedColormaps ColorChart
     ColorNames ComponentMask ComponentMax Cyan DarkGray Gray
     GrayToIndexMap Green GreenShift HalfComponentMask HighLightBitmaps
     IndexedColors LightBlue LightBrown LightCyan LightGray LightGreen LightMagenta
      LightOrange LightRed LightYellow Magenta MaskingMap Orange PaleBlue
     PaleBuff PaleGreen PaleMagenta PaleOrange PalePeach PaleRed PaleTan
     PaleYellow PureBlue PureCyan PureGreen PureMagenta PureRed PureYellow
     RandomStream Red RedShift TranslucentPatterns Transparent VeryDarkGray
     VeryLightGray VeryPaleRed VeryVeryDarkGray VeryVeryLightGray White Yellow'
    poolDictionaries: ''
    category: 'Graphics−Primitives'
```

La variable de clase ColorNames es un array que contiene los nombres de colores usados frecuentemente. Este array es compartido por todas las instancias de Color y su subclase TranslucentColor. Es accesible desde todos los métodos de instancia y de clase.

ColorNames es inicializada una vez en Color class»initializeNames, pero es accedido por instancias de Color. El método Color»name utiliza la variable para averiguar el nombre de un color. Ya que la mayoria de los colores no tienen nombres, se pensó inapropiado añadir una variable de instancia name a cada color.

Inicialización de clases

La presencia de variables de clase hace surgir la pregunta: ¿Cómo las inicializamos? Una solución es la inicialización perezosa. Esta se puede lograr introduciendo un método accessor el cual, cuando es ejecutado inicializa la variable si esta no ha sido inicializada previamente. Esto implica que debemos utilizar el accessor todo el tiempo y nunca utilizar la variable directamente. Aún más, impone el costo del envío del accessor y la prueba de inicialización. También podría decirse que desestima la idea de usar una variable de clase, ya que de hecho esta ya no es mas compartida.

Method 5.21: *Color class»>colorNames*

```
Color class»colorNames
  ColorNames ifNil: [self initializeNames].
  ↑ ColorNames
```

Otra solución es sobrescribir el método de clase initialize.

Method 5.22: *Color class»initialize*

```
Color class»initialize
```

```
...
self initializeNames
```

Si adoptas esta solución, necesitas recordar invocar el método initialize luego de definirlo, *e.g.*, evaluando Color initialize. Aunque los metodos initialize del lado de clase son ejecutados automáticamente cuando el codigo es cargado en la memoria, *no* son ejecutados automáticamente cuando son escritos la primera vez en el navegador y compilados, o cuando son editados y recompilados.

Variables de agrupamiento

Las variables de agrupamiento son variables que se comparten entre varias clases que pueden no estar relacionadas por herencia. Las variables de agrupamiento fueron originalmente almacenadas en diccionarios de agrupamiento; ahora deberían ser definidas como variables de clase de clases dedicadas (subclases de SharedPool). Nuestro consejo es evitarlas; las necesitarás solo en circunstancias raras y especificas. Nuestro objetivo aquí es por lo tanto explicar variables de agrupamiento solo lo suficiente para que puedas comprenderlas cuando te encuentres leyendo código.

Una clase que accede a una variable de agrupamiento debe mencionarla en su definición de clase. Por ejemplo, la clase Text indica que esta usando el diccionario de agrupamiento TextConstants, el cual contiene todas las constantes de texto, tales como CR y LF. Este diccionario tiene una clave #CR la cual esta ligada al valor Character cr, *i.e.*, el caracter de retorno de carro.

Class 5.23: *Pool dictionaries in the* Text *class*

```
ArrayedCollection subclass: #Text
    instanceVariableNames: 'string runs'
    classVariableNames: ''
    poolDictionaries: 'TextConstants'
    category: 'Collections−Text'
```

Esto permite a los métodos de la clase Text acceder a las claves del diccionario en el cuerpo del método *directamente*, *i.e.*, utilizando sintaxis de variable en lugar de una búsqueda explicita en un diccionario. Por ejemplo, podemos escribir el siguiente método.

Method 5.24: *Text»testCR*

```
Text»testCR
    ↑ CR == Character cr
```

Una vez más, recomendamos que evites el uso de variables y diccionarios de agrupamiento.

5.8 Resumen del capítulo

El modelo de objetos de Pharo es tanto simple como uniforme. Todo es un objeto, y practicamente todo sucede mediante el envio de mensajes.

- Todo es un objeto. Entidades primitivas como los enteros son objetos, pero también las clases son objetos de primer orden.
- Todo objeto es instancia de una clase. Las clases definen la estructura de sus instancias a través de variables de instancia *privadas* y su comportamiento mediante métodos *púbicos*. Cada clase es la única instancia de su metaclase. Las variables de clase son variables privadas compartidas entre la clase y todas sus instancias. Las clases no pueden acceder directamente a las variables de instancia de sus instancias, y las instancias no pueden acceder a las variables de instancia de su clase. Se deben definir accessors si esto fuera necesario.
- Toda clase tiene una superclase. La raíz de la jerarquía de herencia es ProtoObject. Las clases que definas, sin embargo, deberían normalmente heredar de Object o sus subclases. No hay una sintaxis para definir clases abstractas. Una clase abstracta es simplemente una clase con un método abstracto — uno cuya implementacióon consiste en la expresión self subclassResponsibility. Aunque Pharo soporta solo herencia simple, es fácil compartir implementaciones de métodos agrupándolos como *traits*.
- Todo sucede mediante el envío de mensajes No "llamamos métodos", "enviamos mensajes". El receptor luego elije su propio método para responder al mensaje.
- El method lookup sigue la cadena de herencia; Los envíos a self son dinámicos y comienzan el method lookup nuevamente desde la clase del receptor, mientras que los envíos a super son estáticos, y comienzan en la superclase de la clase en la cual el envío a super fue escrito.
- Hay tres tipos de variables compartidas. Las variables globales son accesibles desde cualquier lugar del sistema. Las variables de clase son compartidas por una clase y sus subclases e instancias. Las variables de agrupamiento son compartidas por un grupo seleccionado de clases. Deberías evitar las variables compartidas tanto como te sea posible.

Chapter 6

El entorno de progamación Pharo

El objetivo de este capítulo es mostrar la forma de desarrollar programas en el entorno de programación Pharo. Ya se ha visto como definir métodos y clases usando el buscador; este capítulo mostrará más características del buscador, e introducirá algunos de los demás buscadores.

Por supuesto, ocasionalmente puede suceder que el programa no realice la función que de él se espera. Por ello, Pharo tiene un excelente debugger, pero al igual que otras herramientas poderosas, puede resultar confuso cuando se lo usa por primera vez. En este capítulo se dará un ejemplo de depuración y se demostrarán algunas de las características de esta herramienta.

Una de las características únicas de Smalltalk es que mientras se está programando, uno pertenece a un mundo de objetos vivos, muy distinto al de los programas estáticos. Esto hace posible obtener feedback de forma instantánea mientras se programa, lo cual permite al programador ser más productivo. Hay dos herramientas que permiten analizar, e incluso cambiar, los objetos vivos: el *inspector* y el *explorer*.

La consecuencia de programar en un mundo de objetos vivos, en vez de hacerlo con archivos y un editor de texto, es que se debe realizar una acción explícita para exportar el programa desde la imagen de Smalltalk. La forma antigua de hacerlo posible en todos los dialectos Smalltalk, es mediante creación de un *fileout* o un *change set*, los cuales son en esencia archivos de texto encriptados que pueden ser importados desde otro sistema. En cambio, la nueva forma de hacer esto en Pharo is subir el código a un repositorio de versiones en un servidor. Esto se logra usando una herramienta llamada Monticello, y es una manera más poderosa y eficiente de trabajar, especialmente en grupo.

6.1 Resumen

Smalltalk y las interfaces gráficas modernas fueron desarrolladas juntas. Aun antes del primer lanzamiento público de Smalltalk en 1983, este tenía un entorno gráfico de desarrollo auto-alojado, y todo el desarrollo en Smalltalk tomaba lugar en él. Para empezar, se verán las herramientas principales de Pharo.

- El Browser es la principal herramienta de desarrollo. Se usa para crear, definir y organizar las clases y métodos. Usándolo también se puede navegar por todas las librerías de clase: a diferencia de otros entornos donde el código fuente es almacenado en archivos diferentes, en Smalltalk todas las clases y métodos están contenidos en la imagen.

- El Message Names se usa para ver todos los métodos con un selector particular, o con uno que contenga una substring de caracteres.

- El Method Finder también le ayudará a encontrár métodos, pero de acuerdo a lo que *hacen* y además por su nombre.

- El Monticello Browser es el punto de partida para cargar código desde, o guardar código en, los paquetes Monticello.

- El Process Browser permite ver todos los procesos(o *threads*) que se están ejecutando en Smalltalk.

- El Test Runner le permite correr y depurar las pruebas SUnit, y se describe en Chapter 7.

- El Transcript es una ventana en el canal de salida del Transcript, el cual es útil para escribir mensajes de registro y que ya se ha descrito en la Section 1.5.

- El Workspace es una ventana donde se pueden tipear entradas. Puede ser usado con cualquier propósito, pero es más usado para tipear expresiones de Smalltalk y ejecutarlas con do it. El uso del workspace también fue ilustrado en la Section 1.5.

El Debugger tiene una tarea obvia, pero descubrirá que tiene un rol más importante comparado con los debuggers de otros lenguajes de programación , porque en Smalltalk se puede *programar* en el debugger. El debugger no se lanza desde un menú; normalmente aparece al correr una prueba que falla, al tipear CMD−. para interrumpir un proceso en curso, o insertando una expresión self halt en el código.

6.2 El Navegador

Muchos navegadores de clases diferentes se han desarrollado a trevés de los años para Smalltalk. Pharo simplifica esta historia por medio de ofrecer un navegador que integra diversas vistas. Figure 6.1 muestra el navegador como aparece la primera vez que es abierto.[1]

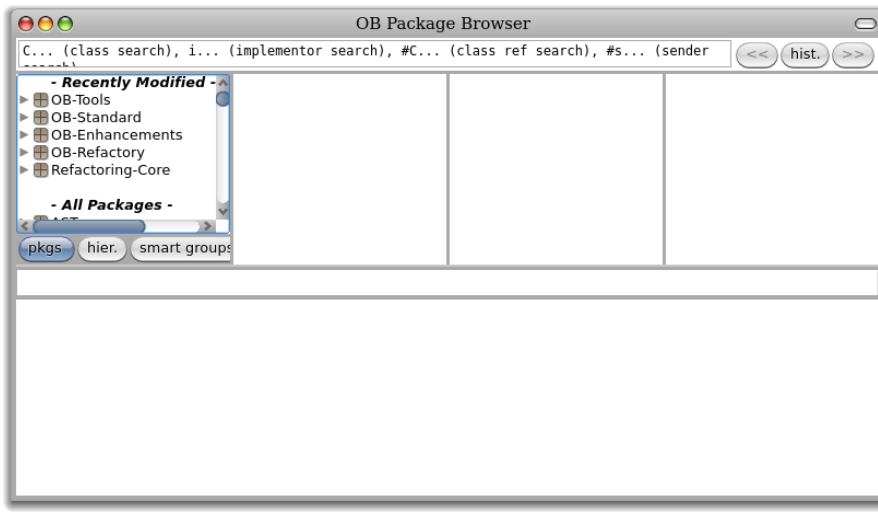


Figure 6.1: El Navegador

Los cuatro paneles pequeños en la parte superior del navegador representan una visión jerárquica de los métodos que se hallan en el sistema, de una manera muy similar a como el *File Viewer* de NeXTstep y el *Finder* de Mac OS X en modo de columna, dan una vista de los archivos en el disco. En el panel que se halla en el extremo izquierdo se encuentran las listas de los *paquetes* de clases; seleccione uno (por ejemplo *Kernel*) y el panel a la derecha mostrará entonces todas las clases en ese paquete.

De forma similar, si selecciona una de las clases en el segundo panel, por ejemplo, Model (véase Figure 6.2), el tercer panel mostrará todos los *protocolos* definidos para esa clase, así como un protocolo virtual *--all--*, que está seleccionado por defecto. Los protocolos son una forma de categorizar los métodos; hacen más facil hallar y pensar en en el comportamiento de una clase al separarla en piezas más pequeñas y conceptualmente coherentes. El cuarto panel muestra los nombres de todos los métodos definidos en el protocolo seleccionado. Si selecciona el nombre de un método, el código fuente

[1] Recuerde que si el navegador que usted tenga no se ve como el mostrado en Figure 1.12, quizás necesite cambiar el navegador por defecto. Véase FAQ 5, p. 332.

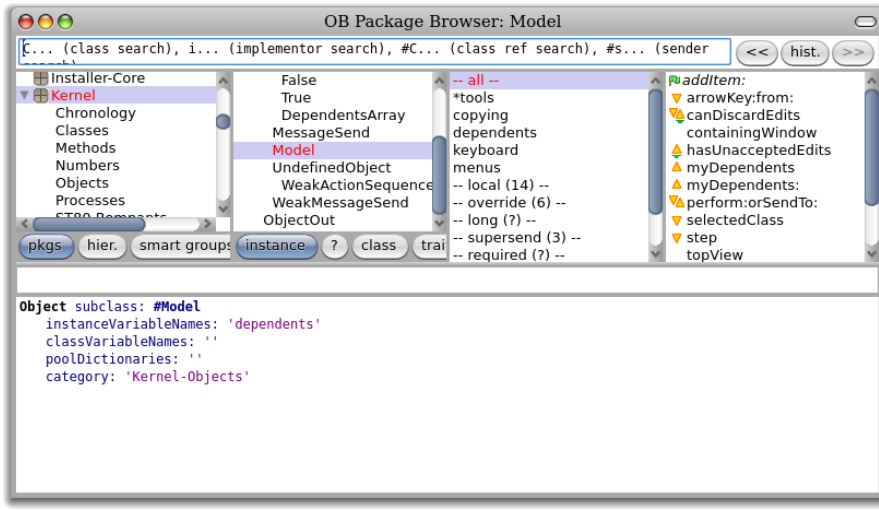


Figure 6.2: El Navegador con la clase Model seleccionada

del método correspondiente aparece en el panel alargado en la parte inferior del navegador, donde puede verlo, editarlo, y guardar la versión editada. Si selecciona la clase `Model`, el protocolo *dependents* y el método `myDependents`, el navegador debería verse como en Figure 6.3.

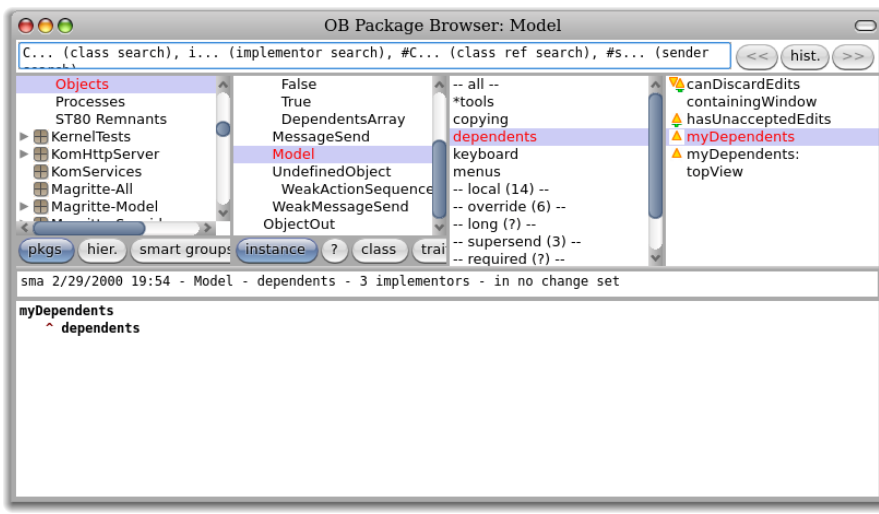


Figure 6.3: El Navegador mostrando el método myDependents en la clase Model

A diferencia de los directorios en el *Finder* de Mac OS X, los cuatro paneles superiores no son exactamente iguales. Mientras que las clases y los métodos son parte del lenguaje Smalltalk, los paquetes y protocolos no lo son: son una comodidad agregada por el navegador para limitar la cantidad de información que debe ser mostrada en cada panel. Por ejemplo, si no hubiera protocolos, el navegador tendría que mostrar una lista de todos los métodos en la clase seleccionada; para muchas clases esta lista sería muy extensa para ser recorrida convenientemente.

Por esta razón, la manera de crear un nuevo paquete o protocolo es diferente de la forma en que se crea una nueva clase o método. Para crear un nuevo paquete, haga click en el panel de paquetes y seleccione `new package`; para crear un nuevo protocolo, haga click en el panel de protocolos y seleccione `new protocol`. Ingrese el nombre de este en el cuadro de diálogo, y está listo: no hay nada más en un paquete o protocolo que su nombre y su contenido.

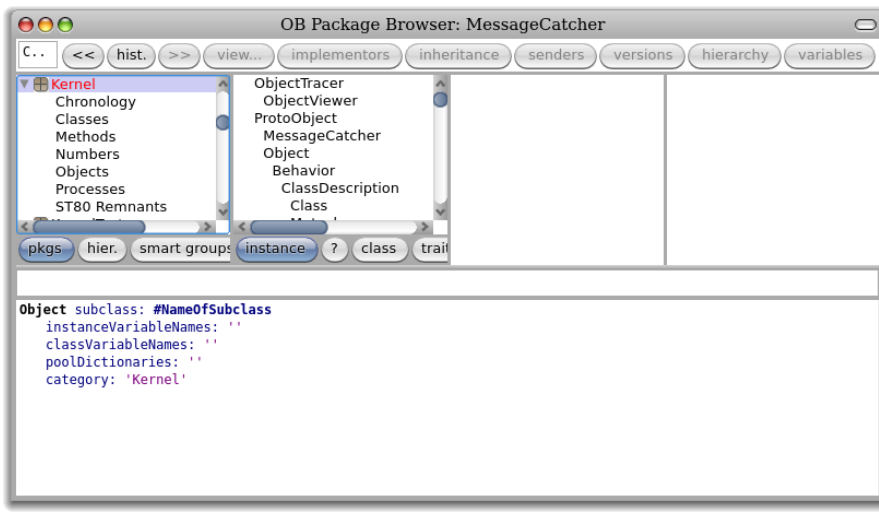


Figure 6.4: El Navegador mostrando la plantilla de creación de clases

Por otro lado, para crear una nueva clase o un nuevo método, se deberá escribir cierto código Smalltalk. Si hace click en el paquete seleccionado(en el panel extremo izquierdo), el panel inferior del navegador mostrará la plantilla para la creación de la clase. (Figure 6.4). Se crea una nueva clase al editar esta plantilla: reemplace `Object` por el nombre de la clase existente de la cual quiere crear una nueva subclase, reemplace `NameOfSubclass` por el nombre que desea darle a la nueva subclase, y complete los nombres de las variables de instancia si los conoce. La categoría de la nueva clase es por defecto la del

paquete que esté seleccionado[2], pero puede cambiar esto también si así lo desea. Si ya tiene el navegador centrado en la clase de la que desea una subclase, puede obtener la misma plantilla con unas pequeñas diferencias en la inicialización mediante action-clicking en el panel de clase, y seleccionando class templates . . . ▷ subclass template. También puede solo editar la definición de una clase existente, cambiando el nombre de la clase a uno nuevo. En todas las clases, cuando acepta la nueva definición, la nueva clase (aquella cuyo nombre está a continuacion de #) es creada (así como su correspondiente metaclase). Crear una clase también crea una variable global que referencia a la clase, que es por lo que uno puede referirse a todas las clases existentes por medio del uso sus nombres.

Puede ver por qué el nombre de la nueva clase tiene que aparecer como un Symbol (*i.e.*, antecedido por #) en la plantilla de creación de la clase, pero después de que la clase ha sido creada, el código puede referirse a la clase mediante usar el nombre como identificador (*i.e.*, sin el #)?

El proceso de creación de un nuevo método es similar. Primero selecciona la clase en la que se desea que exista el método, y luego se elige un protocolo. El navegador mostrará una plantilla de creación de métodos, como se ve en Figure 6.5, la cual puede completar o editar.

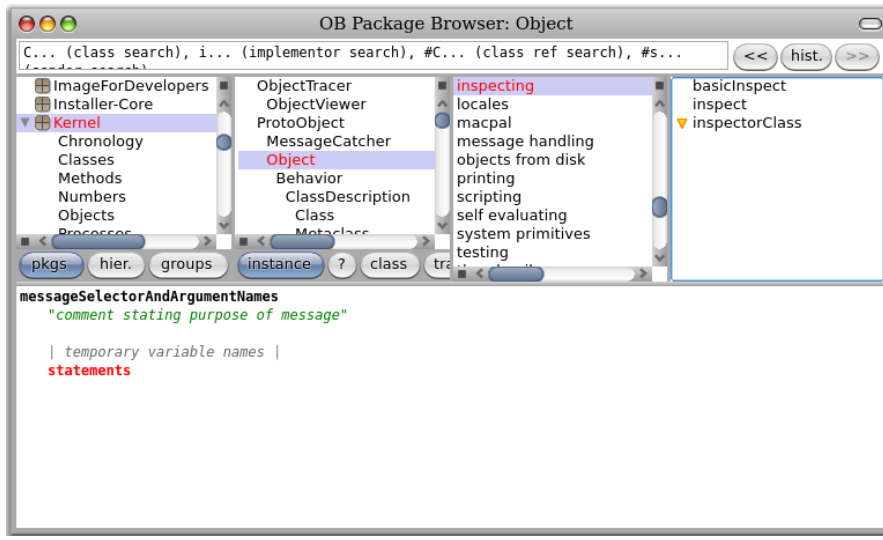


Figure 6.5: El navegador mostrando la plantilla de creación de métodos

[2]Recuerde que los paquetes y categorías no son exactamente lo mismo. Se verá la relación precisa en Section 6.3

Navegando los espacios de código

El navegador provee varias herramientas para explorar y analizar código. Estas herramientas pueden ser accedidas action-clicking en varios de los menúes contextuales, o en el caso de las herramientas más frecuentemente usadas, por medio de un atajo de teclado.

Abriendo un nueva ventana del navegador

El ocasiones, querrás abrir múltiples ventanas de explorador. Cuando estás escribiendo código seguramente necesitarás al menos dos: una para el método que estás escribiendo, y otra para navegar por el sistema para ver como funcionan las cosas. Puedes abrir un navegador en una clase seleccionando cualquier texto usando el acceso directo de teclado CMD−b.

Prueba esto: en un workspace, tipea el nombre de una clase (por ejemplo Morph *), seleccionalo, y presiona* CMD−*b. Este truco es frecuentemente útil; funciona en cualquier ventana de texto.*

Senders e Implementors de un mensaje

Action-clicking browse . . . ▷ senders (n) en el panel de métodos traerá una lista de todos los métodos que puede usar el seleccionado método. Con el navegador abierto en Morph, haga clic en el método drawOn: en el panel de métodos; el cuerpo de drawOn: se mostrará en la parte inferior del navegador. Si ahora seleccionas senders (n) (Figure 6.6), aparecerá un menú con drawOn: como el ítem mas alto, y abajo de este, todos los mensajes que drawOn: envía (Figure 6.7). Seleccionando un ítem en este menú abrirá el navegador con una lista de todos los métodos en la imagen que envía el seleccionado mensaje (Figure 6.8).

La "n" en senders (n) te indica que el acceso rápido del teclado para encontrar los senders de un mensaje es CMD−n. Esto trabajará en *cualquier* ventana de texto.

Seleccionar el texto "drawOn:" en el panel de código y presionar CMD−*n para inmediatamente traer los senders de* drawOn:.

Si estás buscando los senders de drawOn: en AtomMorph»drawOn:, verás que esto es un super send. Por lo tanto sabemos que el método que se ejecutará será en la superclase de AtomMorph . Que clase es esta? Action-click browse ▷ hierarchy implementors y verás que esta es EllipseMorph.

Ahora mire el sexto sender en la lista, Canvas»draw , mostrado en la Figure 6.8. Puedes ver que este método envía drawOn:para cualquier objeto que

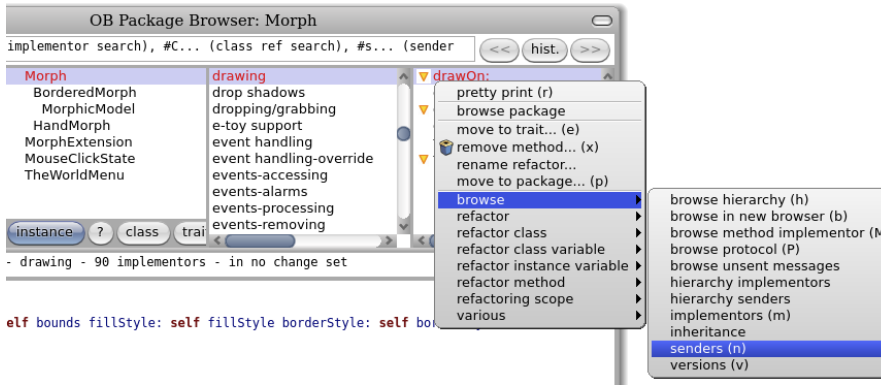


Figure 6.6: Los ítems del menú senders (n) .

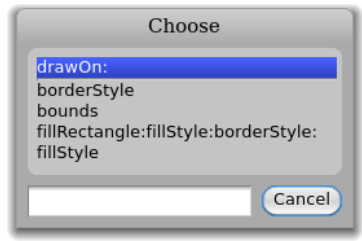


Figure 6.7: Elija los senders del mensaje .

se le pasa como un argumento, lo que potencialmente podría ser una instancia de cualquier clase. Un análisis de flujo puede ayudar a entender la clase del receptor de algunos mensajes, pero en general, no existe una manera sencilla para que el navegador pueda saber cual envío de mensaje puede causar que los métodos serán ejecutados. Por esta razón, los "senders"en el navegador muestran exactamente lo que sugiere su nombre: todos los emisores de los mensajes con el selector elegido. El navegador de senders es sin embargo extremadamente útil cuando necesitas comprender cómo puedes *usar* un método: te permite navegar rápidamente a través del ejemplo utilizado. Dado que todos los métodos con el mismo selector deben usarse de la misma manera, todos los usos de un determinado mensaje tienen que ser similares.

El navegador de los implementors trabaja de una manera similar, pero en vez del listado de emisores de un mensaje, esto lista todas las clases que implementan un método con el mismo selector. Para ver esto, seleccione drawOn: en el panel de métodos y seleccione browse ▷ implementors (m) (o seleccione el texto "drawOn:" en el panel de código y presione CMD−m). Deberías obtener una ventana con una lista de métodos mostrando una lista desplaz-

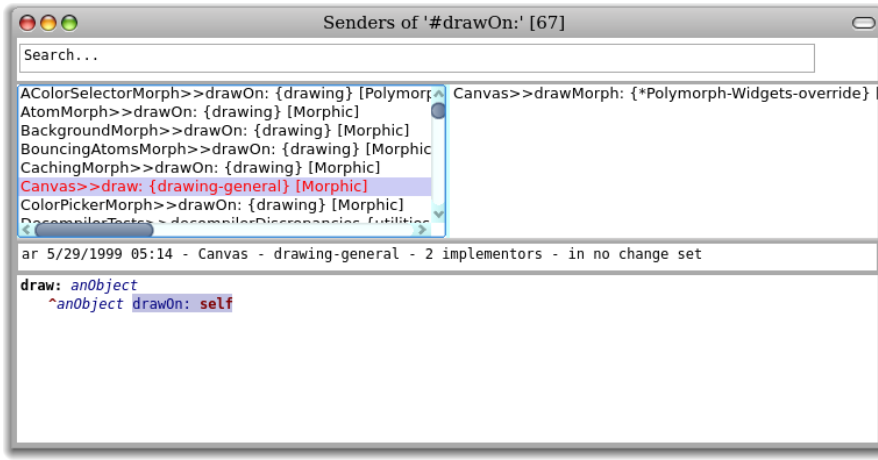


Figure 6.8: El navegador de Senders muestra que el método Canvas»draw envía el mensaje drawOn: a su argumento.

able de 90-odd clases que implementa un método drawOn:. No debería ser tan sorprendente que tantas clases implementen este método: drawOn: es el mensaje que se entiende por cada objeto que es capaz de representarse en la patalla.

Versiones de un método

Cuando guardas una nueva versión de un método, la antigua no se pierde. Pharo mantiene todas las versiones antiguas, y te permite comparar las diferentes versiones y volver ("revert") a una versión antigua. El ítem del menú browse ▷ versions (v) da acceso a las sucesivas modificaciones introducidas en el método seleccionado. En Figure 6.9 podemos ver dos versiones del método buildWorldMenu:.

El panel superior muestra una línea para cada versión del método, en el que figuran las iniciales del programador que las escribió, la fecha y hora en que se guardó, los nombres de la clase y el método, y el protocolo en el que se ha definido. La actual versión (activa) está en la parte superior de la lista; independiente de la versión seleccionada se muestra en la parte inferior del panel. También se proporcionan botones para mostrar las diferencias entre el método seleccionado y la versión actual, y para volver a la versión seleccionada.

La existencia de las versiones del navegador significa que nunca tendrás que preocuparte de preservar el código que piensas que tal ves ya no sea necesario: simplemente suprímelo. Si encuentras que lo *necesitas*, siempre

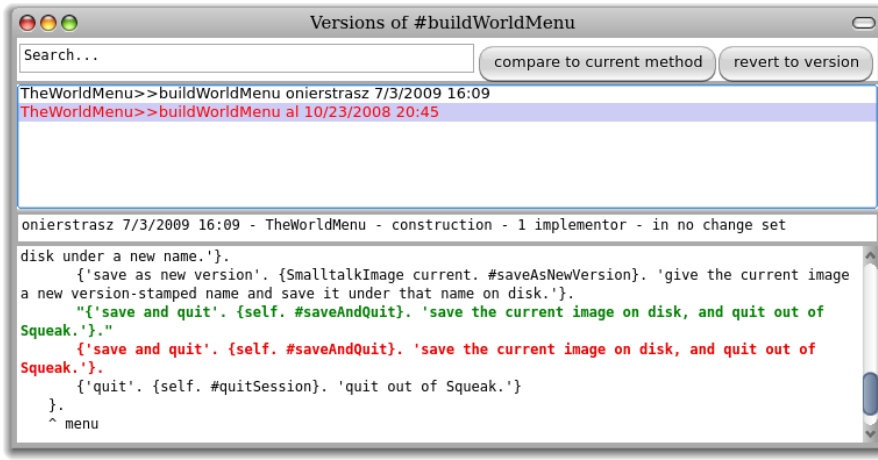


Figure 6.9: El navegador de versiones mostrando dos versiones del método TheWorldMenu»buildWorldMenu:

puedes volver a la versión anterior, o copiar el fragmento de la versión anterior y pegarlo en otro método. Se acostumbra a utilizar versiones; "comentar" el código que ya no es necesario es una mala práctica porque hace que el actual código sea más difícil de leer. La tasa de legilibilidad de código de Smalltalkers es extremadamente alta.

HINT *Que pasa si se elimina completamente un método?, y luego decides que deseas recuperarlo? Puedes encontrar lo borrado en un change set, donde puedes pedir ver las versiones por action-clicking. El change set está descrito en Section 6.8*

Sobre escribiendo métodos

El navegador de herencia mostrará todos los métodos sobre escritos por el método mostrado. Para ver como trabaja, seleccione el método ImageMorph »drawOn: en el navegador. Nota el icono triangular próximo al nombre del método (Figure 6.10). El triángulo que apunta hacia arriba te dice que ImageMorph»drawOn: sobre escribe un método heredado (*i.e.*, Morph»drawOn:), y el triángulo apuntando hacia abajo te dice que es sobre escrito por subclases. (También puedes hacer click sobre los iconos para ir a estos métodos.) Ahora selecciona browse ▷ inheritance. El navegador de herencia te mostrará la jerarquía de los métodos sobre escritos (ver Figure 6.10).

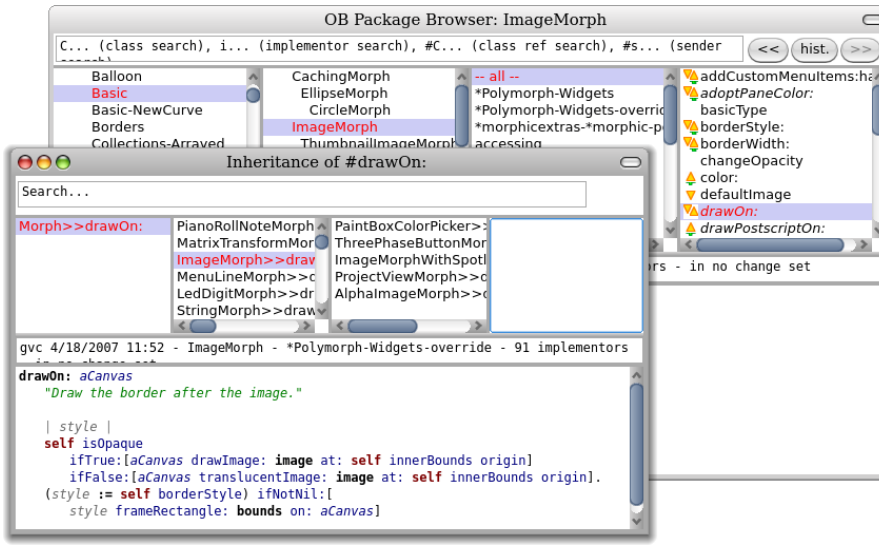


Figure 6.10: ImageMorph»drawOn: y los métodos que este sobre escribe. Los hermanos de los métodos seleccionados son mostrados en las listas de desplazables.

La vista Jerárquica

De forma predeterminada, el navegador presenta una lista de los paquetes en el panel que está más a la izquierda. Sin embargo, es posible cambiar a una vista jerárquica de clases. Simplemente seleccione una clase de particular interés, tal como ImageMorph y, haga clic en el botón hier.. A continuación, podrá ver en el panel que está más a la izquierda una jerarquía de clases mostrando todas las superclases y subclases de la clase seleccionada. El segundo panel muestra los paquetes que implementan los métodos de la clase seleccionada. El navegador jerárquico Figure 6.11, revela que la superclase directa de ImageMorph es Morph.

Encontrar las referencias a variables

Por action-clicking en una clase en el panel de clases, y seleccionando browse ▷ chase variables, puedes encontrar donde se utiliza una variable de clase o una variable de instancia. Se le presentará con un *chasing browser* que le permitirá recorrer los descriptores de acceso de todas las variables de instancia y las variables de clases , y, a su vez, los métodos que envian estos descriptores, etc. (Figure 6.12).

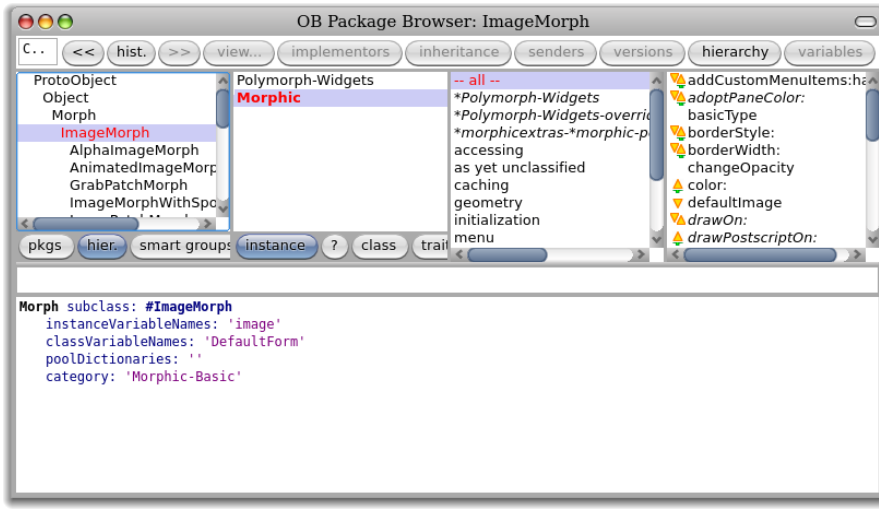


Figure 6.11: Una vista jerárquica de ImageMorph.

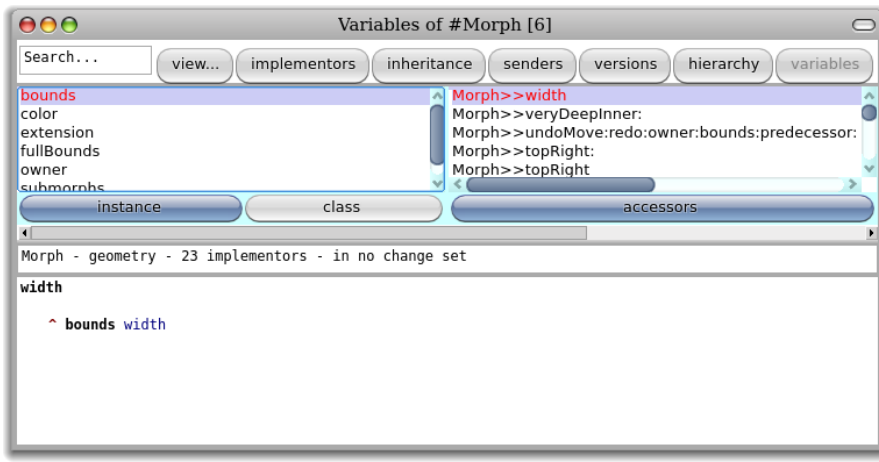


Figure 6.12: Un navegador de seguimiento para Morph.

Fuente

El item del menú various ▷ view ... disponible por action-clicking en el panel del método trae el menú "how to show", el cual te permite escoger como el navegador mostrará el método seleccionado en el panel de fuente. Las opciones incluyen el código fuente, el código fuente prettyPrinted , byteCode y

el código fuente decompiled desde los códigos byte.

Tenga en cuenta que seleccionando prettyPrint en el menú "how to show" *no* es lo mismo como el resaltado de sintaxis de un método antes de guardarlo [3]. Los controles del menú sólo cómo muestra el navegador, y no tiene efecto sobre el código almacenado en el sistema. Esto lo puedes comprobar abriendo dos navegadores, y seleccionando en uno prettyPrint y en el otro source. De hecho, focalizando dos navegadores en el mismo método y seleccionando en uno byteCode y en el otro decompile es una buena forma de aprender acerca del conjunto de instrucciones byte-coded de la máquina virtual Pharo.

Refactorización

Los menúes contextuales ofrecen un gran número de refactorizaciones estándares. Simplemente action-click en cualquiera de los cuatro paneles para ver las operaciones de refactorización actualmente disponibles. Ver Figure 6.13.

La Refactorización estuvo previamente disponible solamente en un navegador especial llamado navegador de refactorización, pero ahora se puede acceder a este desde cualquier navegador.

Los menúes de navegación

Muchas funciones adicionales están disponibles por action-clicking en los paneles del navegador. Incluso si las etiquetas sobre los ítems del menú son los mismos, su *significado* puede ser dependiente del contexto. Por ejemplo, el panel de paquetes, el panel de clases, el panel de protocolos y el panel de métodos todos tienen un ítem del menú file out. Sin embargo, ellas hacen cosas diferentes: del menú del panel de paquetes file out emerge el paquete completo, del menú del panel de clases file out emerge la clase completa, del menú del panel de protocolos file out emerge el protocolo completo, y del menú del panel de métodos file out emerge solo el método mostrado. Si bien esto puede parecer obvio, puede ser una fuente de confusión para los principiantes.

Posiblemente el ítem del menú mas útil es find class... (f) en el panel de packetes. Aunque las categorías son útiles para el código que estamos desarrollando activamente, la mayoría de nosotros no sabemos la categorización de todo el sistema, y es mucho mas rápido escribir CMD−f seguido por los primeros caracteres del nombre la clase que adivinar cual paquete podría estar adentro. recent classes... también puede ayudarte a volver rápidamente a una clase por la cual se ha navegado recientemente, incluso si haz olvidado

[3] pretty print (r) es el primer ítem en el panel de método, o en la mitad hacia abajo en el panel de código.

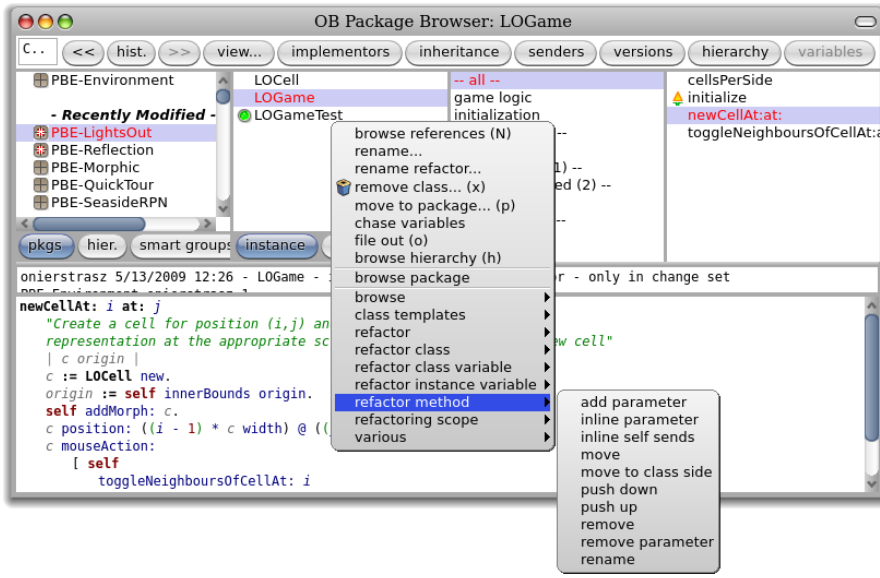


Figure 6.13: Operaciones de Refactorización.

su nombre.

También puedes buscar una clase o método específico escribiendo el nombre en el recuadro de búsqueda en el cuadro superior izquierdo del navegador. Cuando presiones return, una consulta será puesta en el sistema, y los resultados de la consulta serán mostrados. Tenga en cuenta que anteponiendo su consulta con #, puedes buscar las referencias a una clase o senders de un mensaje. Si estás buscando un método en particular de la clase seleccionada, es a menudo mas rápido para navegar por el protocolo *--all--* (que es el valor predeterminado), colocando el mouse en el panel del método, y escriba la primera letra del nombre del método que estás buscando. Esto usualmente desplazará el panel de modo que la búsqueda del nombre del método sea visible.

Trate de navegar en ambas formas para OrderedCollection»removeAt:

Hay muchas otras opciones disponibles en los menúes. Por eso es mejor pasar unos minutos trabajando con el navegador y ver lo que hay.

Compare el resultado de Browse Protocol, Browse Hierarchy, *y* Show Hierarchy *en el menú del panel de clases.*

Navegar mediante programación

La clase SystemNavigation proporciona una serie de métodos útiles que son prácticos para navegar por el sistema. Muchas de las funciones ofrecidas por el clásico navegador son implementadas por SystemNavigation.

 *Abrir un espacio de trabajo y evaluar el siguiente código para navegar por los emisores de* drawOn::

```
SystemNavigation default browseAllCallsOn: #drawOn:
```

Para restringir la búsqueda de los senders de los métodos de una clase específica:

```
SystemNavigation default browseAllCallsOn: #drawOn: from: ImageMorph
```

Debido a que las herramientas de desarrollo son objetos, que son completamente accesibles desde programas, puedes desarrollar tus propias herramientas o adaptar las herramientas existentes para tus necesidades.

El equivalente en programación al nemú `implementors` es:

```
SystemNavigation default browseAllImplementorsOf: #drawOn:
```

Para conocer más acerca de lo que está disponible, explorar la clase SystemNavigation con el navegador. Ejemplos adicionales de navegación pueden ser encontrados en las preguntas frecuentes FAQ (Appendix A).

6.3 Monticello

Hemos dado un rápido repaso de Monticello, el paquete de herramientas de Pharo, en Section 2.9. Sin embargo, Monticello tiene muchas mas características que se han discutido. Debido a que Monticello administra *paquetes*, antes de hablarles más acerca de Monticello, es importante que expliquemos primero exactamente lo que es un paquete .

Paquetes: Categorización declarativa de código Pharo

Hemos indicado anteriormente, en Section 2.3 los paquetes que son más o menos equivalentes a las categorías. Ahora vamos a ver exactamente cuál es la relación. El sistema de paquetes es un sencilla y simple manera de organizar el código fuente smalltalk que explota una simple convención de nombres para las categorías y protocolos.

Veamos esto utilizando un ejemplo. Suponga que está desarrollando un framework determinado con el fin de facilitar la utilización de bases de datos

relacionales de Pharo. Han decidido llamar al framework PharoLink, y han creado una serie de categorías que contienen todas las clases que se han escrito, ejemplo de categoría 'PharoLink−Connections' contiene OracleConnection MySQLConnection PostgresConnection y la categoría 'PharoLink−Model' contiene DBTable DBRow DBQuery, etc. Sin embargo, no todo el código residirá en estas clases. Por ejemplo, también puede tener una serie de métodos para convertir objetos en un formato amistoso de SQL:

```
Object»asSQL
String»asSQL
Date»asSQL
```

Estos paquetes pertenecen a los mismos paquetes como las clases en las categorías PharoLink−Connections y PharoLink−Model. Pero es evidente que el conjunto completo de la clase Object no pertenece a tu paquete! Por lo tanto necesitas una manera de poner ciertos *métodos* en un paquete, aunque el resto de la clase está en otro paquete.

La manera en que lo haces es poniendo esos métodos en un protocolo (de Object, String, Date, y así sucesivamente) llamado **PharoLink* (nota el primer asterisco). La combinación de las categorías *PharoLink-...* y los protocolos **PharoLink* forman un paquete denominado PharoLink. Para ser más precisos, las normas de lo que va en un paquete son de la siguiente manera.

Un paquete llamado Foo contiene:

1. todas las *definiciones de la clase* de las clases en la categoría *Foo*, o en las categorías que comienzan con *Foo-*, y
2. todos los *métodos* en *cualquier clase* en los protocolos llamados **Foo* o **foo*[4], o cuyo nombre comience con **Foo-* or **foo-*, y
3. todos los *métodos* en las clases de la categoría *Foo*, o en la categoría cuyo nombre comience con *Foo-*, *excepto* para esos nombres en los protocolos cuyos nombres comienzan con *.

Una consecuencia de estas reglas es que cada definición de clase y cada método pertenece a exactamente un paquete. La *excepción* en la última regla ha de estar allí porque esos métodos deben pertenecer a otros paquetes. La razón para ignorar la tipografía en la regla 2 es que, por convención, los nombres de los protocolos son normalmente (pero no necesariamente) nimúsculas (y puede incluir espacios), mientras los nombres de las categorías usan CamelCase (y no incluyen espacios).

La clase PackageInfo implementa estas reglas, y una manera de tener una sensación de ellos es experimentar con esta clase.

[4]Al realizar esta comparación, las letras mayúsculas y minúsculas en los nombres son ignoradas.

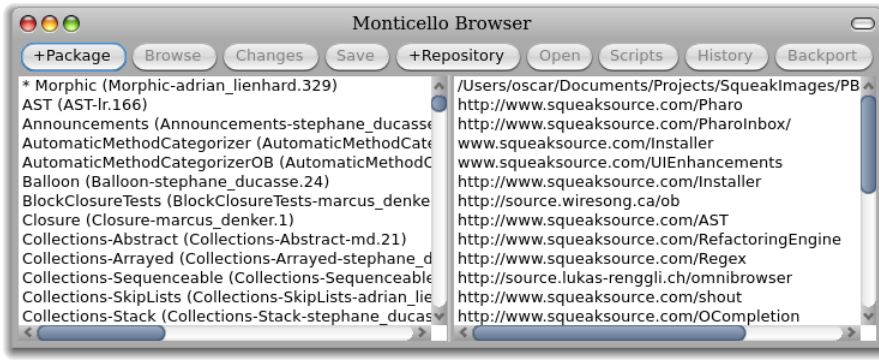


Figure 6.14: El navegador de Monticello.

Evalúe la siguiente expresión en un workspace:

```
mc := PackageInfo named: 'Monticello'
```

Ahora es posible mirar al interior de este paquete. Por ejemplo, imprimiendo mc classes en el panel del workspace devolverá una larga lista de clases que componen el paquete de Monticello. mc coreMethods devolverá una lista de MethodReferences para todos los métodos en esas clases. mc extensionMethods es quizás una de las más interesantes consultas: esta devolverá una lista de todos los métodos contenidos en el paquete Monticello pero no contenido dentro de una clase Monticello.

Los paquetes son una adición relativamente nueva a Pharo, pero ya que las convenciones de nombres de paquetes se basaron en los que estaban en uso, es posible usar PackageInfo para analizar código más antiguo que no ha sido explícitamente adaptado para trabajar con él.

Print (PackageInfo named: 'Collections') externalSubclasses*; esta expresión responderá con una lista de todas las subclases de* Collection *que* no *están en el paquete* Collections.

Monticello básicos

Monticello debe su nombre al hogar de Thomas Jefferson, una mansión neoclásica construíada en la cima de una montaña , tercer presidente de los Estados Unidos y autor del Estatuto de libertad Religiosa de Virginia. El nombre significa "montículo" en Italiano, y por lo tanto, se pronuncia siempre con una "c" italiana, que suena como el "ch" en chair: Mont-y'-che-llo.

Al abrir el navegador de Monticello, verás dos paneles de lista y una fila

de botones, como se muestra en Figure 6.14. El panel de la izquierda muestra una lista de todos los paquetes que han sido cargados en la imagen que estás ejecutando; la versión particular del paquete se muestra entre paréntesis después del nombre.

El panel de la derecha muestra una lista de todos los repositorios de código fuente que Monticello tiene conocimiento, usualmente porque se ha cargado el código desde ellos. Si seleccionas un paquete en el panel de la izquierda, el panel de la derecha se filtra para mostrar únicamente los repositorios que contienen las versiones del paquete seleccionado.

Uno de los repositorios es un directorio llamado *package-cache*, el cual es un subdirectorio del directorio en el cual estás corriendo la imagen. Cuando cargas el código desde o escribes código a un repositorio remoto, una copia es guardada en el cache del paquete. Esto puede ser útil si la red no esta disponible y necesitas acceder a un paquete. También, si le das un archivo Monticello (.mcz) directamente, por ejemplo como un archivo adjunto al correo electrónico, la forma más conveniente de acceder a este es colocarlo en el directorio del package-cache.

Para agregar un nuevo repositorio a la lista, clic en el +Repository, y elegir el tipo de repositorio en el menú emergente. Vamos a agregar un repositorio HTTP.

Abir Monticello, clic en +Repository*, y seleccione* HTTP*. Editar el diálogo para leer:*

```
MCHttpRepository
   location: 'http://squeaksource.com/PharoByExample'
   user: ''
   password: ''
```

Entonces haga clic en Open para abrir el navegador de repositorios en este directorio. Deberías ver algo como Figure 6.15. A la izquierda hay una lista de todos los paquetes en el repositorio; si seleccionas uno, en el panel de la derecha se mostrarán todas las versiones del paquete seleccionado en este repositorio.

Si haz seleccionado una de las versiones, puedes usar el botón Browse (sin cargarlo en tu imagen), botón Load , o buscar con el botón Changes que se harán a tu imagen por la carga de la versión seleccionada. Puedes también hacer un Copy de la versión del paquete, que luego puedes escribir en otro repositorio.

Como puedes ver, los nombres de las versiones contienen el nombre del paquete, las iniciales del autor de la versión, y un número de versión. El nombre de la versión es también el nombre del archivo en el repositorio. Nunca se deben cambiar estos nombres; la correcta operación de Monticello

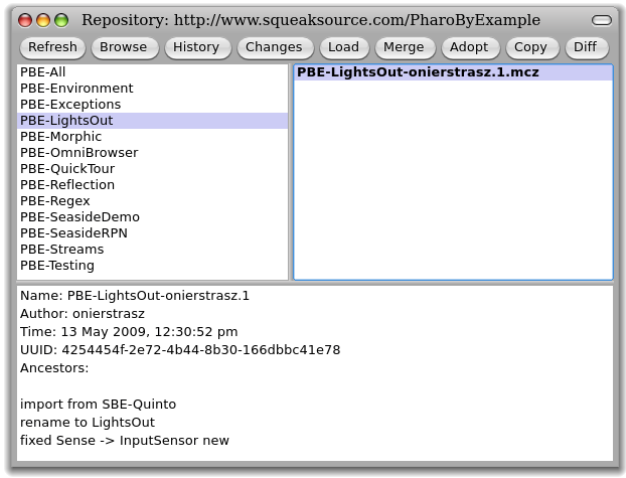


Figure 6.15: Un navegador de Repositorio.

depende de ellos! Los archivos de versiones Monticello son solo archivos zip, y si eres curioso puedes descomprimirlos con una herramienta zip, pero la mejor manera de mirar su contenido es usando el mismo Monticello.

Para crear un paquete con Monticello, tienes que hacer dos cosas: escribir algún código, y decile a Monticello acerca de esto.

Crear un paquete llamado PBE-Monticello, *y poner un par de clases en este, como se muestra en Figure 6.16. También, crear un método en una existente clase, tal como* Object, *y ponerlo en el mismo paquete como sus clases, usando las reglas de la página 124 — ver Figure 6.17.*

Para decirle a Monticello acerca de tus paquetes, clic en el botón +Package, y escriba el nombre del paquete, en este caso "PBE". Monticello agregará PBE a su lista de paquetes; la entrada del paquete será marcada con un asterisco para mostrar que la versión en la imagen todavía no se ha escrito a cualquier repositorio. Tenga en cuenta que ahora deberás tener dos paquetes en Monticello, uno llamado PBE y el otro llamado PBE−Monticello. Esto esta bien, porque PBE contendrá PBE−Monticello, y otros paquetes a partir de PBE−.

Inicialmente, el único repositorio asociado con este paquete será su cache del paquete, como se muestra en Figure 6.18. Esto esta bien: todavía puedes guardar el código, lo que causará que se escriba el el package-cache. Simplemente haz clic en el botón Save y serás invitado a proporcionar un registro de la versión del paquete que estás a punto de guardar, como se muestra en Figure 6.19; cuando aceptas el mensaje, Monticello guardará tu paquete.

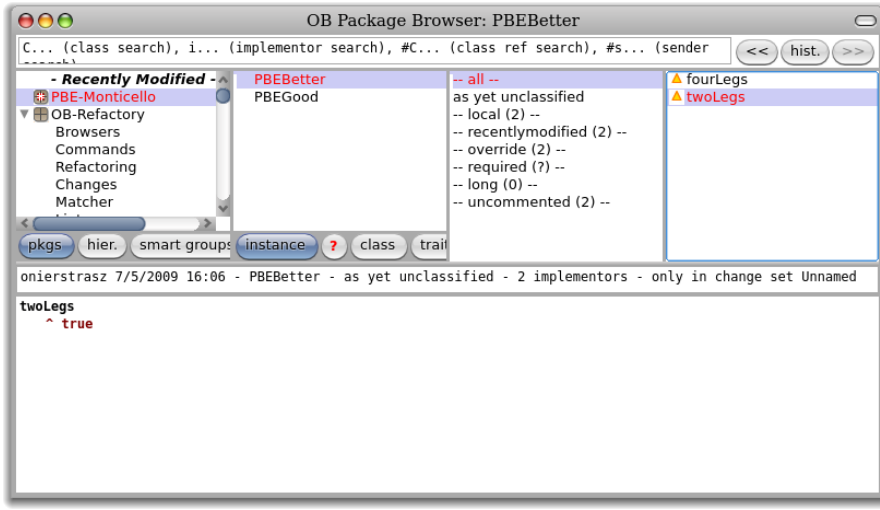


Figure 6.16: Dos clases en el paquete “PBE”.

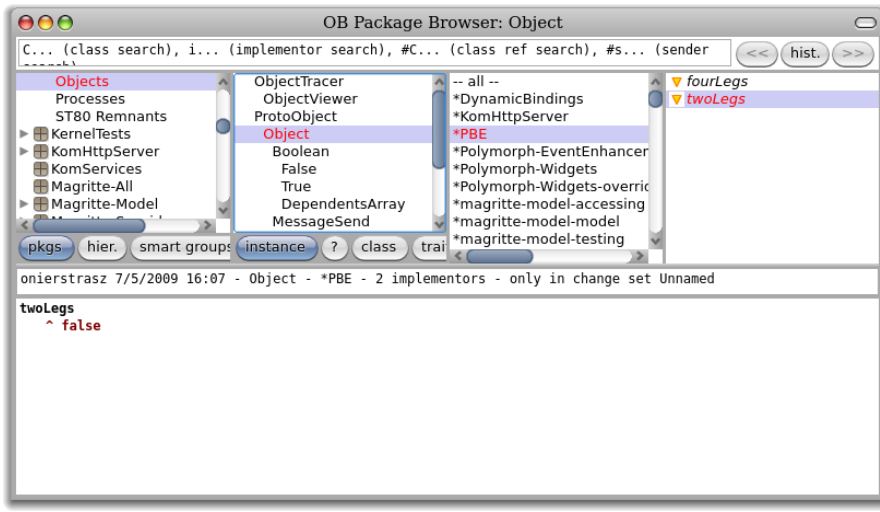


Figure 6.17: Una extensión del método que también está en el paquete “PBE”.

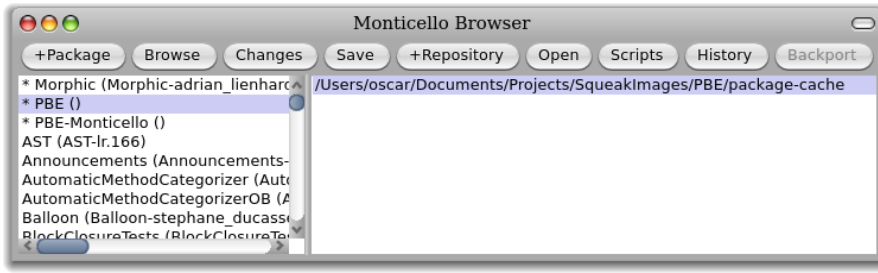


Figure 6.18: El paquete PBE aún no guardado en Monticello.

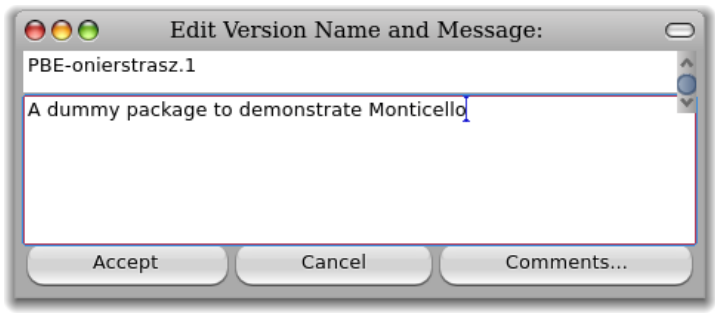


Figure 6.19: Registrando un mensaje para una nueva versión de un paquete.

Para indicar esto, el asterisco decorando el nombre en el paquete en el panel de Monticello será eliminado, y se agregará el número de la versión.

Si a continuación haces un cambio en el paquete — por ejemplo agregando un método a una de las clases — el asterisco volverá a aparecer, mostrando que tienes cambios que no han sido guardados. Si abres un navegador de repositorio en el caché de paquetes, puedes seleccionar la versión guardada, y usar el botón Changes y otros botones. También puedes por su puesto guardar la nueva versión al respositorio; una vez refrescada con el botón Refresh la vista del repositorio, esta debería verse como Figure 6.20.

Para guardar el nuevo paquete de un repositorio que no sea el package-cache, es necesario que te asegures primero que Monticello sepa sobre el repositorio, agregándolo si es necesario. Entonces cuando usas el botón Copy en el navegador del repositorio de package-cache, y seleccionas el repositorio al cual el paquete debería ser copiado. También puedes asociar el repositorio deseado con el paquete action-clicking en el repositorio y seleccionando add to package ... , como se muestra en Figure 6.21. Una vez que el paquete sabe acerca de un repositorio, puedes guardar una nueva versión

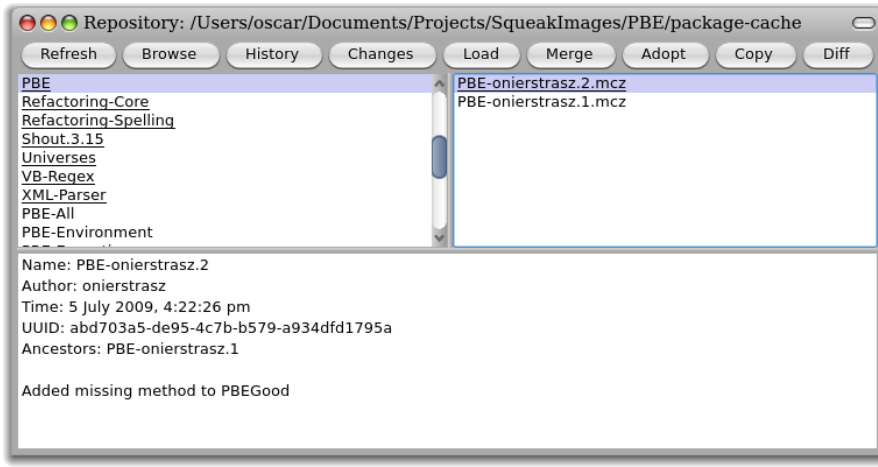


Figure 6.20: Dos versiones de nuestro paquete están ahora en el package-chache.

seleccionando el repositorio y el paquete en el navegador de Monticello, y dando un clic en el botón Save. Por supuesto , debes tener permiso para escribir a un repositorio. El repositorio PharoByExample en *SqueakSource* puede ser leído por cualquier usuario pero no modificado, si tratas de guardar allí, recibirás un mensaje de error. Sin embargo, puedes crear tu propio repositorio en SqueakSource usando la interface web en http://www.squeaksource.com, y usar esta para guardar tu trabajo. Esto es especialmente útil como un mecanismo para compartir tu código con amigos, o si usas múltiples computadores.

Si tratas de guardar a un repositorio donde no tienes permiso de escritura, una versión se escribirá al package-cache de todas maneras. Así puedes recuperar mediante la edición de la información del repositorio (action-click en el navegador Monticello) o escogiendo un diferente repositorio, y entonces usando el botón Copy desde el navegador package-cache.

6.4 El inspector y el Explorador

Una de las cosas que hace tan diferente a Smalltalk de muchos otros entornos de programación es que este te proporciona una ventana a un mundo de objetos vivos, no a un mundo de código estático. Cualquiera de estos objetos puede ser examinado por el programador, e, incluso, cambiar — aunque algún tipo de cuidado es necesario cuando se cambian objetos básicos que soportan el sistema. Experimente por todos los medios, pero guarde la imagen

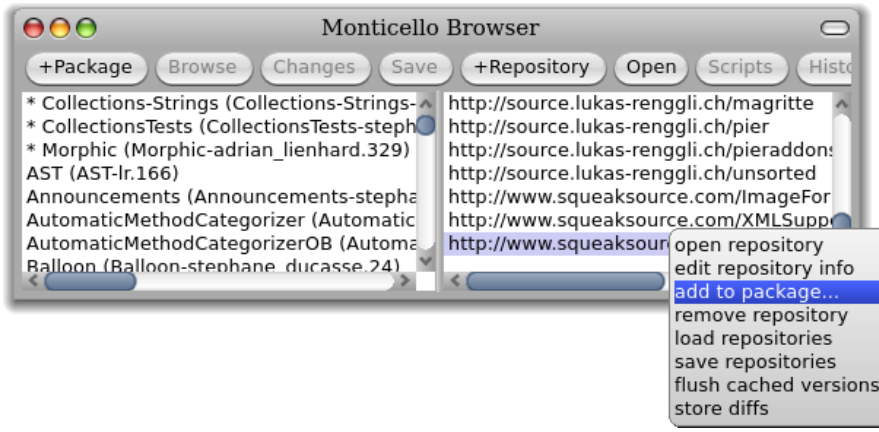


Figure 6.21: Agregando un repositorio al conjunto de repositorios asociados con el paquete.

primero!

El Inspector

Como una ilustración de lo que se puede hacer con un inspector, escriba TimeStamp now *en un workspace, y entonces action-click y seleccione* inspect it.

(No es necesario seleccionar el texto antes de utilizar el menú; si no hay texto seleccionado, las operaciones del menú trabajan en la totalidad de la línea actual. También puedes escribir CMD–i para inspect .)

Una ventana como la mostrada en Figure 6.22 aparecerá. Este es un inspector, y se puede considerar como una ventana al funcionamiento interno de un objeto determinado — en este caso, una instancia en particular de TimeStamp que se creó cuando evaluaste la expresión TimeStamp now. La barra de título de la ventana muestra la representación imprimible del objeto que está siendo inspeccionado. Si seleccionas self en la parte superior del panel de la izquierda, el panel de la derecha mostrará el printstring del objeto. El panel de la izquierda muestra una vista de árbol del objeto, con self en la raíz. Las Variables de instancia pueden ser exploradas expandiendo los triángulos junto a sus nombres.

El panel horizontal en la parte inferior del inspector es una pequeña ventana en el workspace. Esto es útil porque en esta ventana, la pseudo-variable self esta enlazada al objeto que haz seleccionado en el panel izquierdo. Por lo tanto, si inspect it en

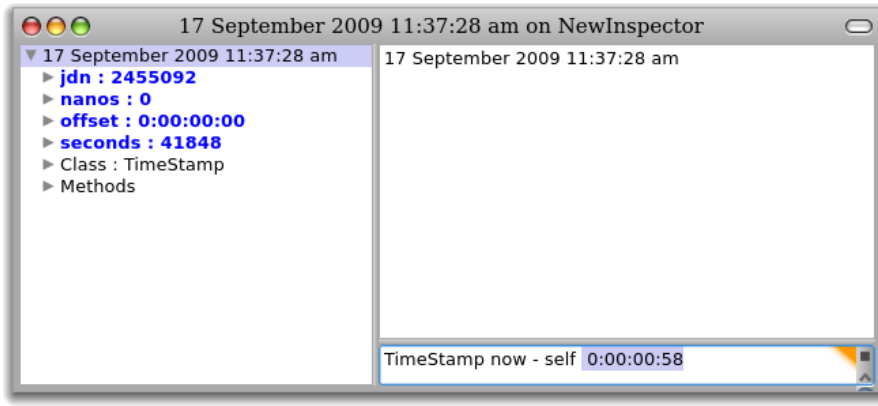


Figure 6.22: Inspeccionando TimeStamp now

```
self − TimeStamp today
```

en el panel del workspace, el resultado será un objeto Duration que representa el intervalo de tiempo entre la medianoche de hoy y el instante en que estás evaluando la expresión TimeStamp now y ha creado el objeto TimeStamp que estás inspeccionando. También puedes tratar de evaluar TimeStamp now − self ; esto te dirá cuánto tiempo haz dedicado a leer esta sección de este libro!

Además de self, todas las variables de instancia del objeto están en el alcance del panel del workspace, para que puedas utilizarlo en expresiones o incluso asignárselas. Por ejemplo, si seleccionas el objeto raíz en el panel izquierdo y evalúas jdn := jdn − 1 en el panel del workspace, puedes ver el valor de la instancia de la variable jdn de hecho cambia, y el valor de TimeStamp now − self se incrementará en un día.

Existen variantes especiales del inspector para los Diccionarios, OrderedCollections, CompiledMethods y algunas otras clases que facilitan la tarea de examinar el contenido de estos objetos especiales.

El Explorador de Objetos

El *explorador de objetos* es conceptualmente similar al inspector, pero presenta su información en forma diferente. Para ver la diferencia, *exploraremos* el mismos objeto que estábamos inspeccionando.

 *Seleccione* self *en el panel izquierdo del inspector, luego action-click y escoja* explore (I) .

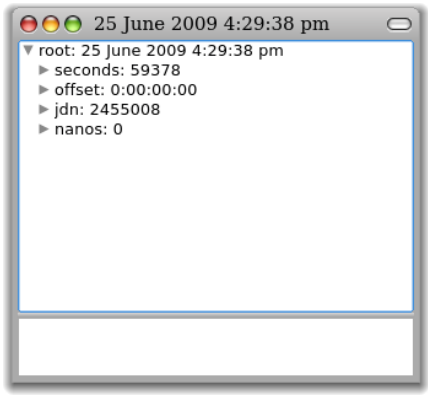


Figure 6.23: Explorando TimeStamp now

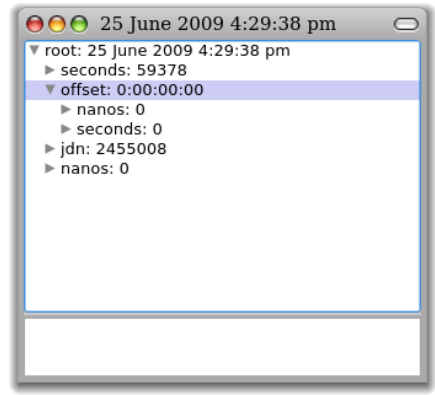


Figure 6.24: Explorando las variables de instancia

La ventana del explorador se ve como Figure 6.23. Si haces clic en el diminuto triángulo que se encuentra junto a root, la vista cambiará a Figure 6.24, que muestra las variables de instancia del objeto que estás explorando. Haga clic en el triángulo que se encuentra junto a offset, y verá *sus* variables de instancia. El explorador es realmente útil cuando necesitas explorar un estructura jerárquica compleja — de ahí el nombre.

El panel del workspace del explorador de objetos trabaja ligeramente diferente a la del inspector. self no está enlazado al objeto root, sino más bien para el objeto que esta seleccionado actualmente; las variables de instancia del objeto seleccionado están también dentro del alcance.

Para ver el valor del explorer, usémoslo para explorar profundamente la estructura de objetos anidados.

Evalúe Object explore *en un workspace.*

Este es el objeto que representa la clase Object en Pharo. Tenga en cuenta que puedes navegar directamente a los objetos representando el diccionario de métodos e incluso los métodos compilados de esta clase (ver Figure 6.25).

6.5 El Depurador

El depurador es probablemente la herramienta más poderosa en la suite de herramientas de Pharo. Se utiliza no sólo para la depuración, sino también para escribir nuevo código. Para demostrar el depurador, comencemos por crear un error!

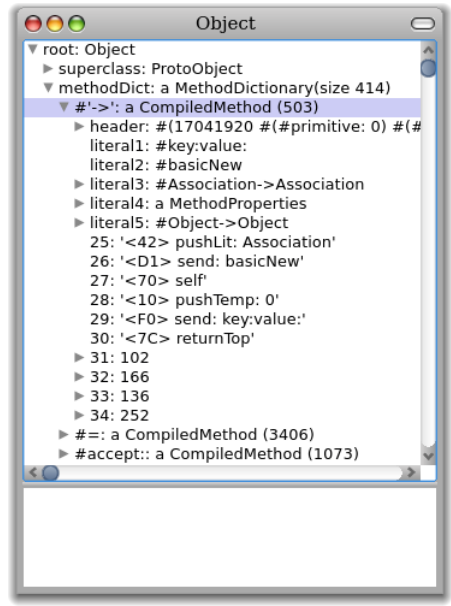


Figure 6.25: Explorando un ExploreObject

Usando el navegador, agregue el siguiente método a la clase String*:*

Method 6.1: *Un método defectuoso*

```
suffix
    "Se supone que soy un nombre de archivo, y respondo con mi sufijo, la parte despu\'
        es del \'ultimo punto"
    | dot dotPosition |
    dot := FileDirectory dot.
    dotPosition := (self size to: 1 by: −1) detect: [ :i | (self at: i) = dot ].
    ↑ self copyFrom: dotPosition to: self size
```

Por supuesto, estamos seguros que ese método trivial funcionará, así que en vez de escribir una prueba SUnit, simplemente escribiremos `'readme.txt' suffix` en un workspace y `print it (p)`. Qué sorpresa! En vez de la esperada respuesta 'txt', aparecerá un pops up PreDebugWindow, como aparece en Figure 6.26.

La PreDebugWindow tiene una barra de título que nos indica que ha ocurrido un error, y nos muestra una *traza de la pila* de los mensajes que condujo al error. A partir de la parte inferior de la traza, UndefinedObject»DoIt representa el código que se compila y se ejecuta cuando lo seleccionamos `'readme.txt' suffix` en el workspace y pedimos a Pharo imprimirlo `print it`. Este código,

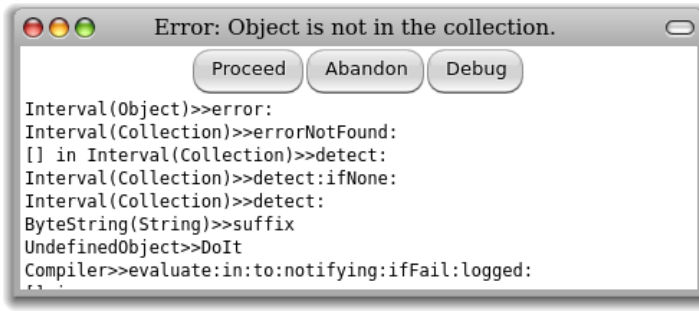


Figure 6.26: Una PreDebugWindow nos notifica un error.

por supuesto, envió el mensaje suffix a un objeto ByteString ('readme.txt'). Esto causó que el heredado método suffix en la clase String se ejecute; toda esta información se codifica en la siguiente línea de la traza de la pila, ByteString (String)»suffix. Trabajando en la pila, podemos ver que suffix envió detect:... y eventualmente detect:ifNone envió errorNotFound.

Para averiguar el *por qué* el punto no fue encontrado, necesitamos el depurador en sí, así que haga clic en el botón Debug.

El depurador es mostrado en Figure 6.27; parece intimidante en un primer momento, pero es bastante fácil de usar. La barra de título y la parte superior del panel son muy similares a los que vimos en la PreDebugWindow. Sin embargo, el depurador combina la traza de la pila con un navegador de métodos, así cuando se selecciona una línea en la traza de la pila, el correspondiente método se muestra en el panel inferior. Es importante darse cuenta que la ejecución que provocó el error se encuentra en su imagen, pero en un estado de suspensión. Cada línea de la traza de la pila representa un frame en la pila de ejecución que contiene toda la información necesaria para continuar con la ejecución. Esto incluye todos los objetos involucrados en el cálculo, con sus variables de instancia, y todas las variables temporales de los métodos de ejecución.

En Figure 6.27 hemos seleccionado el método detect:ifNone: en la parte superior del panel. El cuerpo del método se muestra en el panel central; el resaltado en azul alrededor del mensaje value muestra que el método actual ha enviado el mensaje value y está esperando una respuesta.

Los cuatro paneles en la parte inferior del depurador son realmente dos mini-inspectores (sin paneles de workspace). El inspector de la izquierda muestra el objeto actual, es decir, el objeto llamado self en el panel central. Como selecciones diferentes frames de la pila, la identidad de self puede cambiar, y lo mismo ocurrirá con el contenido del self-inspector. Si haces clic en self en la parte inferior del panel de la izquierda, verás que self es el

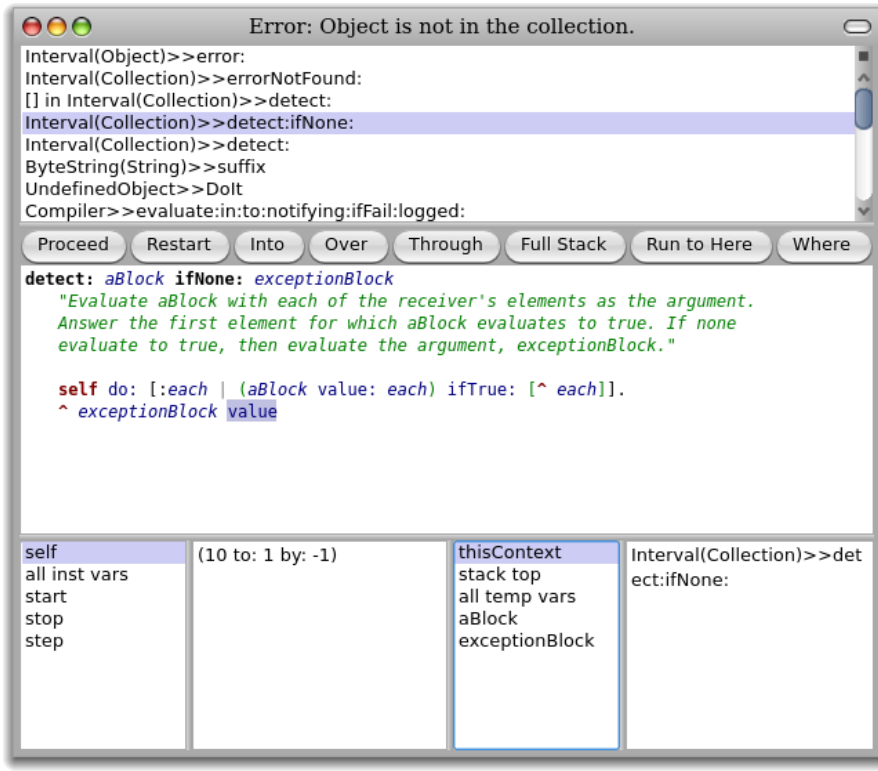


Figure 6.27: El Depurador.

intervalo (10 to: 1 by −1), que es lo que esperamos. Los paneles del workspace no son necesarios en los mini-inspectores del depurador porque todas las variables están también en el ámbito del panel de métodos; debes sentirte libre de escribir o seleccionar expresiones en este panel y evaluarlos. Siempre puedes cancel (l) tus cambios usando el menú o CMD−*l*.

El inspector de la derecha muestra las variables temporales del contexto actual. En Figure 6.27, value fue enviado al parámetro exceptionBlock.

Como podemos ver un método abajo en la traza de la pila, el exceptionBlock es [self errorNotFound: ...], así, no es de raro que veamos el correspondiente mensaje de error.

Por otra parte, si deseas abrir un completo inspector o explorador en una de las variables mostradas en los mini-inspectores, simplemente haga doble clic en el nombre de la variable, o seleccione el nombre de la variable y action-click escoja inspect (i) o explore (I). Esto puede ser útil si quieres ver cómo una variable cambia mientras ejecutas otro código.

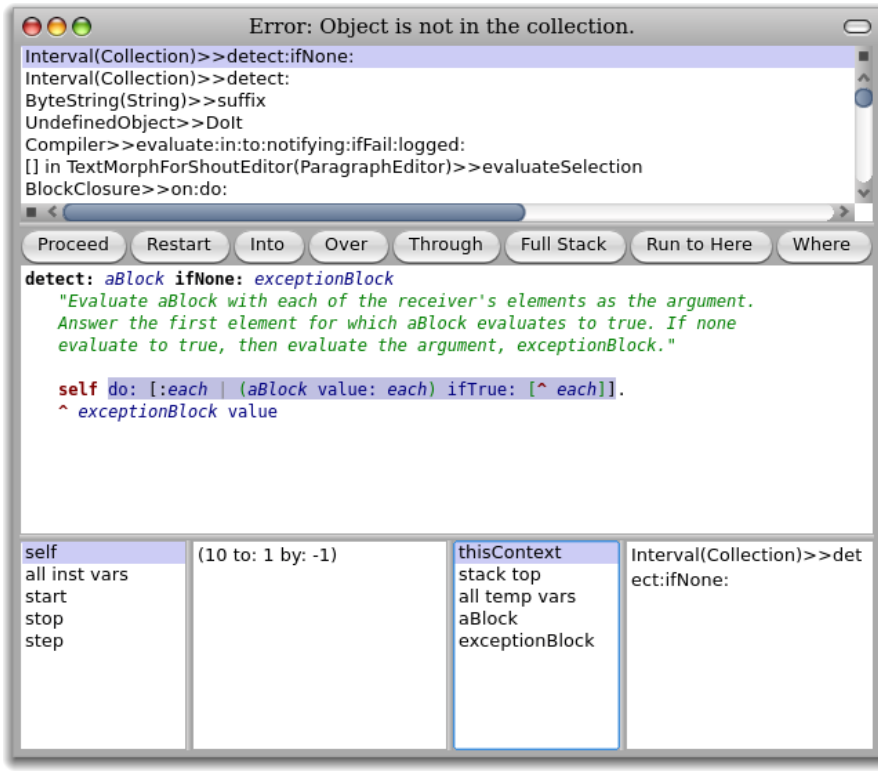


Figure 6.28: El depurador después de reiniciar el método detect: ifNone:

Mirando hacia atrás a la ventana del método, vemos que esperábamos la penúltima línea del método para encontrar el punto en el string 'readme.txt', y que la ejecución nunca debería haber llegado a la última línea. Pharo no nos permite correr una ejecución hacia atrás, pero esto nos permite comenzar un método de nuevo, que funciona muy bien en el código, como este que no muta objetos, pero en su lugar crea otros nuevos.

Clic Restart*, y verás que el lugar de ejecución vuelve a la primera declaración del método actual. El resaltado en azul muestra que el próximo mensaje a ser enviado será* do: *(ver Figure 6.28).*

Los botones Into y Over nos brindan dos diferentes maneras de paso a través de la ejecución. Si hace clic en el botón Over, Pharo ejecuta el actual envío de mensaje (en este caso el do:) en un paso, a menos que exista un error. Así Over nos llevará al próximo envío-mensaje en el actual método, el cual es value — este está exactamente donde comenzamos, y no ayudan mucho. Lo

que debemos hacer es averiguar porque el do: no está encontrando el caracter que estamos buscando.

Después de hacer clic en Over, *hacer clic en* Restart *para volver a la situación mostrada en Figure 6.28.*

Clic Into; *Pharo ingresará en el método correspondiente al envío de mensaje resaltado, en este caso,* Collection»do:.

Sin embargo, resulta que esto no es una gran ayuda: podemos estar bastante seguros de que Collection»do: no es lo que rompe. El error es mucho más probable que esté en *que* estamos pidiendo hacer a Pharo. Through es el botón apropiado para utilizar en este caso: queremos ignorar los detalles de do: en sí y centrarse en la ejecución del argumento del bloque.

Seleccione el método detect:ifNone: *de nuevo y el botón* Restart *para volver al estado mostrado en Figure 6.28. Ahora haga clic en* Through *un par de veces. Seleccione* each *en la ventana del contexto, al hacerlo, verás en* each *una cuenta regresiva desde* 10 *en la medida que se ejecuta el método* do: .

Cuando each es 7 esperamos que el bloque ifTrue: sea ejecutado, pero no es así. Para ver qué es lo que va mal, vaya a Into la ejecución de value: como se ilustra en Figure 6.29.

Después de hacer clic en el botón Into, nos encontramos en la posición indicada en Figure 6.30. Esto parece a primera vista que hemos ido *de vuelta* al método suffix , pero esto es porque ahora estamos ejecutando el bloque que suffix proporciona como argumento a detect:. Si seleccionas dot en el contexto del inspector, verás que su valor es '.'. Y ahora puedes ver por qué ellos no son iguales: el séptimo carácter de 'readme.txt' es por supuesto un Character, mientras que dot es un String.

Ahora que vemos el error, la solución es obvia: tenemos que convertir dot a un caracter antes de comenzar a buscarlo.

Cambiar el código en el depurador de manera que la asignación se lea dot := FileDirectory dot first *y* accept *los cambios.*

Dado que estamos ejecutando el código dentro de un bloque que se encuentra dentro de detect:, varios frames de pilas tendrán que ser abandonados con el fin de realizar este cambio. Pharo nos pregunta si esto es lo que requemos (ver Figure 6.31), y, asumiendo que hacemos clic en yes, se guardará (y compilará) el nuevo método.

La evaluación de la expresión 'readme.txt' suffix terminará, e imprime la respuesta '.txt'.

Es la respuesta correcta? Desafortunadamente, no lo podemos decir con certeza. El sufijo debe ser .txt o txt? El comentario del método en suffix no

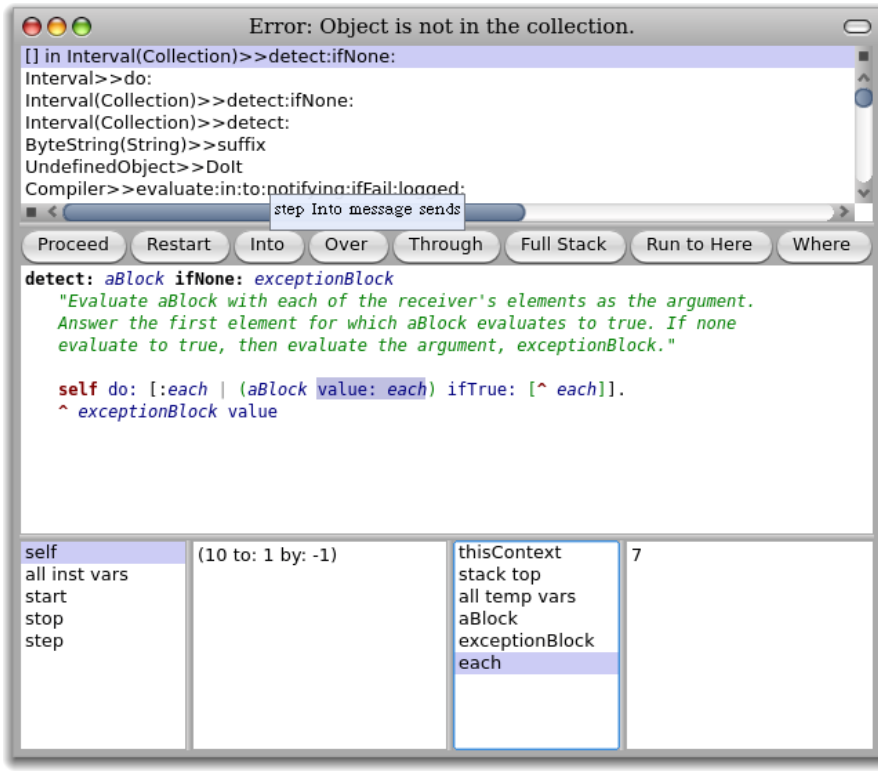


Figure 6.29: El depurador después de avanzar varias veces A través de del método do:

es muy preciso. La manera de evitar este tipo de problema es escribir una prueba en SUnit que defina la respuesta.

Method 6.2: *Una prueba simple para el método* suffix

```
testSuffixFound
   self assert: 'readme.txt' suffix = 'txt'
```

El esfuerzo requerido para hecerlo fue un poco mas que para ejecutar la misma prueba en el workspace, pero usando SUnit guarda el archivo ejecutable de la prueba como documentación , y hace que sea fácil para los demás para que se ejecute. Por otra parte, si agregas el method 6.2 a la clase StringTest y ejecutar este conjunto de pruebas con SUnit, puedes rápidamente volver a depurar el error. SUnit abre el depurador en la afirmación de error (assertion failure), pero solamente necesitas regresar por la pila una trama, Restart la prueba e ir Into del método suffix , y puedes corregir el error, como lo estamos haciendo en Figure 6.32. Solamente existe un segundo trabajo al

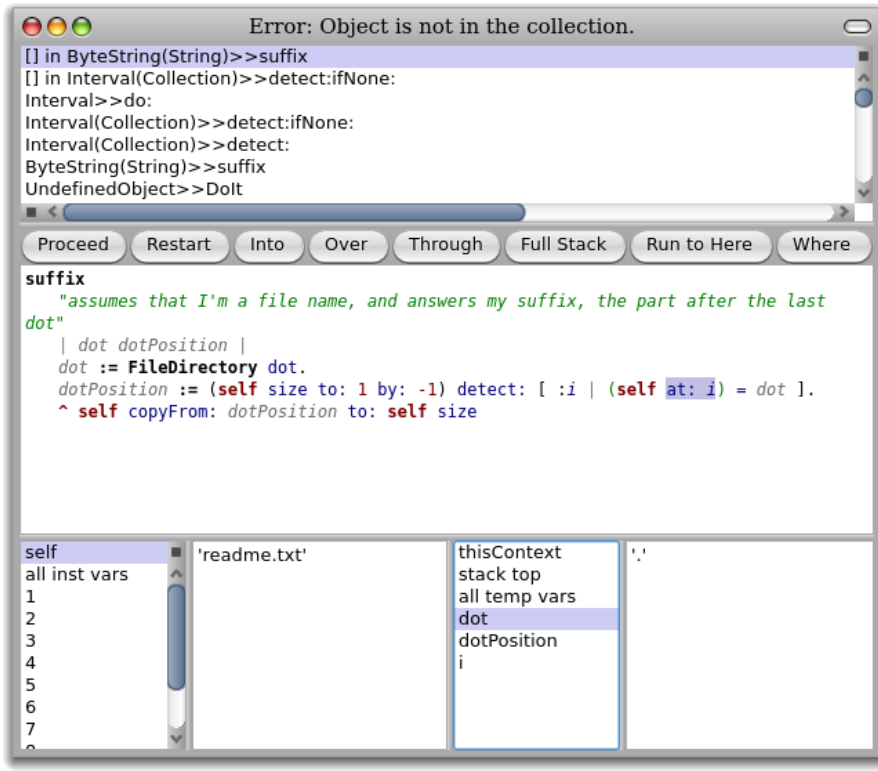


Figure 6.30: El depurador mostrando porque 'readme.txt' at: 7 no es igual a dot

hacer clic en el botón Run Failures en la unidad de pruebas SUnit, y confirmar que la prueba ahora pasa.

Here is a better test:

Method 6.3: *Una mejor prueba para el método* suffix

```
testSuffixFound
   self assert: 'readme.txt' suffix = 'txt'.
   self assert: 'read.me.txt' suffix = 'txt'
```

Por qué esta es una mejor prueba? Porque esta le dice al lector lo que el método debe hacer si hay mas de un punto en el string de destino.

Hay algunas otras maneras de entrar en el depurador además de detectar los errores y afirmaciones de errores. Si ejecutas el código que entra en un bucle infinito, puedes interrumpirlo y abrir un depurador en el cálculo escribiendo CMD−. (eso es una detención completa o un período, dependiendo

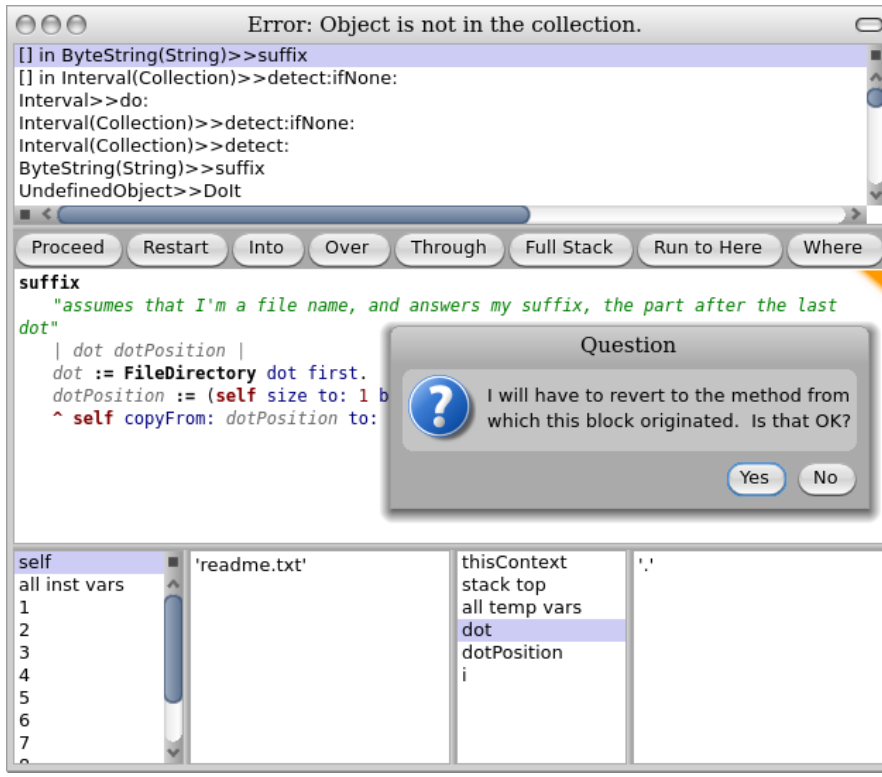


Figure 6.31: Cambiando el método suffix en el depurador: solicitando la confirmación de la salida desde un bloque interior

donde aprendió Ingles).[5] También puedes solo editar el código sospechoso a insertar self halt. Así, por ejemplo, podríamos editar el método suffix para leer de la siguiente manera:

Method 6.4: *Insertando un* halt *en el método* suffix.

```
suffix
    "se supone que soy un nombre de archivo, y respondo mi sufijo, la parte despues del
        ultimo punto"
    | dot dotPosition |
    dot := FileDirectory dot first.
    dotPosition := (self size to: 1 by: −1) detect: [ :i | (self at: i) = dot ].
    self halt.
    ↑ self copyFrom: dotPosition to: self size
```

[5] También es útil saber que puede hacer aparecer un depurador de emergencia en cualquier momento escribiendo CMD−SHIFT−.

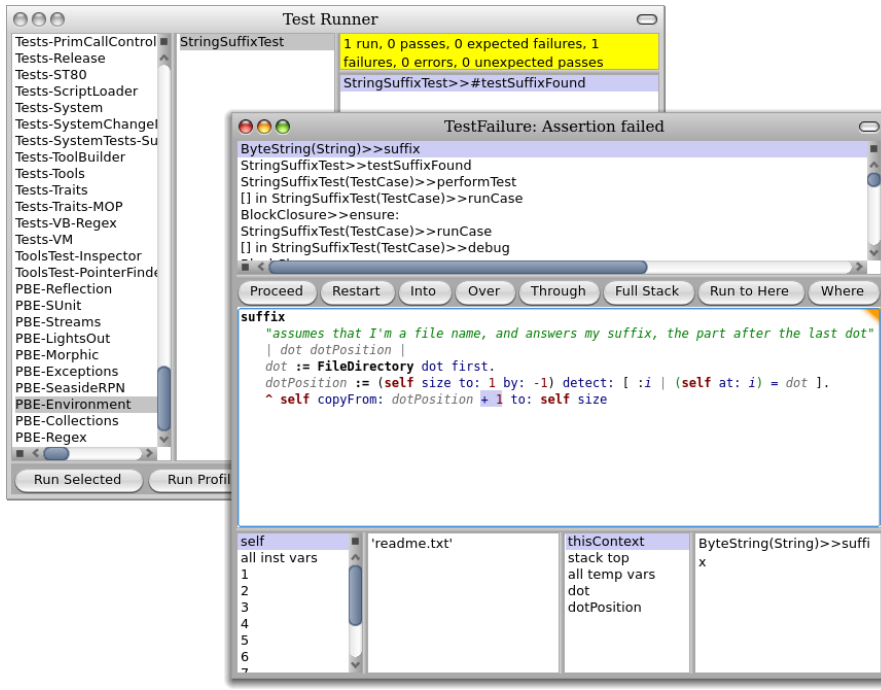


Figure 6.32: Cambiando el método suffix en el debugger: corrigiendo el error off-by-one después de una SUnit de afirmación de error (assertion failure)

Cuando corremos este método, la ejecución de self halt iniciará el predebugger, desde donde podemos proceder, o ir al depurador y mirar las variables, dar pasos en el cálculo, y editar el código.

Eso es todo lo que hay en el depurador, pero no esto lo que hay para el método suffix. El primer fallo debería haber hecho que te dieras cuenta que si no ha un punto en el string de destino, el método suffix provocará un error. Este no es el comportamiento que queremos, así que vamos a agregar una segunda prueba para especificar qué es lo que debería ocurrir en este caso.

Method 6.5: *Una segunda prueba para el método* suffix*: el objetivo no tiene sufijo*

```
testSuffixNotFound
    self assert: 'readme' suffix = ''
```

 *Agregar method 6.5 al conjunto de pruebas en la clase* StringTest*, y ver que la prueba provoca un error. Ingrese al depurador seleccionando la prueba errónea en SUnit, y edite el código de manera que la prueba sea satisfactoria. La forma mas fácil y clara de hacer esto es reemplazar el mensaje* detect: *por* detect: ifNone:*, donde*

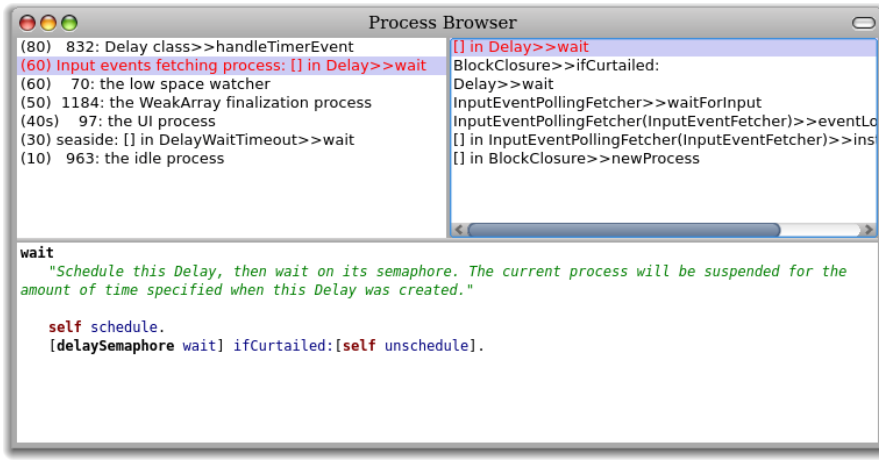


Figure 6.33: El Navegador de Procesos

el segundo argumento es un block que devuelve simplemente el tamaño del string.

Vamos a aprender más sobre SUnit en Chapter 7.

6.6 El Navegador de Procesos

Smalltalk es un sistema multi-hebras: hay muchos procesos ligeros (también conocidos como hebras) ejecutándose simultáneamente en tu imagen. En el futuro la máquina virtual de Pharo puede tomar ventaja de los multiprocesadores cuando estén disponibles, pero en la actualidad la concurrencia esta implementada por división de tiempo (time-slicing).

El procesos es un primo del depurador que te permite echar un vistazo a los diversos procesos que se ejecutan dentro de Pharo. Figure 6.33 muestra un pantallazo. La parte superior del panel de la izquierda lista todos los procesos en Pharo, en orden de prioridad, de la interrupción del temporizador vigilante con prioridad 80 al proceso inactivo con prioridad 10. Por supuesto, en un monoprocesador, el único proceso que puede estar corriendo cuando miras es el proceso UI; todos los demás estarán esperando por algún tipo de evento. Por defecto, la vizualización del proceso es estática; este puede ser actualizado por action-clicking y seleccionando `turn on auto-update (a)`

Si seleccionas un proceso en la parte superior del panel de la izquierda, la traza de la pila se muestra en la parte superior del panel de la derecha, al igual que con el debugger. Si seleccionas un frame de la pila, el método correspondiente se muestra en el panel inferior. El navegador de procesos no

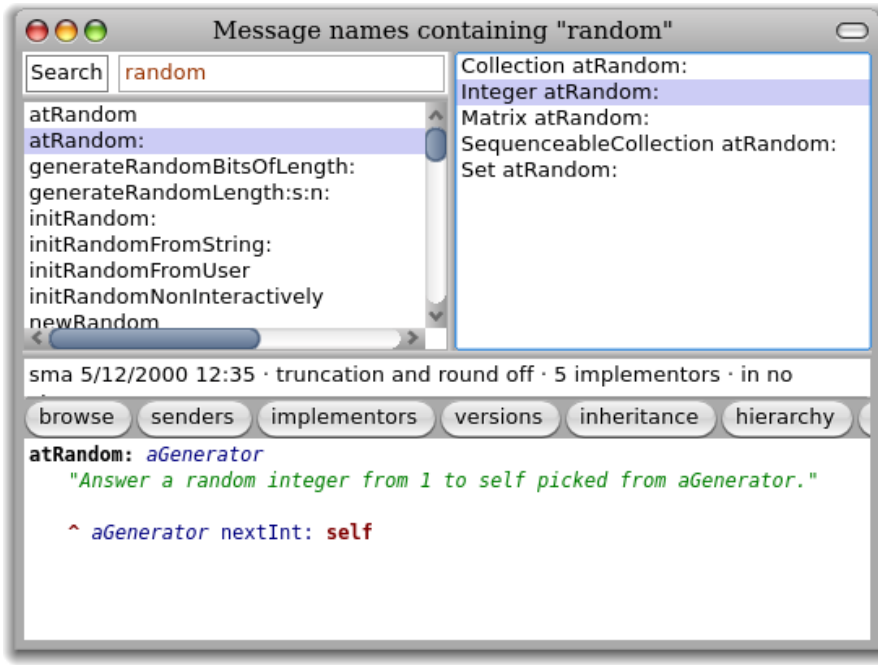


Figure 6.34: El navegador de nombres de mensajes mostrando todos los métodos que contengan el substring random en sus selectores.

esta equipado con mini-inspectores para self y thisContext, pero action-clicking en los frames de la pila proporciona funcionalidad equivalente.

6.7 Buscando métodos

Existen dos herramientas en Pharo para ayudarte a encontrar mensajes. Difieren tanto en funcionalidad como en interfaz.

El *method finder* fue descrito con algún detalle en Section 1.9; puedes usarlo para encontrar métodos por nombre o por funcionalidad. Sin embargo, para mirar el cuerpo de un método, el buscador de métodos abre un nuevo explorador. Esto puede ser abrumador.

El navegador *message names* tiene más limitada funcionalidad de búsqueda: puedes escribir un fragmento de selector de mensajes en el cuadro de búsqueda, y el navegador lista todos los métodos que contienen ese fragmento en sus nombres, como se muestra en Figure 6.34. Sin embargo, es un navegador completo: Si seleccionas uno de los nombres en el

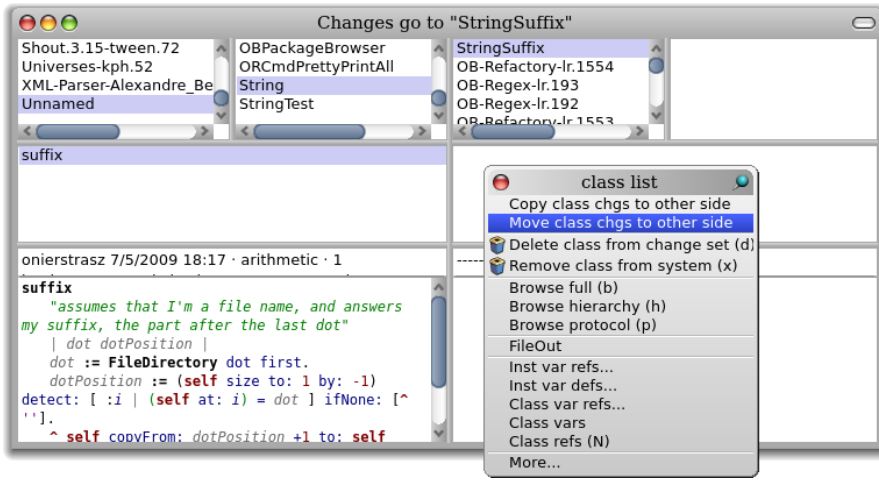


Figure 6.35: The Change Sorter

panel izquierdo , todos los métodos con ese nombre son listados en el panel derecho, y pueden ser consultados en el panel inferior. Al igual como ocurre con el navegador, el navegador de mensajes de nombres tiene una barra de botones que puede ser usada para abrir otros navegadores sobre el método seleccionado o sus clases.

6.8 Change Set y Change Sorter

Cuando estas trabajando en Pharo, cualquier cambio que hagas a los métodos y clases son almacenados en un Change Set change set. Esto incluye crear nuevas clases, renombrar clases, cambiar categorías, agregar métodos a clases ya existentes — todo aquello que sea significativo. De cualquier modo, los *doits* arbitrarios no son incluídos, entonces si, por ejemplo, creas una nueva variable global asignandola en un workspace, la creación de la variable no aparecerá en el archivo change set.

En cualquier momento,existen muchos Change sets, pero solo uno de ellos — ChangeSet current — está juntando los cambios que estan siendo realizados en la imagen. Puedes ver cuál Change Set esta activo y examinar todos los conjuntos de cambios usando Change Sorter, disponible al seleccionar World ▷ Tools . . . ▷ Change Sorter.

Figure 6.35 muestra este navegador. La barra de títulos muestra que Change set está activo, este change set está seleccionado cuando el change sorter es abierto.

Otros change sets pueden ser seleccionados en el panel superior izquierdo; el menu action-clicking te permite configurar a un change set distinto como activo, o crear un nuevo change set. El siguiente panel lista todas las clases afectadas por el change set seleccionado (con sus categorías). Al seleccionar una de esas clases se muestra los nombres de aquellos métodos que estan también incluidos en el change set (*no* todos los métodos en la clase) en el panel central izquierdo, y seleccionando el nombre de un método se muestra la definición del método en el panel inferior. Adviertase que el change sorter *no* muestra si la creación de la clase misma es parte del change set, aunque esta información es alamacenada en la estructura de objetos que es usada para representar el change set.

El change sorter también te permite borrar clases y métodos desde el change set haciendo action-clicking en los items corrrespondientes.

El change sorter te permite ver simultáneamente dos change sets, uno a la izquierda y otro a la derecha. Esta disposición responde a la mayor funcionalidad del change sorter, que es la habilidad de mover o copiar cambios de un set a otro, como se muestra action-clicking en Figure 6.35. También es posible copiar métodos individuales de un lado a otro.

Te estarás preguntando porque debería importarte la composición de un change set.La respuesta es que los change sets proveen un mecanismo sencillo para exportar código desde Pharo al sistema de archivos, desde donde puede ser importado en otra imagen de Pharo, on en otro no-Pharo Smalltalk. La exportación de un change set es conocida como“filing-out”, y puede lograrse action-clicking en cualquier change set, clase o método en cualquier navegador. File outs repetidos crean nuevas versiones del archivo, pero los change sets no son una herramienta de versionado como Monticello: ellos no mantienen registro de las dependencias.

Antes de la llegada de Monticello, los change set eran los principales medios para intercambiar código entre usuarios de Pharo. Tienen la ventaja de la simplicidad (el file out es simplemente un archivo de texto, aunque *no*te recomendamos que trates de editarlo con un editor de texto), y un grado de portabilidad.

La principal desventaja de los change set, comparados con los paquetes de Monticello, es que no soportan la noción de dependencias. Un change set al que se le ha hecho un file out es un conjunto de *acciones* que cambian cualquier imagen en la que sea cargado. Cargar exitosamente un change set requiere que la imagen esté en u estado apropiado. Por ejemplo, el change set podría contener una acción para agregar un método a una clase; eso sólo puede ser logrado si dicha clase ya está definida en la imagen. En forma similar, el change set podría renombrar o re-categorizar una clase, lo cual obviamente solo funcionará si la clase está presente en la imagen; los métodos podrían usar variables de instancia que fueron declaradas cuando se realizo el file out, pero que no existen en la imagen a la cual son importadas.El prob-

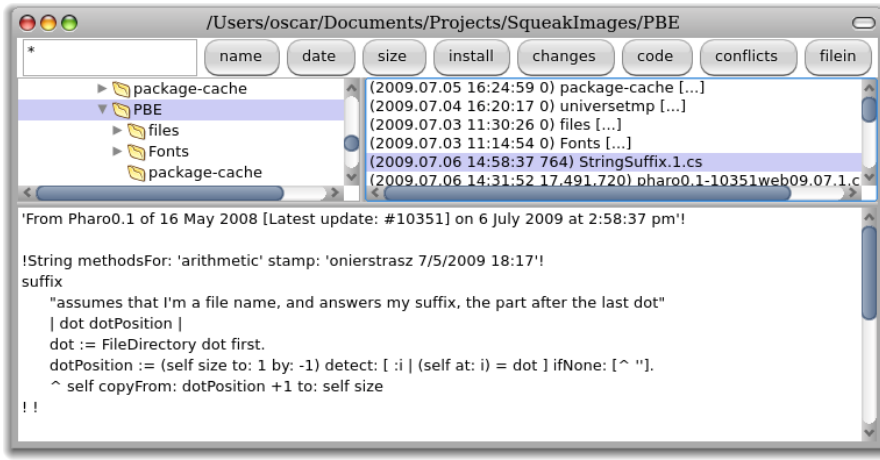


Figure 6.36: A file list browser

lema es que los change sets no describen explícitamente las condiciones bajo las cuales pueden ser importados: el proceso de importación solo espera lo mejor,usualmente resultando en un mensaje críptico de error y una traza de pila cuando las cosas salen mal.Aún si la importación funciona, un change set podría deshacer silenciosamente un cambio hecho por otro change set.

En contraste, los paquetes de Monticello representan código de forma declarativa: ellos describen el estado en el que la imagen debería estar después de ser cargados.Esto permite a Monticello advertirte sobre conflictos (cuando dos paquetes requieren estados finales contradictorios)y ofrecerte cargar series de paquetes en orden de dependencia.

A pesar de estos defectos,los change sets aún tienen sus usos; en particular, puedes encontrar change sets en Internet que quieras mirar y tal vez usar. Entonces,habiendo exportado un change set usando el change sorter, ahora te diremos como importarlo.Esto requiere el uso de otra herramienta, el navegador de lista de archivos.

6.9 El Navegador de Lista de Archivos

El navegador de lista de archivos es de hecho una herramienta de propósito general para navegar en el sisema de archivos(y también servidores FTP) desde Pharo. Puedes abrirlo desde el menu World ▷ Tools . . . ▷ File Browser. Lo que veas depende por supuesto del contenido de tu sistema de archivos local, pero una vista típica es mostrada en Figure 6.36.

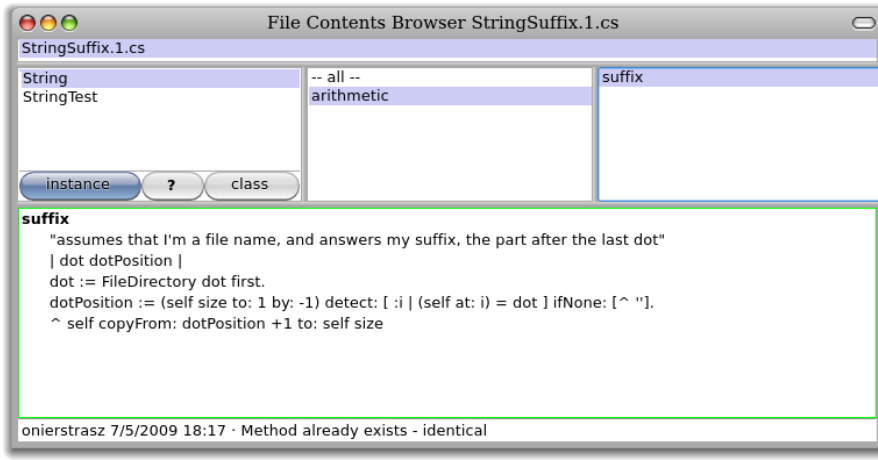


Figure 6.37: A File Contents Browser

Cuando abres por primera vez un navegador de lista de archivos estará enfocado en el directorio actual, esto es,aquel desde el cual iniciaste Pharo. La barra de títulos muestra la dirección hacia este directorio.El panel más largo a la izquierda puede ser usado para navegar el sistema de archivos de modo convencional.Cuando un directorio es seleccionado,los archivos que contiene (pero no los directorios) son mostrados a la derecha. Esta lista de archivos puede ser filtrada ingresando un patrón estilo Unix en el pequeño recuadro en la esquina superior izquierda de la ventana.Inicialmente, este patrón es *, el cual coincide con todos los nombres de archivo, pero puedes tipear una cadena de caracteres ahi y hacer accept it, cambiando el patrón. (Nótese que un * es implícitamente añadido adelante y concatenado al patrón que tipeaste.) El criterio de ordenamiento de los archivos puede ser cambiado usando los botones name, date y size. El resto de los botones depende del nombre del archivo seleccionado en el navegador. En Figure 6.36, el nombre del archivo tine el sufijo .cs, por lo tanto el navegador asume que es un change set, y provee botones para instalarlo install(el cual lo *importa* a un nuevo change set cuyo nombre se deriva del nombre del archivo), para navegar por los cambios changes en el archivo, para examinar el código code en el archivo, y para importar filein el código al change set *actual*. Podrías pensar que el botón conflicts te mostraría cosas sobre aquellos cambios en el change que generaron conflictos con el código existente en la imagen, pero no lo hace.

En cambio solo chequea potenciales problemas en el archivo que podrían indicar que éste no puede ser cargado correctamente (como la presencia de avances de línea).

Debido a que la elección de botones a mostrar depende del *nombre* del archivo, y no de su contenido, algunas veces el botón que quieres no estará en la pantalla. De todos moodos, el conjunto completo de opciones está siempre disponible desde el menu action-click more ... ,asi que puedes resolver este problema fácilmente.

El botón code es quizás el más útil para trabajar con change sets; abre un navegad del archivo del change set; un ejemplo se muestra en la Figure 6.37. El navegador de contenidos de archivos es similar al navegador excepto que no muestra categorias,solo clases, protocolos y métodos. Para clase, el navegador te dirá si la la clase ya existe en el sistema y si está definida en el archivo (pero *no* si las definiciones son idénticas). Mostrará los métodos en cada clase, y (como se muestra en Figure 6.37) te mostrará las diferencias entre la versión actual y la versión en el archivo. Los items del menu contextual en cada uno de los cuatro paneles te permirán también importar el change set entero, o la correspondiente clase, protocolo, o método.

6.10 En Smalltalk, no puedes perder código

Es muy posible hacer caer Pharo: como un sistema experimental, Pharo te permite cambiar cualquier cosa, incluyendo cosas que son vitales para que Pharo trabaje!

Para maliciosamente hacer caer Pharo, intente Object become: nil.

La buena noticia es que nunca perderás todo el trabajo, aún si cae y vuelve a la última versión guardada de tu imagen, la cual podría haber sido horas atrás. Esto es porque todo el código que haz ejecutado es guardado en el archivo *.changes*. Integramente! Esto incluye una línea de entrada que evalúas en un workspace, así como el código que haz agregado a una clase durante la programación.

Así que aquí están las instrucciones sobre cómo obtener el código de nuevo. No hay necesidad de leer esto hasta que lo necesites. Sin embargo, cuando lo necesitas, lo encontrarás aquí esperándote.

En el peor caso, puedes usar un editor de texto en el archivo *.changes*, pero dado que este tiene muchos megabytes, esto puede ser lento y no es recomendado. Pharo le ofrece una mejor manera.

Cómo obtener el código de vuelta

Reinicie Pharo desde el más reciente snapshot, y seleccione World ▷ Tools ... ▷ Recover lost changes.

Esto te dará la oportunidad para decidir cuánto tiempo atrás en la historia deseas examinar. Normalmente, basta con examinar los cambios hasta el último snapshot. (Puedes conseguir el mismo efecto mediante la edición de `ChangeList browseRecent: 2000` por lo que el número `2000` se vuelve algo más, mediante prueba y error.)

Uno tiene un navegador *recent changes*, mostrando, digamos, cambios nuevos hasta tu último captura, tendrás una lista de todo lo que haz hecho en Pharo durante ese tiempo. Puedes borrar los ítems desde esta lista usando el menú action-click. Cuando estés satisfecho, puedes archivar lo que queda, por lo tanto incorporando los cambios en tu nueva imagen. Es una buena idea comenzar un nuevo change set, usando el ordinario navegador del change set, antes de hacer el archivo, de manera que todo tu código recuperado será un nuevo change sets. Puedes entonces sacar este change sets.

Lo útil que se pueda hacer en el navegador *recent changes* es `remove doIts`. Usualmente, no querrás entrar en el archivo (y así re-ejecutar) doIts. Sin embargo, hay una excepción. Creando una clase aparece como un `doIt`. *Antes que puedas ingresar en un archivo los métodos de una clase, la clase debe existir.* Por lo tanto, si haz creado cuaquiera nueva clase, *primero* ingresa en el archivo la creación de la clase doIts, luego `remove doIts` e ingresa al archivo los métodos.

Cuando termino con la recuperación, me gusta sacar del archivo mi nuevo changes set, salir de Pharo sin guardar la imagen, reiniciar, y asegurarse que los nuevos archivos con el changes set vuelvan de manera limpia.

6.11 Resumen del Capítulo

Con el fin de desarrollar eficazmente con Pharo, es importante invertir algún esfuerzo en aprender las herramientas disponibles en el ambiente.

- El *navegador* estándar es tu principal interfaz para navegar por las existentes categorías, clases, métodos y protocolos de métodos, y para definir nuevos. El navegador ofrece varios botones prácticos para saltar directamente a los senders o implementors de un mensaje, versiones de un método, y así sucesivamente.
- Existen varios navegadores diferentes (tales como el OmniBrowser y el navegador de Refactorización), y varios navegadores especializados (tales como el navegador jerárquico) que proporciona distintos puntos de vista de métodos y clases.
- Desde cualquiera de las herramientas, puedes resaltar el nombre de una clase o un método e inmediatamente saltar a un navegador usando un acceso rápido de teclado CMD–b.

- También puedes navegar en el sistema Smalltalk programáticamente enviando mensajes por defecto a SystemNavigation .

- *Monticello* es una herramienta para exportación, importación, versionamiento y compartir paquetes de clases y métodos. Un paquete de Monticello consiste de una categoría, subcategoría, y protocolos relacionados de métodos en otras categorías.

- El *inspector* y el *explorador* son dos herramientas que son útiles para explorar e interactuar con objetos vivos en tu imagen. Incluso puedes inspeccionar herramientas por meta-clicking para iniciar el halo de su morphic y seleccionando el manejador de depuración .

- El *debugger* es una herramienta que no solo te permite inspeccionar la pila en tiempo de ejecución de tu programa cuando se genera un error, sino que también te permite interactuar con todos los objetos de tu aplicación, incluído el código fuente. En muchos casos, puedes modificar el código fuente desde depurador y continuar la ejecución. El debugger es especialmente efectivo como una herramienta de apoyo a las primeras pruebas de desarrollo junto a SUnit (Chapter 7).

- El *navegador de procesos* te permite monitorear, consultar e interactuar con los procesos actuales que están corriendo en tu imagen.

- El *buscador de métodos* y el *navegador de nombres de mensajes* son dos herramientas para localizar métodos. El primero de ellos es más útil cuando no estás seguro del nombre , pero no saben el comportamiento esperado. La segunda ofrece una interfaz de navegación mas avanzada cuando conoces al menos un fragmento del nombre.

- Los *Change Sets* son registros de todos los cambios en el código fuente de tu imagen que se generan automáticamente . En gran parte han sido reemplazados por Monticello como un medio para almacenar e intercambiar las versiones de tu código fuente, pero aún son útiles, especialmente para la recuperación de fallas catastróficas, sin embargo es muy raro que esto suceda.

- El *navegador de lista de archivo* es una herramienta que permite navegar por el sistema de archivos. También te permite hacer filein de código fuente del sistema de archivos.

- En el caso de que tu imagen se caiga antes de que puedas guardarla o respaldar tu código fuente con Monticello, siempre puedes recuperar tus cambios más recientes usando un *navegador de lista de cambios*. Puedes seleccionar los cambios que deseas reproducir y presentarlos en la copia más reciente de tu imagen.

Chapter 7

SUnit

7.1 Introducción

SUnit es un pequeño pero poderoso framework que cuenta con soporte para la creación y despliegue de pruebas. Como puede intuirse por su nombre, el diseño de SUnit se enfoca en las *Unit Tests* (pruebas unitarias), pero también puede ser utilizado para pruebas de integración y pruebas funcionales. SUnit fue originalmente desarrollado por Kent Beck y posteriormente ampliado por Joseph Pelrine entre otros, para integrar la noción de un recurso, el cual describiremos en Section 7.6.

El interés en las pruebas y en el Desarrollo guiado por pruebas -Test Driven Development no está limitado a Pharo o a Smalltalk. Las pruebas automatizadas se han convertido en un sello del movimiento del desarrollo de software Ágil (Agile software development), y cualquier desarrollador que se ocupe de la mejora de la calidad de software hará bien en adoptarlo. Es más, muchos programadores en diferentes lenguajes han llegado a apreciar el poder de las pruebas unitarias, y ahora existen versiones de xUnit para muchos lenguajes, tales como Java, Python, Perl, .Net y Oracle.

Este capítulo describe SUnit 3.3 (la versión más reciente al momento de editar este texto); el sitio web oficial de SUnit es sunit.sourceforge.net, donde pueden encontrarse otras actualizaciones.

Ni las pruebas, ni la construcción de conjuntos de pruebas, es nuevo: todos saben que las pruebas son una buena forma de capturar errores.

La Programación eXtrema (eXtreme Programming), que hace de las pruebas una práctica central y enfatiza las pruebas automatizadas (*automated* tests), ha ayudado a que el desarrollo de pruebas se convierta en algo productivo y entretenido, y no en una tarea más que a los programadores les disguste.

La comunidad Smalltalk tiene una larga tradición en la construcción de pruebas porque el estilo de desarrollo incremental está respaldado por su ambiente de desarrollo.

En el desarrollo tradicional en Smalltalk, el programador escribiría pruebas en un workspace tan pronto como un método se termina de escribir. A veces una prueba puede ser incorporada como un comentario en la cabecera del método sobre el cual se está trabajando, o las pruebas que necesiten alguna configuración pueden ser incluídas como métodos de ejemplo en la clase. El problema con estas prácticas es que estas pruebas no están disponibles para otros programadores que modifiquen el código; los comentarios y los métodos de ejemplo son mejores en ese sentido, pero aún no existe una manera fácil de hacer un seguimiento de ellos y de hacer que corran automáticamente. Las pruebas que no se ejecutan no ayudan a encontrar bugs!

Más aún, un método de ejemplo no informa al lector sobre el resultado esperado: puedes correr un ejemplo y ver —quizás con gran sorpresa— el resultado, pero no sabrás si el comportamiento observado es el correcto.

SUnit es valioso porque nos permite escribir pruebas que son chequeadas por sí mismas (self-checking): la prueba misma define cuál debería ser el resultado correcto. También nos ayuda a organizar pruebas en grupo, para describir el contexto en el cual las pruebas deben correrse, y a correrlas en conjunto automáticamente.

En menos de dos minutos puedes escribir pruebas usando SUnit, así que en lugar de escribir pequeños fragmentos de código en un workspace, te alentamos a usar SUnit y obtener todas las ventajas de las pruebas acumulables y automáticamente ejecutables.

En este capítulo comenzaremos debatiendo por qué hacemos pruebas y qué es lo que hace a un buen desarrollo de pruebas. Luego presentamos una serie de pequeños ejemplos mostrando cómo usar SUnit.

Por último, miraremos la implementación de SUnit, y de esa forma entender cómo Smalltalk usa el poder de la reflexión en el soporte de sus herramientas.

7.2 Por qué es importante el desarrollo y la ejecución de las pruebas

Desafortunadamente, muchos desarrolladores creen que el desarrollo de pruebas es una pérdida de tiempo. Después de todo, *ellos* no escriben bugs — sólo los *otros* programadores lo hacen. La mayoría de nosotros ha dicho, en algún momento: "Yo escribiría pruebas si tuviera más tiempo." Si nunca escribieras bugs, y si tu código nunca fuera a ser cambiado en el futuro, en-

tonces las pruebas sí serían una pérdida de tiempo. Sin embargo, esto probablemente significa que tu aplicación es trivial, o que ya no es usada ni por ti ni por nadie más. Piensa en las pruebas como una inversión para el futuro: tener un paquete de pruebas será bastante útil ahora, pero será *extremadamente* útil cuando tu aplicación, o el entorno en el cual se ejecuta, cambie en el futuro.

Las pruebas juegan muchos roles. Primero, proveen documentación sobre la funcionalidad que cubren. Más aún, la documentación es activa: ver que las pruebas pasan correctamente te dice que la documentación está actualizada.

Segundo, las pruebas ayudan a los desarrolladores a confirmar que los cambios que le acaban de hacer a un paquete no ha roto nada más en el sistema — y a encontrar las partes que se rompen cuando la confianza resulta estar fuera de lugar.

Finalmente, escribir las pruebas al mismo tiempo — o antes — de programar te obliga a pensar sobre la funcionalidad que quieres diseñar, *y cómo debería mostrarse al cliente*, y no acerca de cómo implementarla.

Escribiendo las pruebas primero — antes que el código — estás obligado a indicar el contexto en el que la funcionalidad se ejecutará, la forma en la que va a interactuar con el código cliente, y los resultados esperados.

Tu código mejorará: pruébalo.

No podemos probar todos los aspectos de una aplicación realista. Cubrir una aplicación completa es sencillamente imposible y no sería el objetivo del desarrollo de pruebas. Incluso con un buen conjunto de pruebas algunos bugs aún quedarán en la aplicación, donde permanecerán latentes, esperando la oportunidad para dañar su sistema. Si encuentras que esto ha ocurrido, ¡aprovechálo! Tan pronto como descubras un bug, escribe una prueba que lo exponga, ejecuta la prueba, y mírala fallar. Ahora puedes comenzar a arreglar el bug: la prueba te dirá cuándo has terminado.

7.3 ¿Qué hace a una buena prueba?

Escribir buenas pruebas es una habilidad que puede ser aprendida facilmente con práctica. Veamos las propiedades que las pruebas deberían tener para obtener el máximo beneficio:

1. Las pruebas deben poder repetirse. Debes ser capaz de correr una prueba las veces que lo desees, y siempre obtener la misma respuesta.

2. Las pruebas deben correr sin la intervención humana. Debes ser capaz incluso de correrlas durante la noche.

3. Las pruebas deben contar una historia. Cada prueba debería cubrir un aspecto de alguna pieza de código. Una prueba debería actuar como un escenario al que tú o alguien más pueda leer para entender una parte de la funcionalidad.

4. Las pruebas deberían tener una frecuencia de cambio menor a la de la funcionalidad que cubren: no deberías tener que cambiar todas tus pruebas cada vez que modificas la aplicación. Una forma de lograr esto es escribiendo pruebas a base de interfases públicas de la clase que estás probando. Es correcto escribir una prueba para un método "de ayuda" privado si crees que ese método es lo suficientemente complicado para necesitar una prueba, pero debes tener en cuenta que dicha prueba puede que sea cambiada, o directamente desechada, cuando se te ocurra una mejor implementación.

Una consecuencia de la propiedad (3) es que el número de pruebas debería ser algo proporcional al número de funciones a ser probadas: cambiando un aspecto del sistema no debería romper todas las pruebas excepto por un número limitado de ellas. Esto es importante porque teniendo 100 pruebas fallando debería enviar un mensaje más fuerte que teniendo 10 pruebas fallando.

Sin embargo, no siempre es posible alcanzar este caso ideal: en particular, si un cambio rompe la inicialización de un objeto, o el set-up de una prueba, seguramente causará que las otras pruebas fallen.

eXtreme Programming aboga por escribir pruebas antes de escribir el código. Esto parace ir en contra de nuestros instintos como desarrolladores de software. Todo lo que podemos decir es: adelante e inténtalo! Hemos encontrado que escribir pruebas antes del código nos ayuda a saber qué queremos codificar, nos ayuda a saber cuándo hemos terminado, y nos ayuda a visualizar la funcionalidad de una clase y a diseñar su interfaz.

Más aún, el desarrollo basado en escribir las pruebas primero nos da el ánimo para ir rápido, porque no tenemos miedo de estar olvidándonos algo importante.

7.4 SUnit by example

Antes de adentrarnos en detalles de SUnit, mostraremos un ejemplo paso a paso.

Usaremos un ejemplo que prueba la clase `Set`. Intenta copiar el código como sigue.

Paso 1: crear la clase de prueba

 *Primero debes crear una nueva subclase de* TestCase *llamada* ExampleSetTest. *Agrega dos variables de instancia de forma tal que la clase luzca así:*

Class 7.1: *Ejemplo de una clase de prueba de Set.*

```
TestCase subclass: #ExampleSetTest
   instanceVariableNames: 'full empty'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MySetTest'
```

Usaremos la clase ExampleSetTest para agrupar todas las pruebas relacionadas con la clase Set. La misma define el contexto en el cual las pruebas correrán. Aquí el contexto es descripto por las dos variables de instancia full y empty que usaremos para representar el set lleno y el set vacío.

El nombre de la clase no es crítico, pero por convención debería terminar con Test. Si defines una clase llamada Pattern y nombras a la correspondiente clase de pruebas PatternTest, las dos clases serán indexadas alfabéticamente en forma conjunta en el navegador (asumiendo que ambas se encuentran en la misma categoría). Es de *suma importancia* que tu clase sea una subclase de TestCase.

Paso 2: inicializar el contexto de prueba

El método setUp define el contexto en el cual las pruebas correrán, es una especie de método de inicialización. setUp es invocado antes de la ejecución de cada método de prueba definido en la clase de pruebas.

 *Define el método* setUp *como sigue, para inicializar la variable* empty *que se refiere a un set vacio y a la variable* full *para referirse a un set que contiene elementos.*

Method 7.2: *Setting up a fixture*

```
ExampleSetTest»setUp
   empty := Set new.
   full := Set with: 5 with: 6
```

En la jerga del desarrollo de pruebas, el contexto es llamado el *fixture* para la prueba.

Paso 3: escribir algunos métodos de prueba

Vamos a crear algunas pruebas definiendo algunos métodos en la clase ExampleSetTest. Cada método representa una prueba: el nombre del método

debería comenzar con la cadena 'test' para que SUnit la tome en su juego de pruebas.

Los métodos de pruebas no toman argumentos.

Define los siguientes métodos de prueba.

La primer prueba, llamada testIncludes, prueba al método includes: de Set. La prueba dice que enviando el mensaje includes: 5 a un conjunto que contiene 5 elementos debería retornar verdadero –true. Claramente, esta prueba depende de que el método setUp ya haya sido corrido.

Method 7.3: *Probando miembro de Set*

```
ExampleSetTest»testIncludes
  self assert: (full includes: 5).
  self assert: (full includes: 6)
```

La segunda prueba, llamada testOccurrences, verifica que el número de ocurrencias de 5 en el set full es igual a uno, aún cuando agregamos otro elemento 5 al set.

Method 7.4: *Probando ocurrencias*

```
ExampleSetTest»testOccurrences
  self assert: (empty occurrencesOf: 0) = 0.
  self assert: (full occurrencesOf: 5) = 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) = 1
```

Finalmente, probamos que el conjunto ya no contiene al elemento 5 después de que lo hemos removido.

Method 7.5: *Probando eliminar*

```
ExampleSetTest»testRemove
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Nota el uso del método deny: para afirmar que algo no debería ser verdadero.

aTest deny: anExpression es equivalente a aTest assert: anExpression not, pero mucho más legible.

Paso 4: correr las pruebas

La forma más fácil de correr las pruebas es directamente desde el navegador. Simplemente action-click sobre el paquete, nombre de clase o sobre un método de prueba individual, y selecciona run the tests(t)

Los métodos de prueba serán marcados con rojo o verde, dependiendo si pasaron o no, y la clase será marcada completamente o parcialmente en verde o rojo dependiendo de si todas, algunas o ninguna prueba pasó.

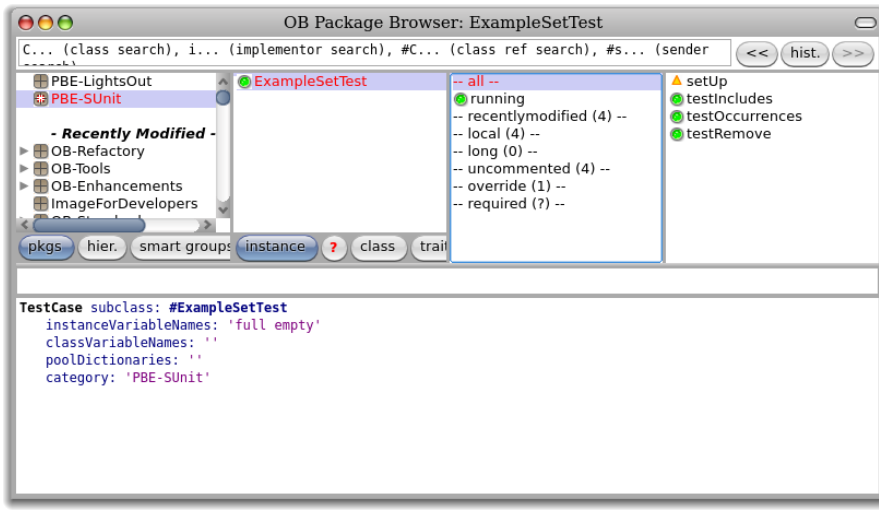


Figure 7.1: Running SUnit tests from the browser

También puedes elegir varios juegos de pruebas a correr, y obtener un log mas detallado de los resultados usando SUnit *Test Runner*, el cual puedes abrir seleccionando World ▷ Test Runner.

El *TestRunner*, mostrado en Figure 7.2, esta diseñado para hacer más fácil la ejecución de grupos de pruebas.

El panel de más a la izquierda lista todas las categorías que contienen las clases de pruebas (*i.e.*, subclases de TestCase); cuando algunas de estas categorías es seleccionada, las clases de pruebas que ellas contienen aparecen en el panel de la derecha. Las clases abstractas se muestran en letra itálica, y la jerarquía de clases de pruebas se muestra con indentación, así las subclases de ClassTestCase estan indentadas más que las subclases de Testcase.

Abre un Test Runner, selecciona la categoría MyTest*, y haz clic en el botón* Run Selected.

Introduce un error en ExampleSetTest»testRemove *y corre las pruebas nuevamente. Por ejemplo, cambia* 5 *por* 4.

Las pruebas que no pasaron (si es que hay alguna) son listadas a las ventanas de la derecha del *Test Runner*; si quieres debugguear una, para ver por qué falló, sólo haz clic sobre el nombre.

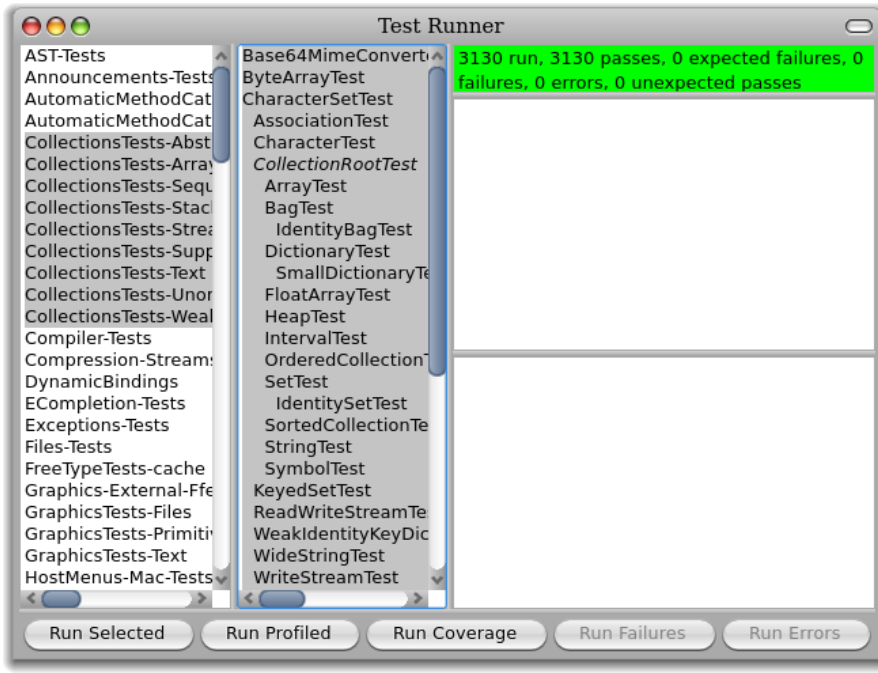


Figure 7.2: The Pharo SUnit Test Runner

Paso 5: interpretar los resultados

El método assert: , el cual está definido en la clase TestCase, espera un argumento de valor booleano, usualmente el valor de una expresion evaluada. Cuando el argumento es verdadero (true), la prueba pasa; cuando el argumento es falso (false), la prueba falla.

En realidad, existen tres posibles retornos de una prueba. La salida que esperamos es que todas aseveraciones en la prueba sean verdaderas, en cuyo caso la prueba pasa. En el test runner, cuando todas las pruebas pasan, la barra de arriba se vuelve verde. Sin embargo, hay dos tipos de cosas que pueden ir mal cuando corres una prueba.

Como es lógico, una de las afirmaciones puede ser falsa, causando que la prueba *falle*.

No obstante, es posible que algun otro tipo de error ocurra durante la ejecución de una prueba, como un error de *message not understood* o un error de *index out of bounds*.

Si un error ocurre, las aserciones en la prueba pueden que no se hayan ejecutado del todo, por lo que no podemos decir que la prueba ha fallado;

no obstante, algo está claramente mal!

El el *test runner*, las pruebas que fallan causan que la barra superior se vuelva amarilla, y son listadas en la ventana del medio sobre la derecha, mientras que las pruebas erroneas causan que la barra se ponga roja, y son listadas en la ventana inferior de la derecha.

Modifica tus pruebas para provocar errores y fallas.

7.5 El libro de recetas de SUnit

Esta sección te dará mas detalles sobre cómo usar SUnit. Si has usado otro framework para hacer pruebas como Junit[1], gran parte de todo esto te resultará familiar, desde ya que todos estos frameworks tienen sus raices en SUnit. Usualmente usarás la GUI de SUnit para correr las pruebas, pero habrá situaciones donde puedes no querer usarla.

Otras aserciones

Además de assert: y deny:, existen varios otros métodos que pueden ser usados para hacer aserciones.

Primero, assert:description: y deny:description: toman un segundo argumento que es un mensaje en una cadena que puede ser usada para describir el motivo de la falla, si es que no es obvia desde la prueba misma.

Estos métodos estan definidos en Section 7.7.

Luego, SUnit provee dos métodos adicionales, should:raise: y shouldnt:raise: para probar la propagación de una excepción.

Por ejemplo, podrías usar (self should: aBlock raise: anException) para probar si una excepción en particular es lanzada durante la ejecución de aBlock. Method 7.6 ilustra el uso de should:raise:.

Intenta correr esta prueba.

Nota que el primer argumento de los métodos should: y shouldnt: es un *bloque* que *contiene* la expresión a ser evaluada.

Method 7.6: *Probando lanzamiento de un error*

```
ExampleSetTest»testIllegal
  self should: [empty at: 5] raise: Error.
  self should: [empty at: 5 put: #zork] raise: Error
```

[1] http://junit.org

SUnit es portable: puede ser usado en todos los dialectos Smalltalk. Para hacer a SUnit portable, sus desarrolladores no tomaron en consideración los aspectos que son dependientes del dialecto. El método de clase TestResult class»error responde a la clase de error del sistema en una forma que es independiente del dialecto.

Puedes aprovechar esto: si quieres escribir pruebas que funcionarán en cualquier dialecto de Smalltalk, en vez de method 7.6 podrías escribir:

Method 7.7: *Portable error handling*

```
ExampleSetTest»testIllegal
  self should: [empty at: 5] raise: TestResult error.
  self should: [empty at: 5 put: #zork] raise: TestResult error
```

 *Inténtalo.*

Corriendo una sola prueba

Normalmente, correrás las pruebas usando el Test Runner.

Si no quieres abrir el test Runner desde el menú open..., puedes ejecutar TestRunner open como un print it.

Puedes correr una sola prueba como sigue.

```
ExampleSetTest run: #testRemove  ⟶  1 run, 1 passed, 0 failed, 0 errors
```

Corriendo todas las pruebas en una clase de pruebas

Cualquier subclase de TestCase responde al mensaje suite, el cual construirá un juego de pruebas que contendrá todos los métodos en la clase cuyos nombres empiezen con la cadena "test".

Para correr las pruebas en el juego, envía el mensaje run.

Por ejemplo:

```
ExampleSetTest suite run  ⟶  5 run, 5 passed, 0 failed, 0 errors
```

¿Debo heredar de TestCase?

En JUnit puedes construir un TestSuite desde cualquier clase arbitraria que contenga métodos test*. En Smalltalk puedes hacer lo mismo pero tendrás que crear un juego a mano y tu clase deberá implementar todos los métodos esenciales de TestCase como por ejemplo, assert:.

Recomendamos que no intentes hacer esto. El framework ya está ahí: úsalo.

7.6 El framework SUnit

SUnit consiste en cuatro clases principales : TestCase, TestSuite, TestResult, y TestResource, como se muestra en Figure 7.3. La idea de un *test resource* fue introducida en SUnit 3.1 para representar a un recurso que es caro de alistar pero que puede ser usado por una serie completa de pruebas. Un TestResource especifica un método setUp que es ejecutado sólo una vez antes del juego de pruebas; esto es para distinguir del método TestCase»setUp, el cual es ejecutado antes de cada prueba.

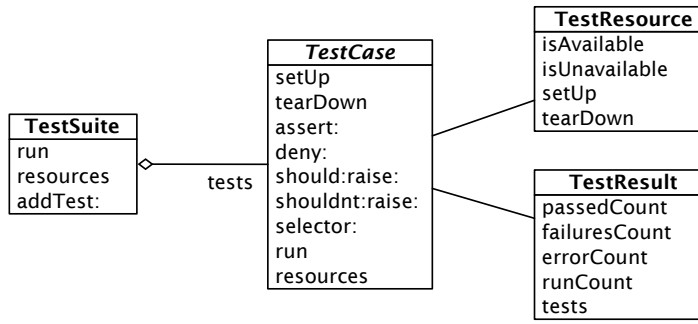


Figure 7.3: Las cuatro clases que representan el corazón de SUnit

TestCase

TestCase es una clase abstracta que esta diseñada para ser heredada por alguna otra subclase; cada una de sus subclases representa un grupo de pruebas que comparten un contexto en común (esto es, un set de pruebas).

Cada prueba se corre creando una nueva instancia de una subclase de TestCase, corriendo setUp, corriendo el método de prueba en sí mismo, y luego corriendo tearDown.

El contexto esta especificado por las variables de instancia de la subclase y por la especialización del método setUp, el cual inicializa esas variables de instancia. Las subclases de TestCase pueden también sobreescribir el método tearDown, el cual es invocado luego de la ejecución de cada prueba, y puede ser usado para liberar cualquier objeto creado durante setUp.

TestSuite

Las instancias de la clase TestSuite contienen una colección de casos de prueba. Una instancia de TestSuite contiene pruebas, y otra sets de pruebas.

Esto es, un set de pruebas contiene sub-instancias de TestCase y TestSuite.

Individualmente, tanto los TestCases como los TestSuites entienden el mismo protocolo, por lo que ambas pueden ser tratadas en la misma forma; por ejemplo, ambas pueden correrse.

Esto es de hecho una aplicación del patrón 'composite' en el cual TestSuite es el compuesto y las TestCases son las hojas — see *Design Patterns* para más información sobre este patrón[2].

TestResult

La clase TestResult representa los resultados de la ejecución de un TestSuite. Graba los números de pruebas pasadas correctamente, el número de pruebas fallidas y el número de errores encontrados.

TestResource

Una de las características mas importantes de un set de pruebas es que deberán ser independientes unas de otras: la falla de una prueba no debería causar una avalancha de fallas en las otras pruebas que dependen de ella, tampoco debería importar el orden en que se corren las pruebas.

Ejecutando setUp antes de cada prueba y tearDown al final ayuda a reforzar esta independencia.

Sin embargo, hay ocasiones en que inicializar el contexto necesario consume demasiado tiempo por lo que no es práctico de hacer antes de la ejecución de cada prueba.

Más aún, si se sabe que los casos de prueba no afectan los recursos de las pruebas, entonces es un desperdicio inicializarlas de nuevo por cada prueba; es suficiente inicializarlas una vez por cada juego de pruebas. Supongamos, por ejemplo, que un set de pruebas necesita consultar a una base de datos, o hacer algún tipo de análisis sobre un código compilado. En esos casos, tendría sentido preparar la base de datos y abrir una conexíon, o compilar algo de código, antes de que cualquier prueba comienze a correr.

Dónde deberíamos preservar estos recursos, así pueden ser compartidos por un set de pruebas? Las variables de instancia de una subclase de TestCase

[2]Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison Wesley, 1995, ISBN 0–201–63361–2–(3).

en particular no nos sirven, dado que una instancia existe sólo lo que dura una prueba.

Una variable global funcionaría, pero usar demasiadas variables globales ensucia el espacio de nombres, y el vínculo entre lo global y las pruebas que dependen de ello no será explícito.

Una mejor solución es poner los recursos necesarios en un objeto singleton de alguna clase. La clase TestResource existe para que ese tipo de clases de recursos hereden de ella.

Cada subclase de TestResource comprende el mensaje actual, al cual responderá una instancia singleton de esa subclase.

Los métodos setUp y tearDown deberían sobreescribirse en la subclase, para tener seguridad de que el recurso es inicializado y finalizado.

Una cosa más nos queda pendiente: de alguna forma, SUnit tiene que ser avisado sobre cuáles recursos estan asociados a determinado juego de pruebas. Un recurso es asociado a una subclcase en particular de TestCase sobreescribiendo el método de *clase resources*.

Por defecto, los recursos de un TestSuite son la unión de los recursos que contienen los TestCases.

Aquí hay un ejemplo. Definamos una subclase de TestResource llamada MyTestResource y la asociamos con MyTestCase especializando el método de clase resources para retornar un vector de las clases de pruebas que vamos a usar.

Class 7.8: *Ejemplo de una subclase de TestResource*

```
TestResource subclass: #MyTestResource
   instanceVariableNames: ''

MyTestCase class»resources
   "associate the resource with this class of test cases"
   ↑{ MyTestResource }
```

7.7 Cacterísticas avanzadas de SUnit

Además de TestResource, la versión actual de SUnit contiene cadenas de descripción de aserciones, soporte para logging, y fallas de pruebas reanudables.

Cadenas de descripción de aserciones

El protocolo de aserción de TestCase incluye un número de métodos que permiten al programador proveer descripción sobre lo que se esta comparando en la aserción. La descripción es un String; si la prueba falla, la cadena se mostrará en el test runner. Por supuesto, esta cadena puede construirse dinámicamente.

```
| e |
e := 42.
self assert: e = 23
   description: 'expected 23, got ', e printString
```

Los métodos relevantes en TestCase son:

```
#assert:description:
#deny:description:
#should:description:
#shouldnt:description:
```

Soporte para logging

las cadenas de descripción que arriba se detallaron también pueden ser logueadas a un Stream como por ejemplo el Transcript, o un archivo de caracteres. Puedes elegir hacer reportes con sólo sobreescribir TestCase»isLogging en tu clase de prueba; también deberás elegir dónde reportar sobreescribiendo TestCase»failureLog para responder a un stream apropiado.

Continuar después de una falla

SUnit también nos permite especificar si una prueba debería continuar luego de una falla, o no. Esta es una poderosa característica que usa los mecanismos de excepciones ofrecidos por Smalltalk. Para ver en qué puede ser usado, echemos un vistazo a un ejemplo. Considera la siguiente expresión:

```
aCollection do: [ :each | self assert: each even]
```

En este caso , tan pronto como la prueba encuentra que primer elemento de la colección no es par, la prueba se detiene. Sin embargo, usualmente nos gustaría continuar, y ver también cuántos elementos hay, y qué elementos hay, no son pares, y tal vez reportar esta información también. Puedes hacer lo siguiente:

```
aCollection do:
   [:each |
   self
```

```
assert: each even
description: each printString , ' is not even'
resumable: true]
```

Esto imprimirá un mensaje en tu stream para logging para cada elemento que falle. No acumula fallas, *i.e.*,, si la aserción falla 10 veces en tu método de prueba, verás solamente una sola falla. Todos los otros métodos de aserción que hemos visto no son reanudables; `assert: p description: s` es equivalente a `assert: p description: s resumable: false`.

7.8 La implementación de SUnit

La implementación de SUnit constituye un interesante caso de estudio de un framework Smalltalk. Echemos un vistazo a algunos aspectos clave de la implemntación siguiendo la ejecución de una prueba.

Corriendo una prueba

para ejecutar una prueba, evaluamos la expresión `(aTestClass selector: aSymbol ) run`.

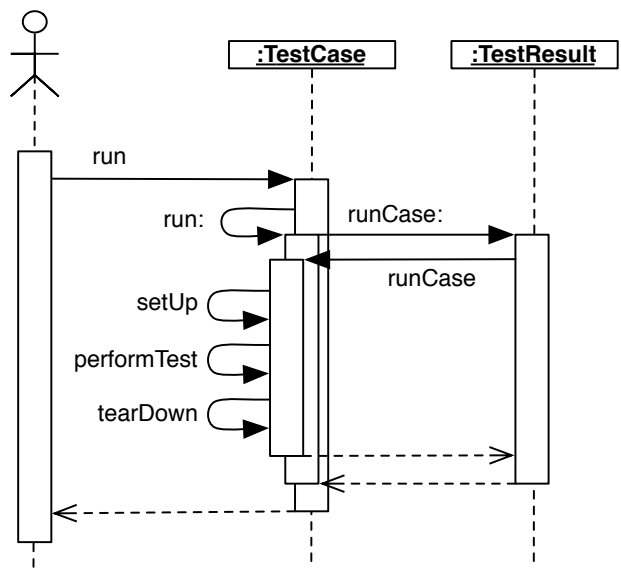


Figure 7.4: Running one test

El método TestCase»run crea una instancia de TestResult que acumulará los resultados de las pruebas, luego se envía a sí mismo el mensaje run:.

(Ver Figure 7.4.)

Method 7.9: *Running a test case*

```
TestCase»run
  | result |
  result := TestResult new.
  self run: result.
  ↑result
```

El método TestCase»run: envía el mensaje runCase: al resultado de la prueba:

Method 7.10: *Passing the test case to the test result*

```
TestCase»run: aResult
  aResult runCase: self
```

El método TestResult»runCase: envía el mensaje runCase a una prueba individual, para ejecutar la prueba.

TestResult»runCase no trata con ninguna excepción que pueda ser lanzada durante la ejecución de la prueba, hace correr unTestCase enviandole el mensaje runCase, y cuenta los errores, fallas y pasadas.

Method 7.11: *Catching test case errors and failures*

```
TestResult»runCase: aTestCase
  | testCasePassed |
  testCasePassed := true.
  [[aTestCase runCase]
        on: self class failure
        do:
           [:signal |
           failures add: aTestCase.
           testCasePassed := false.
           signal return: false]]
              on: self class error
              do:
                 [:signal |
                 errors add: aTestCase.
                 testCasePassed := false.
                 signal return: false].
  testCasePassed ifTrue: [passed add: aTestCase]
```

El método TestCase»runCase envía los mensajes setUp y tearDown como se muestra a continuación.

Method 7.12: *Test case template method*

```
TestCase»runCase
  [self setUp.
  self performTest] ensure: [self tearDown]
```

Corriendo un TestSuite

Para correr mas de una prueba, enviamos el mensaje run a un TestSuite que contenga las pruebas relevantes.

La clase TestCase provee algunas funcionalidades para construir un juego de pruebas desde sus métodos. La expresión MyTestCase buildSuiteFromSelectors retorna un set que contiene todas las pruebas definidas en la clase MyTestCase.

El núcleo de este proceso es

Method 7.13: *Contrucción automática del conjunto de pruebas*

```
TestCase class»testSelectors
  ↑self selectors asSortedCollection asOrderedCollection select: [:each |
    ('test*' match: each) and: [each numArgs isZero]]
```

El método TestSuite»run crea una instancia de TestResult, verifica que todos los recursos estén disponibles, y luego se envía a sí mismo el mensaje run:, el cual corre todas las pruebas en el set. Todos los recursos son liberados luego.

Method 7.14: *Ejecutando el conjunto de pruebas*

```
TestSuite»run
  | result |
  result := TestResult new.
  self resources do: [ :res |
    res isAvailable ifFalse: [↑res signalInitializationError]].
  [self run: result] ensure: [self resources do: [:each | each reset]].
  ↑result
```

Method 7.15: *Pasaje del resultado de la prueba al conjunto de pruebas*

```
TestSuite»run: aResult
  self tests do: [:each |
    self changed: each.
    each run: aResult].
```

La clase TestResource y sus subclases mantienen un registro de sus instancias actualmente creadas (uno por clase) que puede ser accedido y creado usando el método de clase current. Esta instancia se limpia cuando laas pruebas han terminado de correr y los recursos son restaurados.

La comprobación de disponibilidad de recursos hace posible que el recurso sea creado de vuelta si es necesario, como se muestra en el método de clase TestResource class»isAvailable. Durante la creación de la instancia de TestResource, es inicializado y el método setUp es invocado.

Method 7.16: *Disponibilidad del recurso de pruebas*

```
TestResource class»isAvailable
  ↑self current notNil and: [self current isAvailable]
```

Method 7.17: *Creación del recurso de prueba*

```
TestResource class»current
  current isNil ifTrue: [current := self new].
  ↑current
```

Method 7.18: *inicialización de Test resource*

```
TestResource»initialize
  super initialize.
  self setUp
```

7.9 Algunos consejos sobre testing

Mientras la mecánica de hacer pruebas es fácil, escribir buenas pruebas no lo es. Aquí hay algunos consejos sobre cómo diseñar pruebas.

Reglas de Feathers para pruebas unitarias. Michael Feathers, un consultor de procesos ágiles y autor, escribe:[3]

> *Una prueba no es una prueba unitaria si:*
>
> - *se comunica con la base de datos,*
> - *se comunica a través de la red,*
> - *manipula el sistema de archivos,*
> - *no puede correr al mismo tiempo que otra de tus pruebas unitarias, o*
> - *tienes que hacerle cosas especiales a tu entorno (como por ejemplo editar archivos de configuración) para correrla.*
>
> *Las pruebas que hacen estas cosas son malas. Usualmente son valiosas de escribir, y pueden ser escritas en una unit test harness. Sin embargo, es importante ser capaz de separarlas de las verdaderas pruebas unitarias para poder guardar un conjunto de pruebas que puedan correrse rápidamente en cualquier momento que hagamos nuestros cambios.*

[3] Ver http://www.artima.com/weblogs/viewpost.jsp?thread=126923. 9 Septiembre 2005

Nunca llegues a una situación donde no quieres correr tus pruebas unitarias porque lleva demasiado tiempo.

Pruebas Unitarias *vs*. Pruebas de Aceptación. Las pruebas unitarias capturan una parte de la funcionalidad, y por tal motivo hacen fácil identificar los bugs en esa funcionalidad.

Siempre que sea posible, trata de tener pruebas unitarias por cada método que puede llegar a fallar, y agrúpalas por clase. No obstante, para ciertas situaciones, como inicializaciones sumamente recursivas o complicadas, es más fácil escribir pruebas que representen un escenario en toda la aplicación; estos son llamados pruebas de aceptación o pruebas funcionales.

Las pruebas que rompen las reglas de Feathers podrían ser buenas pruebas de aceptación.

Agrupa las pruebas de aceptación de acuerdo a la funcionalidad que ellas ensayan. Por ejemplo, si estás escribiendo un compilador, podrías escribir pruebas de aceptación que haga aserciones sobre el código generado para cada posible sentencia del lenguaje fuente.

Tales pruebas podrían hacer uso de muchas clases, y podría llevar un largo tiempo correrlas porque manipulan el sistema de archivos.

Puedes escribirlas usando SUnit, pero no querrás correrlas cada vez que hagas un cambio pequeño, así que deberíamos separarlas de las verdaderas pruebas unitarias.

La regla de Black para el Testing. Para cada prueba en el sistema, tienes que ser capaz de identificar alguna propiedad a la cual tu prueba incrementa tu confianza.

Es obvio que habrán propiedades no importantes que no estas comprobando. Esta regla plantea el hecho menos obvio de que no debería haber pruebas que no agreguen valor al sistema incrementando tu confianza en alguna propiedad del mismo.

Por ejemplo, muchas pruebas del mismo atributo no hacen bien. De hecho, dañan en dos formas.

Primero, hacen más dificil inferir el comportamiento de la clase leyendo solamente la prueba. Segundo, como un bug en el código podría romper muchas pruebas, hace más dificil estimar cuántos bugs aún restan en el código.

Entonces, ten un atributo en mente cuando escribes una prueba.

7.10 Resumen del capítulo

En este capítulo se ha explicado por qué las pruebas son una importante inversión en el futuro de tu código.

Explicamos en un modo paso a paso cómo definimos unas cuantas pruebas para la clase Set. Luego hicimos un recorrido por el núcleo del framework SUnit presentando las clases TestCase, TestResult, TestSuite y TestResources. Por último, profundizamos en SUnit siguiendo la ejecución de una prueba y de un conjunto de pruebas.

- Para maximixar su potencia, las pruebas unitarias deberían ser rápidas, repetibles, independientes de cualquier interacción humana y cubrir una única funcionalidad.
- Las pruebas para una clase llamada MyClass pertencen a una clase clasificada como MyClassTest, la cual debería ser presentada como una subclase de TestCase.
- Inicializar tus datos de prueba en un método setUp.
- Cada método de prueba debería empezar con la palabra "test".
- Usa los métodos de TestCase : assert:, deny: y otros para hacer aserciones.
- Corre las pruebas usando la herramienta test runner de SUnit (en la barra de herramientas).

Chapter 8

Clases Básicas

La mayoría de la magia de Smalltalk no está tanto en el lenguaje como en las librerías de clases. Para programar efectivamente con Smalltalk, necesita aprender cómo la librería de clases soporta el lenguaje y el entorno. Toda la libería de clases a sido enteramente escrita en Smalltalk y puede ser facilmente ampliada desde un paquete que agregue alguna nueva funcionalidad a una clase si esta no se encuentra definida en ella.

Nuestro objetivo no es presentar en un tedioso detalle toda la librería de clases de Pharo, pero sí resaltar las clases clave y los métodos que necesita conocer para usar o evitar al programar efectivamente. En este capítulo cubriremos las clases básicas que necesitará en cualquier aplicación: Object, Number and its subclasses, Character, String, Symbol and Boolean.

8.1 Object

Para todos los usos y propósitos, Object es la raíz de toda la jerarquía. Actualmente, en Pharo la verdadera raíz de la jerarquía es ProtoObject, la cual es usada para definir entidades mínimas que son tomadas como objetos, pero esto es un tema que podremos ignorar por el momento.

Objectpuede ser encontrado en la categoría *Kernel-Objects*. Sorprendentemente, aquí hay más de 400 métodos (incluyendo extensiones). en otras palabras, cada clase que defina automáticamente proveerá esos 400 métodos, independientemente lo sepa o no. Note que algunos de los métodos deberán ser eliminados y las nuevas versiones de Pharo pueden remover algunos de los ás superfluos.

El comentario de la clase Object dice:

> Object *es la clase raíz para casi todas las otras clases de la jerarquía de lcases. Las excepciones son* ProtoObject *(la superclase de* Object*) y sus subclases. La clase* Object *provee un comportamiento común por defecto para todos los objetos normales, tal como acceso, copia, comparaciíon, manejo de errores, envío de errores y reflexíon. También los mensajes útiles que todos los objetos deberían responder están definidos aquí.* Object *no tiene variables de instancia y no deberían agregársele. Esto es gracias a varias clases de objetos que heredan de* Object *y tienen implementaciones especiales (*SmallInteger *y* UndefinedObject *por ejemplo) o por la VM que conoce y depede de la estructura y diseño de ciertas clases estandar.*

Si comenzamos a navegar en las categorías de los métodos del lado de la instancia de Object comenzaremos a ver algunos de los comportamientos claves que provee.

Imprimiendo

Cada objeto en Smalltalk puede retornar una forma impresa de si mismo. Puede elegir cualquier expresión en un workspace y seleccionar la opción print it del menú: esto ejecuta la expresión y le pide al objeto retornado que se imprima. En realidad, se envía el mensaje printString al objeto devuelto. El método printString, el cual es un método plantilla [template method], envía de su núcleo el mensaje printOn: a su receptor. El mensaje printOn: es un gancho que puede ser modelado.

Object»printOn: is por mucho el método que más frecuentemente se sobreescribirá. Este método toma como argumento un Stream en el cual una representación String del objeto será escrito. La implementación por defecto simplemente escribre el nombre de la clase predecida por "a" o "an". Object» printString devuelve el String que está escrito.

Por ejemplo, la clase Browser no redefine el método printOn: y envía el mensaje printString a una instanacia que ejecuta el método definido en Object.

```
Browser new printString  ⟶  'a Browser'
```

La clase Color muestra un ejemplo de modelado de printOn:. Imprime el nombre de la clase seguida del método usado para generar ese color.

Method 8.1: *printOn: redefinición.*

```
Color»printOn: aStream
   | name |
   (name := self name) ifNotNil:
      [ ↑ aStream
         nextPutAll: 'Color ';
         nextPutAll: name ].
   self storeOn: aStream
```

```
Color red printString  ⟶  'Color red'
```

Note que el mensaje printOn: no es igual a storeOn:. El mensaje storeOn: pone en sus argumentos una expresión que puede ser usada para recrear el receptor. Esta expersión es evaluada cuando el parámetro es leído usando el mensaje readFrom:. printOn: solo retorna una versión textual del receptor. Por supuesto, puede ocurrir que esta representación textual puedea representar al receptor como una epxresión autoevaluada.

Unas palabras sobe representación y representación autoevaluada. En programación funcional, las expresiones retornar valores cuando son ejecutadas.En Smalltalk, los mensajes (expresiones) retornan objetos (valores). Algunos objetos tienen la simptática propiedad de que su valor son ellos mismos. Por ejemplo, el valor del objeto true es sí mismo, el objeto true. Podemos llamar a tales objetos, *objetos autoevaluables*. Puede ver una versión impresa del objeto valor cuando imprima el objeto en un workspace. Aquí hay algunos ejemplos de expresiones autoevaluadas.

```
true        ⟶  true
3@4         ⟶  3@4
$a          ⟶  $a
#(1 2 3)    ⟶  #(1 2 3)
Color red   ⟶  Color red
```

Notese que altunos objetos como los arrays son autoevaluados o no dependientes de los objetos que contengan. Por ejemplo, un array de booleanos es autoevaluado mientras que un array personas no. El siguiente ejemplo muestra que un dinámico es autoevaluable solo si los objetos contenidos lo son:

```
{10@10. 100@100}          ⟶  {10@10. 100@100}
{Browser new . 100@100}   ⟶  an Array(a Browser 100@100)
```

Recuerde que los literales pueden contener solamente literales. Por lo tanto, el siguiente array no coniente dos puntos, sino más bien seis elementos.

```
#(10@10 100@100)  ⟶  #(10 #@ 10 100 #@ 100)
```

Muchas de las especializaciones de printOn: implementan comportamientos autoevaluables. Las implementaciones de Point»printOn: y Interval»printOn: son autoevaluables.

Method 8.2: *Autoevaluación de* Point

```
Point»printOn: aStream
   "The receiver prints on aStream in terms of infix notation."
   x printOn: aStream.
   aStream nextPut: $@.
   y printOn: aStream
```

Method 8.3: *Autoevaluación de* Interval

```
Interval»printOn: aStream
   aStream nextPut: $(;
      print: start;
      nextPutAll: ' to: ';
      print: stop.
   step ~= 1 ifTrue: [aStream nextPutAll: ' by: '; print: step].
   aStream nextPut: $)
```

```
1 to: 10  ⟶  (1 to: 10)  "intervals are self−evaluating"
```

Identidad e Igualdad

En Smalltalk, el mensaje = comprueba la *igualdad* de los objetos (por ejemplo cuando dos objetos representan el mismo valor) Por otro lado, == comprueba la *identidad* del objeto (por ejemplo, cuando dos expresiones representan al mismo objeto).

La implementación por defecto de la igualdad de un objeto es para comprobar la identidad de un objeto:

Method 8.4: *Igualdad de objetos*

```
Object»= anObject
   "Answer whether the receiver and the argument represent the same object.
   If = is redefined in any subclass, consider also redefining the message hash."
   ↑ self == anObject
```

Este es un método que frecuentemente quiera sobreescribir. Considere el caso de os números complejos:

```
(1 + 2 i) = (1 + 2 i)   ⟶  true   "same value"
(1 + 2 i) == (1 + 2 i)  ⟶  false  "but different objects"
```

Esto funciona porque Complex sobreescribe = como se muestra a continuación:

Method 8.5: *Igualdad de números complejos*

```
Complex»= anObject
   anObject isComplex
      ifTrue: [↑ (real = anObject real) & (imaginary = anObject imaginary)]
      ifFalse: [↑ anObject adaptToComplex: self andSend: #=]
```

La implementación por defecto de Object»~= simplemente niega Object»=, y no debería normalmente ser cambiado.

```
(1 + 2 i) ~= (1 + 4 i)    ⟶    true
```

Si sobreescribe =, debería considerar sobreescriribr hash. Si las instancias de su clase son usadas como claves en un Dictionary, entonces debería asegurarse que las instancias consideradas a ser iguales tengan el mismo valor hash:

Method 8.6: *Hash debe ser reimplementado para números complejos*

```
Complex»hash
   "Hash is reimplemented because = is implemented."
   ↑ real hash bitXor: imaginary hash.
```

De la misma forma que debería sobreescribir = y hash conjuntamente, *nunca* debería sobreescribir == (la semántica de la identidad de los objetos es la misma para todas las clases). == es un método primitivo de ProtoObject.

Note que Pharo tiene algunos comportamientos extraños comparado con otros Smalltalks. Por ejemplo, un símbolo y una cadena pueden ser iguales (consideramos esto como un error y no como una ventaja)

```
#'lulu' = 'lulu'    ⟶    true
'lulu' = #'lulu'    ⟶    true
```

Membresía de clases

Muchos métodos permiten consultar la clase de un objeto.

class. Puede preguntar a cualquier objeto sobre su clase usando el mensaje class.

```
1 class    ⟶    SmallInteger
```

Por otro lado, puede preguntar si un objeto es una instancia de una clase específica:

```
1 isMemberOf: SmallInteger  ⟶  true   "must be precisely this class"
1 isMemberOf: Integer       ⟶  false
1 isMemberOf: Number        ⟶  false
1 isMemberOf: Object        ⟶  false
```

Dado que Smalltalk está escrito en sí mismo, puede navegar realmente a través de su estructura usando la correcta combinación de mensajes de clases y superclases (vea Chapter 13).

isKindOf: Object»isKindOf: Responde cuando la clase del receptor es de la misma clase o es una sublcase de la clase del agumento.

```
1 isKindOf: SmallInteger  ⟶  true
1 isKindOf: Integer       ⟶  true
1 isKindOf: Number        ⟶  true
1 isKindOf: Object        ⟶  true
1 isKindOf: String        ⟶  false

1/3 isKindOf: Number      ⟶  true
1/3 isKindOf: Integer     ⟶  false
```

1/3 el cual es una Fraction como un tipo de Number, desde que la clase Number es una superclase de la clase Fraction, pero 1/3 no es Integer.

respondsTo: Object»respondsTo: Responde cuando el receptor entiende el selector de mensaje pasado como argumento.

```
1 respondsTo: #,  ⟶  false
```

Normalmente es una mala idea preguntar a un objeto por su clase, o preguntar cuales mensajes entiende. En lugar de timar decisiones basadas en la clase de un objeto, simplemente debería enviar un mensaje al objeto y que él decida (por ejemplo, en lo más profundo de su clase) cómo debería reaccionar.

Copiando

La copia de objetos introduce algunos temas adicionales. Desde que las variables de instancia son accedidas por referencia, una *copia superficial [shallow copy]* de un objeto debería compartir sus referencias a las variables de instancia del objeto original:

```
a1 := { { 'harry' } }.
a1  ⟶  #(#('harry'))
a2 := a1 shallowCopy.
```

```
a2    ⟶    #(#('harry'))
(a1 at: 1) at: 1 put: 'sally'.
a1    ⟶    #(#('sally'))
a2    ⟶    #(#('sally'))    "the subarray is shared"
```

Object»shallowCopy es un método primitivo que crea una copia superficial de un objeto. Desde que a2 es solo una copia superficial de a1, los dos array comparten la referencia al array anidado que contienen.

Object»shallowCopy es la "interfa pública" a Object»copy y debería ser sobreescrito si las instancias son únicas. Este es el caso, por ejemplo con las clases Boolean, Character, SmallInteger, Symbol y UndefinedObject.

Object»copyTwoLevel hace lo obvio cuando una simple copia superficial no es suficiente:

```
a1 := { { 'harry' } } .
a2 := a1 copyTwoLevel.
(a1 at: 1) at: 1 put: 'sally'.
a1    ⟶    #(#('sally'))
a2    ⟶    #(#('harry'))    "fully independent state"
```

Object»deepCopy hace una copia profuda en forma arbitraria de un objeto.

```
a1 := { { { 'harry' } } } .
a2 := a1 deepCopy.
(a1 at: 1) at: 1 put: 'sally'.
a1    ⟶    #(#('sally'))
a2    ⟶    #(#(#('harry')))
```

El problema con deepCopy es que no terminará cuando se aplique a una esctructura mutuamente recursiva:

```
a1 := { 'harry' }.
a2 := { a1 }.
a1 at: 1 put: a2.
a1 deepCopy    ⟶    ... does not terminate!
```

Dado que es posible sobreescribir deepCopy para hacer las cosas correctamente, Object»copy brinda una mejor solución:

Method 8.7: *Copiando objetos como un método plantilla*

```
Object»copy
    "Answer another instance just like the receiver.
    Subclasses typically override postCopy;
    they typically do not override shallowCopy."
    ↑self shallowCopy postCopy
```

Debería sobreescribir postCopy para copiar cualqueir variable de instancia que no debiera ser compartida. postCopy debería hacer siempre un super postCopy.

Debugging

El método más importante aquí es halt. Al establecer un punto de ruptura en un método, simplemente inserte el envío del mensaje self halt en algún punto en el cuepo del método. Cuando este mensaje es enviado, la ejecución será interrumpida y el debugger se abrirá en este punto de su programa. (Vea Chapter 6 por más detalles del debbuger)

El siguiente mensaje más importante es assert:, el cual toma un bloque como su argumento. Si el bloque devuelve true, la ejecución continúa. De otra forma, una escepción AssertionFailure será disparada. Si esta no es capturada por otro, el debugger se abrirá en este punto de la ejecución. assert: es especialmente útil para soportar el *diseño por contrato*.El uso típico es para chequar precondiciones no triviales a los métodos públicos de los objetos. Stack»pop puede facilmente haber sido implementado así:

Method 8.8: *Chequeando una precondición*

```
Stack»pop
   "Return the first element and remove it from the stack."
   self assert: [ self isEmpty not ].
   ↑self linkedList removeFirst element
```

No confunda Object»assert: con TestCase»assert:, el cual ocurre en el framewort de pruebas Sunit (vea Chapter 7). Mientras que el primero espera un bloque como argumento[1], el segundo espera un Boolean. Si bien ambos son útiles para debuguear, cada uno tiene una muy distinta intención.

Manejo de errores

Este protocolo contiene muchos métodos útiles a la hora de señalar errores en tiempo de ejecución.

Enviar self deprecated: *anExplanationString* señala que el método actual no debería segur usándose si la desoeciación ha sido encendida en el protocolo *debug* del navegador. El argumento String debería ofrecer una alternativa.

```
1 doIfNotNil: [ :arg | arg printString, ' is not nil' ]
      ⟶    SmallInteger(Object)>>doIfNotNil: has been deprecated. use ifNotNilDo:
```

[1] Actualmente, esto podría tomar cualquier argumento que entienda value, incluyendo a Boolean.

doesNotUnderstand: es enmviado cuando la búsqueda de mensajes falla. La implementación por defecto, por ej Object»doesNotUnderstand: disparará al debugger en este punto. Puede ser útil sobreescribir doesNotUnderstand: para proporcionar algún otro comportamiento.

Object»error y Object»error: son métodos genéricos que pueden ser utilizados para disparar excepciones (Generalmente es mejor disparar sus propias y diseñadas excepciones, de esta forma puede distinguir errores derivados de su código de aquellos que provienen de las clases del kernel)

Los métodos abstractos en Smalltalk son omplementados por convención con el cuerpoself subclassResponsibility. Una clase abstractra debería ser instanciada por accidente, originando la llamada a métodos abstractos que resultarían en una evaluación de Object»subclassResponsibility.

Method 8.9: *Señalando un método como abastracto*

```
Object»subclassResponsibility
   "This message sets up a framework for the behavior of the class' subclasses.
   Announce that the subclass should have implemented this message."
   self error: 'My subclass should have overridden ', thisContext sender selector
      printString
```

Magnitude, Number y Boolean sib ejemplos clásicos de clases abstractas que veremos brevemente en este capítulo.

```
Number new + 1    ⟶    Error: My subclass should have overridden #+
```

self shouldNotImplement es enviado por convención para marcar un método heredado que no es aporpiado pra su subclase. Esto generalmente marca que algo no va bien con el diseño de la jeraquía de esa clase. Debido a la limitaciones de la herencia simple, a veces es muy difícil evitar tales soluciones.

Un ejemplo típico es Collection»remove: el cual es heredado por Dictionary pero señalado como no implementado (Un Dictionary provee en su lugar removeKey:).

Probando

los métodos de *prueba* no tienen nada que ver con las pruebas de SUnit! Un método de prueba es aquyel que le permite realizar una pregunta sobre el estado del receptor y devolver un Boolean.

Numerosos métodos de prueba son provistos por Object. Ya hemos visto isComplex. Otros son isArray, isBoolean, isBlock, isCollection y as´í. Generalmente tales métodos deben ser evitados ya que conturar a un objeto por su clase es una forma de violar el encapsulamiento. En lugar de probar a un objeto por su clase, debería simplemente enviar un requerimiento y dejar que el objeto decida cómo manejarlo.

Sin embargo, algunos de estos étodos de prueba son sin lugar a dudas útiles. Los más utiles son probablemente ProtoObject»isNil y Object»notNil (pensar en el patrón de diseño Null Object[2] puede obviar unclusive la necesidad de usar estos métodos).

Lanzar la inicialización

Un método final pero clave que no ocurre en Object pero si en ProtoObject es initialize.

Method 8.10: initialize *es un método gancho vacío*

```
ProtoObject»initialize
    "Subclasses should redefine this method to perform initializations on instance creation"
```

La razón de su importancia es que en Pharo, el étodo por defecto new definido por cada nueva clase en el sistema enviará initialize a cada instancia nueva creada.

Method 8.11: new *como un método de clase*

```
Behavior»new
    "Answer a new initialized instance of the receiver (which is a class) with no indexable
    variables. Fail if the class is indexable."
    ↑ self basicNew initialize
```

Esto significa que simplemente por sobreescribir el método initialize, las nuevas instancias de su clase serán inicializazadas automáticamente. El mmétodo initialize debería normalmente realizar un a super initialize para establecer las invariantes para cualquier viariable de clase heredada. (Notese que esto *no* es el comportamiento estandar de otros Smalltalks)

8.2 Numbers

Destacadamente, los números en Smalltalk no son datos primivitos sino objetos reales. Por supuesto, los números son implementados eficientemente en la máquina virtual, pero la jerarquía de Number es tan perfectamente accesible y extensble como cualquier otra jerarquía de clases en Smalltalk.

Los números pueden ser encontrados en la categoría *Kernel-Numbers*. La raíz abstracta de su jerarquía es Magnitude, la cual representa todos los tipos de clases que soportan los operadores de comparación. Number agrega varios

[2]Bobby Woolf, Null Object. In Robert Martin, Dirk Riehle and Frank Buschmann, editors, Pattern Languages of Program Design 3. Addison Wesley, 1998.

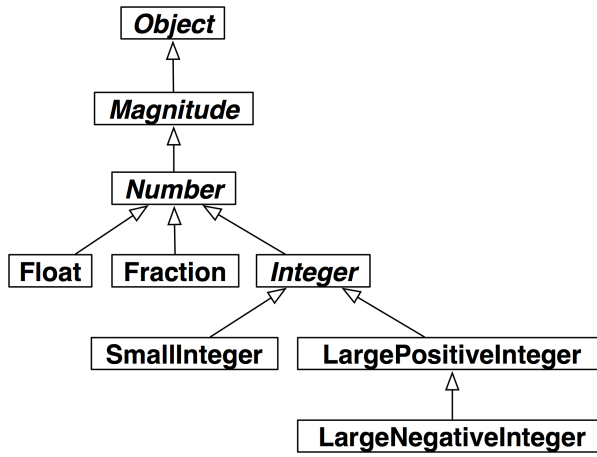


Figure 8.1: The Number Hierarchy

operadores aritméticos y otros como métodos abstractos. Float y Fraction representan, respectivamente, números en punto flotante y valores fraccionarios. Integer es tambiénabstractro, debiendo distinguirse entre las subclases SmallInteger, LargePositiveInteger y LargeNegativeInteger. La mayoría de los usuarios no necesitan estar alertas entre las tres clases Integer ya que los valores son automáticamente convertidos según sea necesario.

Magnitude

Magnitude no solo es el ancestro de la clase Number, sino tambienén de otras clases que soportan operaciones de comparación, tales como Character, Duration y Timespan. (Los números Complex no son comparables, por lo tanto no heredan de Number)

Los métodos < and = son abstractos. el resto de las operaciones están generalmente definidas:

Method 8.12: *Métodos abstractos de comparación*

```
Magnitude» < aMagnitude
   "Answer whether the receiver is less than the argument."
   ↑self subclassResponsibility

Magnitude» > aMagnitude
   "Answer whether the receiver is greater than the argument."
   ↑aMagnitude < self
```

Number

Similarmente, Number define +, −, * y / para ser abstractos, pero todo el resto de los operadores aritméticos están definidos.

Todos los objetos Number soportan varios operadores de *conversión*, tales como asFloat y asInteger. Estos también tienen numerosos *métodos constructores atajo*, tales como i, el cual convierte un Number a una instancia de Complex con un componente real cero y otros, los cuales generan Durations, tales como hour, day y week.

Numbers soporta directamente las funciones matemáticas comunes, como sin, log, raiseTo:, squared, sqrt y similares.

Number»printOn: es implementado en términos de método abstracto Number »printOn:base:. (La base por defecto es 10)

los métodos de prueba incluyen even, odd, positive y negative.Como era de esperar Number sobreescribe isNumber. Más interesante es que isInfinite está definido para devolver false.

Truncation métodos que incluyen floor, ceiling, integerPart, fractionPart y similares.

```
1 + 2.5      ⟶   3.5       "Addition of two numbers"
3.4 * 5      ⟶   17.0      "Multiplication of two numbers"
8 / 2        ⟶   4         "Division of two numbers"
10 − 8.3     ⟶   1.7       "Subtraction of two numbers"
12 = 11      ⟶   false     "Equality between two numbers"
12 ~= 11     ⟶   true      "Test if two numbers are different"
12 > 9       ⟶   true      "Greater than"
12 >= 10     ⟶   true      "Greater or equal  than"
12 < 10      ⟶   false     "Smaller than"
100@10       ⟶   100@10    "Point creation"
```

El siguiente código funciona esperablemente bien en Smalltalk:

```
1000 factorial / 999 factorial    ⟶   1000
```

Note que el factorial de 1000 es realmente calculado. Lo cual en otros lenguajes puede ser algo muy difícil de computar. Esto es un excelente ejemplo de coerción automática y un exacto manejo de un número.

 *Trate de mostrar el resultado del* factorial de 1000. *Toma más tiempo mostrarlo que calcularlo!*

Float

Float implementa los métodos abstractos de Number para los números de punto flotante.

Más interesante es saber que la clase Float (del lado de la clase) provee métodos para devolver las siguientes *constantes*: e, infinity, nan y pi.

```
Float pi                    ⟶  3.141592653589793
Float infinity              ⟶  Infinity
Float infinity isInfinite   ⟶  true
```

Fraction

Las Fracciones son representadas por variables de instancia para el numerador y denominador, las cuales deben serInteger. Fractions es normalmente creado por la división de Integer (en lugar de usar el método constructor de Fraction»numerator:denominator:):

```
6/8             ⟶  (3/4)
(6/8) class     ⟶  Fraction
```

Multiplicando una Fraction por un Integer o por otra Fraction puede resultar en un Integer:

```
6/8 * 4    ⟶  3
```

Integer

Integer es el ancestro abstracto de tres implementaciones concretas de enteros. En suma para proveer implementaciones concretas de mmuchos métodoss abstractos. Number, también suma unos pocos métodosespecíficos para enteros como factorial, atRandom, isPrime, gcd: y muchos otros.

SmallInteger es especial en tanto que sus instancias son representadas en forma compacta. En lugar de ser almacenadas por referencia, un SmallInteger es representado directamente usando los bits que serían usados de otra forma para señalar la referencia. El primer bit de una referencia de objeto marca si el objeto es un SmallInteger o no.

los métodos de clase minVal y maxVal nos ciden el rango de un SmallInteger:

```
SmallInteger maxVal = ((2 raisedTo: 30) − 1)      ⟶  true
SmallInteger minVal = (2 raisedTo: 30) negated    ⟶  true
```

Cuando un SmallInteger se va fuera de rango, es automáticamente convertido en un LargePositiveInteger o en un LargeNegativeInteger, según sea necesario:

```
(SmallInteger maxVal + 1) class    ⟶  LargePositiveInteger
(SmallInteger minVal − 1) class    ⟶  LargeNegativeInteger
```

Los enteros largos son similarmente convertidos a enteros cortos cuando es apropiado.

En la mayoría de los lenguajes de programación, los enteros pueden ser útiles para especificar comportamientos iterativos. Aquí tenemos el método timesRepeat: para evaluar un bloque repetidamente. Vimos un ejemplo parecido en Chapter 3:

```
n := 2.
3 timesRepeat: [ n := n*n ].
n    ⟶    256
```

8.3 Characters

Character es definido en la categoría *Collections-Strings* como una subclase de Magnitude. Los caracteres imprimibles son representados en Pharo como $⟨*char*⟩. Por ejemplo:

```
$a < $b    ⟶    true
```

los caracteres no imprimibles son generados por varios métodos de clase. Character class»value: toma el valor entero Unicode (o ASCII) como argumento y devuelve el correspondiente caracter. El protocolo *accessing untypeable characters* contiende un conveniente número de métodos constructores tales como backspace, cr, escape, euro, space, tab, y similares.

```
Character space = (Character value: Character space asciiValue)    ⟶    true
```

El método printOn: es suficientemente inteligente para saber cual de las tres formas de generar caracteres es la que ofrece la representación más apropiada:

```
Character value: 1     ⟶    Character home
Character value: 2     ⟶    Character value: 2
Character value: 32    ⟶    Character space
Character value: 97    ⟶    $a
```

Varios métodos de prueba son convenientemente construidos aquí: isAlphaNumeric, isCharacter, isDigit, isLowercase, isVowel, y similares.

Para convertir un Character en una cadena que contenga solo ese caracter, envíe asString. En este caso, asString y printString devuelven resultados distintos:

```
$a asString      ⟶    'a'
$a               ⟶    $a
$a printString   ⟶    '$a'
```

Cada Character ascii es una única instancia, almacenada en la variable de clase CharacterTable:

```
(Character value: 97) == $a    ⟶    true
```

Characters fuera del rango 0..255 no son únicos. Si embargo:

```
Character characterTable size                    ⟶    256
(Character value: 500) == (Character value: 500)    ⟶    false
```

8.4 Strings

La clase String esta definida también en la categoría *Collections-Strings*. Un String es una Collection indexada que almacena solamente Characters.

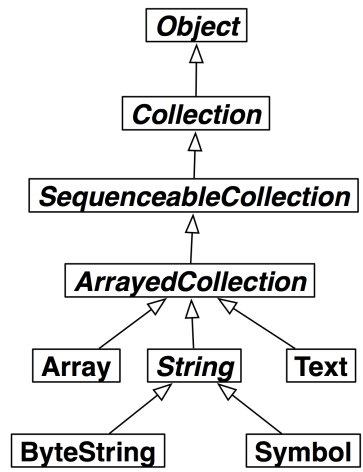


Figure 8.2: The String Hierarchy

En efecto, String es abstracta y las cadenas en Pharo son actualmente instancias de la clase concreta ByteString.

```
'hello world' class    ⟶    ByteString
```

La otra subclase importante de String es Symbol. La principal diferencia es que esta es solo una simple instancia de Symbol con un valor dado (Es algo a veces llamdado "la propiedad de única instancia"). En contraste, dos Strings sontruidos separadamente que contengan la misma secuencia de caracteres serán objetos diferentes.

```
'hel','lo' == 'hello'    ⟶    false
```

```
('hel','lo') asSymbol == #hello  ⟶  true
```

Otra diferencia importante es que un String es mutable, mientras que un Symbol es inmutable.

```
'hello' at: 2 put: $u; yourself  ⟶  'hullo'
```

```
#hello at: 2 put: $u  ⟶  error
```

Es facil de olvidar qur como las cadenas son colecciones, entoces entiende los mismos mensajes de as otras colecciones:

```
#hello indexOf: $o  ⟶  5
```

Como String no hereda de Magnitude, este debe soportar los métodos usuales de *comparación*, <, = y similares. En suma, String»match: es util para algunos apareos básicos:

```
'*or*' match: 'zorro'  ⟶  true
```

Si necesita un más avanzado soporte para expresiones regulares, revise el paquete *Regex* realizado por Vassili Bykov.

Las cadenas soportan un gran número de métodos de conversi'on. Muchos de ellos son métodos atajo constructores para otras clases, tales como asDate, asFileName y similares. También hay un número de métodos útiles para convertir una cadena en otra cadena, tales como capitalized y translateToLowercase.

Para más cadenas y colecciones, vea Chapter 9.

8.5 Booleans

La clase Boolean ofrece una fascinante inmersión en cómo el lenguaje Smalltalk ha sido utilizado dentro de su librería de clases. Boolean es la superclase abstracta con Singleton de las clases True y False.

La mayoría de los comportamientos de Booleans pueden ser entendidos al considerar el método ifTrue:ifFalse:, el cual toma dos Blocks como argumentos.

```
(4 factorial > 20) ifTrue: [ 'bigger' ] ifFalse: [ 'smaller' ]  ⟶  'bigger'
```

El método es abstracto en Boolean. La implementación está en sus subclases, siendo concreta y trivial:

Method 8.13: *Implementations of* ifTrue:ifFalse:

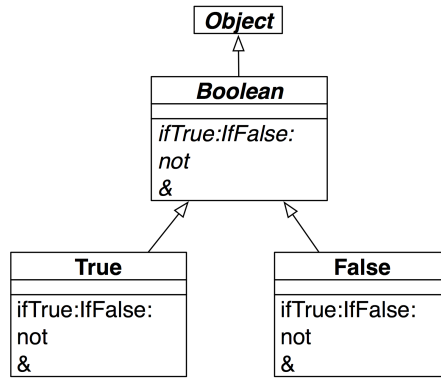


Figure 8.3: The Boolean Hierarchy

```
True»ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ↑trueAlternativeBlock value

False»ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ↑falseAlternativeBlock value
```

En efecto, esta es la escencia de OOP: Cuando un mensaje es enviado a un objeto, el objeto por sí mismo determina cuál objeto será usado para responder. En este caso, una instancia de True simplemente evalúa la alternativa *verdadera* , mientras que una instancia de False evalúa la alternativa *falsa*. Todos los métodos abstractos de Boolean son implementados de esta manera por True y False. Por ejemplo:

Method 8.14: *Implementando la negación*

```
True»not
  "Negation--answer false since the receiver is true."
  ↑false
```

Booleans ofrecen varios métodos útiles como ifTrue:, ifFalse:, ifFalse:ifTrue. Usted puede elegir entre conjunciones y disjunciones ansiosas o perezosas

```
(1>2) & (3<4)                  ⟶  false   "must evaluate both sides"
(1>2) and: [ 3<4 ]             ⟶  false   "only evaluate receiver"
(1>2) and: [ (1/0) > 0 ]       ⟶  false   "argument block is never evaluated, so no
      exception"
```

En el primer ejemplo, ambas subexpresiones Boolean son evaludadas, desde & toma un argumento Boolean. En el segudo y tercer ejemplos, solo el primero es evaluado, desde and: expera un Block como su argumento. El Blockes evaluado solo si el primer argumento es true.

 *Trate de imaginar cómo* and: *y* or: *están implementados. Chequee las implementaciones en* Boolean, True *y* False.

8.6 Resumen del Capítulo

- Si sobreescribe = debería sobreescribir hash también.
- Sobreescriba postCopy para implementar correctamente la copia de dus objetos.
- Envíe self halt para fijar un punto de ruptura.
- Devuelva self subclassResponsibility para hacer un método abstracto.
- Para dar a un objeto una representación String debería sobreescribir el método printOn:.
- Sobreescriba el método initialize para inicializar apropiadamente las instancias.
- Los métodos de Number automáticamente convierten entre Floats, Fractions e Integers.
- Fractions realmente representa fracciones en lugar de números con punto flotante.
- Characters son instancias únicas.
- Strings son mutables; Symbols no. Sin embargo tenga cuidado de no mutar cadenas literales!
- Symbols son únicos; Strings no.
- Strings y Symbols son Collections y por ello soportan los m'etodos de Collection.

Chapter 9

Colecciones

9.1 Introducción

Las clases de colecciones forman un grupo vagamente definido de subclases de propósito general de Collection y de Stream. El grupo de clases que aparecen en el "Blue Book"[1] contiene 17 subclases de Collection y 9 subclases de Stream, para un total de 28 clases, y ya había sido rediseñado varias veces antes que el sistema Smalltalk-80 fuera publicado. Éste grupo de clases a menudo se considerada un ejemplo paradigmático de diseño orientado a objetos.

En Pharo, la clase abstracta Collection tiene 101 subclases, y la clase abstracta Stream tiene 50 subclases, pero muchas de ellas (como Bitmap, FileStream y CompiledMethod) son clases de propósito especial hechas para uso en otras partes del sistema o en aplicaciones, y por lo tanto no categorizadas como "Collections" por la organización del sistema. A los efectos de este capítulo, usamos el término "Jerarquía de colecciones" para referirnos a Collection y sus 47 subclases que están *además* en las categorías etiquetadas *Collections-**. Usamos el término "Jerarquía de Stream" para referirnos a Stream y sus 9 subclases que están *además* en las categorías *Collections-Streams*. Éstas 56 clases responden a 982 mensajes y definen un total de 1609 métodos!

En éste capítulo nos enfocamos principalmente en el subconjunto de colecciones mostradas en Figure 9.1. Los Streams serán discutidos de forma separada en Chapter 10.

[1] Adele Goldberg and David Robson, *Smalltalk 80: the Language and its Implementation*. Reading, Mass.: Addison Wesley, May 1983, ISBN 0–201–13688–0.

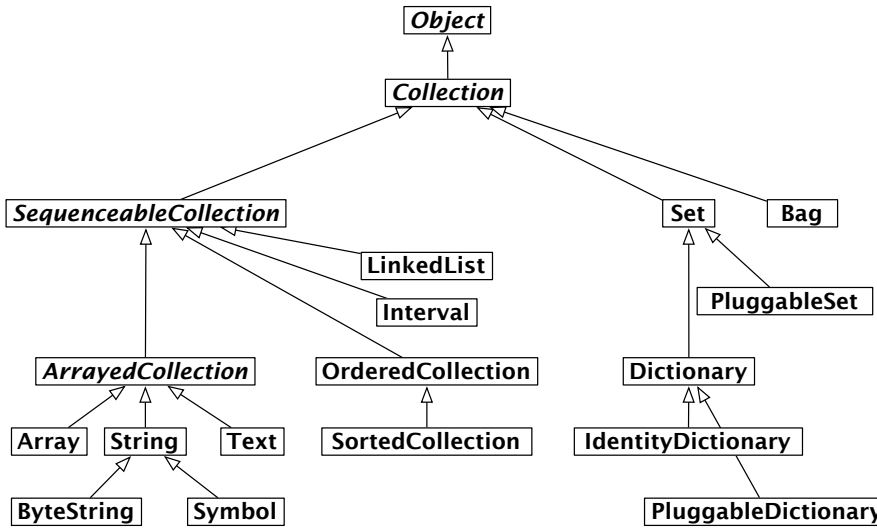


Figure 9.1: Algunas de las clases clave de colecciones en Pharo.

9.2 Las variedades de colecciones

Para usar correctamente las clases de colecciones, el lector necesita al menos un conocimiento superficial de la vasta variedad de colecciones que implementan, y sus puntos en común y diferencias.

Programar con colecciones en lugar de elementos individuales es una manera importante de elevar el nivel de abstracción de un programa. La función map de Lisp, que aplica una función a cada uno de los elementos de una lista y retorna una nueva lista conteniendo el resultado es un ejemplo de primera hora de este estilo, pero Smalltalk-80 adoptó la programación basada en colecciones como un principio básico. Los lenguajes de programación funcional modernos tales como ML y Haskell han seguido el ejemplo de Smalltalk.

¿Por qué es una buena idea? Suponga que usted tiene una estructura de datos que contiene una colección de registros de estudiantes, y desea realizar alguna acción en todos los estudiantes que cumplen algún criterio. Los programadores educados en un lenguaje imperativo pensarán de manera inmediata en un bucle. Pero el programador de Smalltalk escribirá:

```
estudiantes select: [ :estudiante | estudiante promedio < umbral ]
```

que evalúa a una nueva colección que contiene precisamente aquellos ele-

Protocolo	**Métodos**
accessing	size, capacity, at: *anIndex*, at: *anIndex* put: *anElement*
testing	isEmpty, includes: *anElement*, contains: *aBlock*, occurrencesOf: *anElement*
adding	add: *anElement*, addAll: *aCollection*
removing	remove: *anElement*, remove: *anElement* ifAbsent: *aBlock*, removeAll: *aCollection*
enumerating	do: *aBlock*, collect: *aBlock*, select: *aBlock*, reject: *aBlock*, detect: *aBlock*, detect: *aBlock* ifNone: *aNoneBlock*, inject: *aValue* into: *aBinaryBlock*
converting	asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: *aBlock*
creation	with: *anElement*, with:with:, with:with:with:, with:with:with:with:, withAll: *aCollection*

Figure 9.2: Protocolos estándar de Collection

mentos de estudiantes para los cuales la función entre corchetes retorna true[2]. El código Smalltalk tiene la simplicidad y elegancia de un lenguaje de consulta de dominio específico.

El mensaje select: es entendido por *todas* las colecciones en Smalltalk. No hay necesidad de saber si la estructura de datos es un array o una lista enlazada (linked list): el mensaje select: es entendido por ambos tipos de colecciones. Notar que es un poco diferente a usar un loop, donde uno debe saber si estudiantes es un array o una lista enlazada antes de que el loop pueda construirse.

En Smalltalk, cuando uno habla de una colección sin ser más especifico sobre el tipo de colección, uno se refiere a un objeto que soporta un protocolo bien definido para testear inclusión y enumeración de sus elementos. *Todas* las colecciones entienden los mensajes de *testing* includes:, isEmpty y occurrencesOf:. *Todas* las colecciones entienden los mensajes de *enumeration* do:, select:, reject: (que es el opuesto a select:), collect: (que es como el map de LISP), detect:ifNone:, inject:into: (que realiza un fold a izquierda) y muchos más. Es la omnipresencia de este protocolo, como así también su variedad, que lo hace tan poderoso.

Figure 9.2 resume los protocolos estándares soportados por la mayoría de las clases en la jerarquía de colecciones. Estos métodos están definidos, redefinidos, optimizados o incluso ocasionalmente prohibidos por las subclases de Collection.

[2]La expresión entre corchetes puede ser pensada como una expresión λ que define una función anónima $\lambda x.x$ promedio $<$ umbral.

Más allá de esta uniformidad básica, hay muchos tipos de colecciones diferentes también soportando diferentes protocolos, o proveyendo diferente comportamiento para los mismos mensajes. Hagamos un breve estudio de algunas de las principales diferencias:

- **Sequenceable:** Las instancias de las subclases de SequenceableCollection empiezan con un primer elemento (obtenido con first) y son procesadas en un orden bien definido hasta el último elemento (obtenido con last). Las instancias de Set, Bag y Dictionary, por otro lado, no son secuenciables.

- **Sortable:** Una colección SortedCollection mantiene sus elementos en un orden determinado.

- **Indexable:** La mayoría de las colecciones secuenciables son también indexables, esto es, sus elementos pueden ser recuperador con at:. Array es la estructura de datos indexada conocida con un tamaño fijo; unArray at: n recupera el $n-esimo$ elemento de unArray, y unArray at: n put: v cambia el $n-esimo$ elemento a v. las instancias de LinkedList y SkipList son secuenciables pero no indexables, esto es, entienden los mensajes first y last, pero no at:.

- **Keyed:** Las instancias de Dictionary y sus subclases son accedidas por claves en vez de índices.

- **Mutable:** La mayoría de las colecciones son mutables, pero las instancias de Interval y Symbol no lo son. Un Interval es una colección inmutable que representa un rango de Integers. Por ejemplo, 5 to: 16 by: 2 es un intervalo que contiene los elementos 5, 7, 9, 11, 13 y 15. Es indexable con at:, pero no puede ser cambiada usando at:put:.

- **Growable:** Las instancias de Interval y Array son siempre de tamaño fijo. Otro tipo de colecciones (sorted collections, ordered collections, y linked lists) pueden crecer luego de su creación.

  La clase OrderedCollection es más general que Array; el tamaño en una OrderedCollection crece bajo demanda, y tiene métodos para agregar al inicio (addFirst:) y al final (addLast:) así como también at: y at:put:.

- **Duplicados:** Un Set filtrará duplicados, , pero un Bag no. Dictionary, Set y Bag usan el método = provisto en los elementos; las variantes Identity de estas clases usan el método ==, que verifica si los argumentos son el mismo objeto, y las variables Pluggable usan una relación de equivalencia arbitraria suministrada por el creador de la colección.

- **Heterogéneas:** La mayoría de las colecciones contendrán cualquier tipo de elementos. Un String, CharacterArray ó Symbol, sin embargo, sólo contendrán instancias de Character. Un Array contendrá una mezcla de

Arrayed Implementation	Ordered Implementation	Hashed Implementation	Linked Implementation	Interval Implementation
Array String Symbol	OrderedCollection SortedCollection Text Heap	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList SkipList	Interval

Figure 9.3: Algunas clases de colecciones categorizadas por la técnica de implementación.

objetos, pero un ByteArray sólo contiene instancias de Byte, un IntegerArray sólo contiene instancias de Integer y un FloatArray sólo instancias de Float. Una LinkedList está restringida a contener elementos que se ajustan al protocolo *Link ▷ accessing*.

9.3 Implementaciones de colecciones

Estas categorizaciones por funcionalidad no son nuestra única preocupación; debemos considerar también cómo están implementadas las clases de colecciones. Como se muestra en Figure 9.3, cinco técnicas principales de implementación son empleadas.

1. Los Arrays almacenan sus elementos en las variables de instancia (indexables) del objeto colección en sí mismo; como consecuencia, los arrays deben ser de tamaño fijo, pero pueden ser creados con una alocación de memoria simple.

2. OrderedCollections y SortedCollections almacenan sus elementos en un array que es referenciado por una de las variables de instancia de la colección. Consecuentemente, el array interno puede ser reemplazado por uno más largo si la colección crece más allá de su capacidad de almacenamiento.

3. Los varios tipos de conjuntos y diccionarios también referencian un array auxiliar para el almacenamiento, pero usan el array como una tabla hash. Los Bags usan un Dictionary auxiliar, con los elementos de ese bag como claves y el número de ocurrencias como valores.

4. Las LinkedLists usan una representación estándar simplemente enlazada.

5. Los Intervals son representados por tres enteros que registran los dos puntos finales y el tamaño del paso.

Además de estas clases, hay también variantes"weak" de Array, Set y de los varios tipos de dictionary. Éstas colecciones contienen sus elementos de manera débil, *i.e.*, de una forma tal que no previenen que los elementos sean reclamados por el garbage collector. La máquina virtual de Pharo es consciente de estas clases y las maneja de manera especial.

Los lectores interesados en aprender más acerca de las colecciones de Smalltalk pueden referir al excelente libro de LaLonde y Pugh'[3].

9.4 Ejemplos de clases clave

Ahora presentamos las clases más comunes e importantes de colecciones usando ejemplos de código sencillos. Los principales protocolos de colecciones son: at:, at:put: — para acceder a un elemento, add:, remove: — para agregar o quitar un elemento, size, isEmpty, include: — para obtener determinada información sobre la colección, do:, collect:, select: — para iterar sobre la colección. Cada colección podría o no implementar dichos protocolos, y cuando lo hacen, los interpretan para encajar en su semántica. Le sugerimos que heche un vistazo a las clases para identificar protocolos más específicos y avanzados.

Nos enfocaremos en las colecciones más comunes: OrderedCollection, Set, SortedCollection, Dictionary, Interval, y Array.

Protocolo común de creación. Hay muchas maneras de crear instancias de colecciones. Las más genéricas usan los métodos new: y with:. new: anInteger crea una colección de tamaño anInteger cuyos elementos serán todos nil. with: anObject crea una colección y agrega anObject a la colección creada. Diferentes tipos de colecciones realizarán este comportamiento de manera diferente.

Puede crear colecciones con elementos iniciales usando los métodos with:, with:with: etc. para hasta seis elementos.

```
Array with: 1    ⟶    #(1)
Array with: 1 with: 2    ⟶    #(1 2)
Array with: 1 with: 2 with: 3    ⟶    #(1 2 3)
Array with: 1 with: 2 with: 3 with: 4    ⟶    #(1 2 3 4)
Array with: 1 with: 2 with: 3 with: 4 with: 5    ⟶    #(1 2 3 4 5)
Array with: 1 with: 2 with: 3 with: 4 with: 5 with: 6    ⟶    #(1 2 3 4 5 6)
```

[3]Wilf LaLonde and John Pugh, *Inside Smalltalk: Volume 1*. Prentice Hall, 1990, ISBN 0–13–468414–1.

Puede usar también addAll: para agregar todos los elementos de un tipo de colección a otra de otro tipo:

```
(1 to: 5) asOrderedCollection addAll: '678'; yourself  ⟶  an OrderedCollection(1 2 3
    4 5 $6 $7 $8)
```

Tenga cuidado que addAll: también retorna su argumento, y no el objeto receptor!

También puede crear muchas colecciones con withAll: ó newFrom:

```
Array withAll: #(7 3 1 3)                    ⟶  #(7 3 1 3)
OrderedCollection withAll: #(7 3 1 3)        ⟶  an OrderedCollection(7 3 1 3)
SortedCollection withAll: #(7 3 1 3)         ⟶  a SortedCollection(1 3 3 7)
Set withAll: #(7 3 1 3)                      ⟶  a Set(7 1 3)
Bag withAll: #(7 3 1 3)                      ⟶  a Bag(7 1 3 3)
Dictionary withAll: #(7 3 1 3)               ⟶  a Dictionary(1->7 2->3 3->1 4->3 )
```

```
Array newFrom: #(7 3 1 3)                                  ⟶  #(7 3 1 3)
OrderedCollection newFrom: #(7 3 1 3)                      ⟶  an OrderedCollection(7 3 1
    3)
SortedCollection newFrom: #(7 3 1 3)                       ⟶  a SortedCollection(1 3 3 7)
Set newFrom: #(7 3 1 3)                                    ⟶  a Set(7 1 3)
Bag newFrom: #(7 3 1 3)                                    ⟶  a Bag(7 1 3 3)
Dictionary newFrom: {1 -> 7. 2 -> 3. 3 -> 1. 4 -> 3}      ⟶  a Dictionary(1->7 2->3
    3->1 4->3 )
```

Notar que esos métodos no son idénticos. En particular, Dictionary class»withAll: interpreta su argumento como una colección de valores, mientras que Dictionary class»newFrom: espera una colección de asociaciones (associations).

Array

Un Array es una colección de tamaño fijo de elementos accedidos por índices enteros. Al contrario que la convención de C, el primer elemento de un array en Smalltalk está en la posición 1 y no en la 0. El protocolo principal para acceder a los elementos del array es el método at: y at:put:. at: anInteger retorna el elemento en el índice anInteger. at: anInteger put: anObject coloca anObject en el índice anInteger. Los arrays son colecciones de tamaño fijo y por ende no podemos agregar o quitar elementos al final de un array. El siguiente código crea un array de tamaño 5, pone valores en las primeras 3 ubicación y retorna el primer elemento.

```
anArray := Array new: 5.
anArray at: 1 put: 4.
anArray at: 2 put: 3/2.
anArray at: 3 put: 'ssss'.
anArray at: 1  ⟶  4
```

Hay varias formas de crear instancias de la clase Array. Podemos usar new:, with:, y las construcciones #( ) y { }.

Creación con new: new: anInteger crea un array de tamaño anInteger. Array new: 5 crea un array de tamaño 5.

Creación con with: Los métodos with: permiten a uno especificar el valor de los elementos. El siguiente código crea un array de tres elementos que consiste en el número 4, la fracción 3/2 y la cadena de caracteres 'lulu'.

```
Array with: 4 with: 3/2 with: 'lulu'  ⟶  {4 . (3/2) . 'lulu'}
```

Creación de literales con #(). #() crea arrays literales con elementos estáticos (ó "literal") que deben ser conocidos cuando la expresión se compila, y no cuando es ejecutada. El siguiente código crea un array de tamaño 2 donde el primer elemento es el número (literal) 1 y el segundo es la cadena de caracteres (literal) 'here'.

```
#(1 'here') size  ⟶  2
```

Ahora bien, si evalúa #(1+2), no se obtiene un array con un elemento simple 3 sino el array #(1 #+ 2) *i.e.*, con tres elementos: 1, el símbolo #+ y el número 2.

```
#(1+2)  ⟶  #(1 #+ 2)
```

Esto ocurre porque el constructor #() causa que el compilador interprete literalmente la expresión contenida en el array. La expresión es escaneada y los elementos resultantes alimentan el nuevo array. Los arrays literales contienen números, nil, true, false, símbolos y cadenas de caracteres.

Creación dinñamica con { }. Finalmente, puede crear un array de manera dinámica usando el constructor {}. { a . b } es equivalente a Array with: a with: b. Esto siginifica en particular que las expresiones encerradas por { y } son ejecutadas.

```
{ 1 + 2 }  ⟶  #(3)
{(1/2) asFloat} at: 1  ⟶  0.5
{10 atRandom . 1/3} at: 2  ⟶  (1/3)
```

Acceso a los elementos. Los elementos de todas las colecciones secuenciables pueden ser accedidos con at: y at:put:.

```
anArray := #(1 2 3 4 5 6) copy.
anArray at: 3    ⟶    3
anArray at: 3 put: 33.
anArray at: 3    ⟶    33
```

Tenga cuidado con el código que modifica arrays literales! El compilador intenta asignar espacio sólo una vez para arrays literales. A menos que copie el array, la segunda vez que evalúa el código su array "literal" quizás no tenga el valor que espera. (Sin la clonación, la segunda vez, el literal #(1 2 3 4 5 6) será en realidad #(1 2 33 4 5 6)!) Los arrays dinámicos no tienen este problema.

OrderedCollection

OrderedCollection es una de las colecciones que puede crecer de tamaño, y para la cual los elementos pueden ser agregados secuencialmente. Ofrece una variedad de métodos tales como add:, addFirst:, addLast:, y addAll:.

```
ordCol := OrderedCollection new.
ordCol add: 'Seaside'; add: 'SqueakSource'; addFirst: 'Monticello'.
ordCol    ⟶    an OrderedCollection('Monticello' 'Seaside' 'SqueakSource')
```

Quitar elementos. El método remove: anObject remueve la primera ocurrencia de anObject de la colección. Si la colección no incluye el objeto, levanta un error.

```
ordCol add: 'Monticello'.
ordCol remove: 'Monticello'.
ordCol    ⟶    an OrderedCollection('Seaside' 'SqueakSource' 'Monticello')
```

Hay una variante deremove: llamada remove:ifAbsent: que permite a uno especificar como segundo argumento un bloque que es ejecutado en caso que el elemento a ser removido no esté en la colección.

```
res := ordCol remove: 'zork' ifAbsent: [33].
res    ⟶    33
```

Conversión. Es posible obtener una OrderedCollection a partir de un Array (o de cualquier otra colección) enviando el mensaje asOrderedCollection:

```
#(1 2 3) asOrderedCollection    ⟶    an OrderedCollection(1 2 3)
'hello' asOrderedCollection    ⟶    an OrderedCollection($h $e $l $l $o)
```

Interval

La clase Interval representa rangos de números. Por ejemplo, el intervalo de números de 1 a 100 se define de la siguiente manera:

```
Interval from: 1 to: 100    ⟶    (1 to: 100)
```

El método printString de este intervalo revela que la clase Number nos provee un método cómodo llamado to: para generar intervalos:

```
(Interval from: 1 to: 100) = (1 to: 100)    ⟶    true
```

Podemos usar Interval class»from:to:by: ó Number»to:by: para especificar los pasos entre dos números de la siguiente manera:

```
(Interval from: 1 to: 100 by: 0.5) size    ⟶    199
(1 to: 100 by: 0.5) at: 198    ⟶    99.5
(1/2 to: 54/7 by: 1/3) last    ⟶    (15/2)
```

Dictionary

Los diccionarios son colecciones importantes cuyos elementos son accedidos usando claves. Entre los mensajes más comúnmente usados de los diccionarios puede encontrar at:, at:put:, at:ifAbsent:, keys y values.

```
colors := Dictionary new.
colors at: #yellow put: Color yellow.
colors at: #blue put: Color blue.
colors at: #red put: Color red.
colors at: #yellow    ⟶    Color yellow
colors keys           ⟶    a Set(#blue #yellow #red)
colors values         ⟶    {Color blue . Color yellow . Color red}
```

Los diccionarios comparan las claves por igualdad. Dos claves son consideradas las mismas si retornan true cuando se comparan usando =. Un problema común y difícil de reconocer es usar como clave un objeto cuyo método = ha sido redefinido pero no así su método hash. Ambos métodos son usados en la implementación del diccionario y cuando se comparan objetos.

La clase Dictionary ilustra claramente que la jerarquía de colecciones está basada en la subclasificación y no en la subtificación. A pesar que Dictionary es una subclase de Set, no quisiéramos usar normalmente unDictionary donde se espera un Set. En su implementación, sin embargo, un Dictionary puede ser claramente visto como un conjunto de asociaciones (clave valor) creados usando el mensaje −>. Podemos crear un Dictionary a partir de una colección de asociaciones, o bien convertir un diccionario a un array de asociaciones.

```
colors := Dictionary newFrom: { #blue->Color blue. #red->Color red. #yellow->Color
      yellow }.
colors removeKey: #blue.
colors associations    ⟶    {#yellow->Color yellow . #red->Color red}
```

IdentityDictionary. Mientras un diccionario usa el resultado de los mensajes = y hash para determinar si dos claves son la misma, la clase IdentityDictionary usa la identidad (mensaje ==) de las claves en vez de sus valores, *i.e.*, considera dos claves iguales *sólo* si son el mismo objeto.

A menudo las instancias de Symbol son usadas como clave, en cuyo caso es natural usar un IdentityDictionary, dado queSymbol garantiza ser globalmente único. Si, por el contrario, sus claves son cadenas de caracteres (String), es mejor usar un Dictionary, o es probable que tenga problemas:

```
a := 'foobar'.
b := a copy.
trouble := IdentityDictionary new.
trouble at: a put: 'a'; at: b put: 'b'.
trouble at: a          ⟶    'a'
trouble at: b          ⟶    'b'
trouble at: 'foobar'   ⟶    'a'
```

Dado que a y b son objetos diferentes, son tratados como objetos diferentes. Es interesante notar que el literal 'foobar' es instanciado sólo una vez, por lo cual es realmente el mismo objeto que a. Usted no quiere que su código dependa en un comportamiento como éste! Un Dictionary le daría el mismo valor para cualquier clave igual a 'foobar'.

Use sólo objetos únicos globalmente (como instancias de Symbol ó SmallInteger) como claves de un IdentityDictionary, y cadenas de caracteres (u otros objetos) como claves de un Dictionary.

Notar que Smalltalk es una instancia de SystemDictionary, una subclase de IdentityDictionary, y por lo tanto, todas sus claves son instancias de Symbol (en realidad de ByteSymbol, que contiene sólo caracteres de 8 bits).

```
Smalltalk keys collect: [ :each | each class ]    ⟶    a Set(ByteSymbol)
```

Enviando keys ó values a un Dictionary resulta en un Set, que veremos a continuación.

Set

La clase Set es una colección que se comporta como un conjunto matemático, *i.e.*, como una colección sin elementos duplicados y sin ningún orden. En

un Set los elementos se agregan usando el mensaje add: y no pueden ser accedidos usando el mensaje at:. Los objetos colocados en un conjunto deben implementar los métodos hash y =.

```
s := Set new.
s add: 4/2; add: 4; add:2.
s size    ⟶    2
```

También puede crear conjuntos usando Set class»newFrom: o el mensaje de conversión Collection»asSet:

```
(Set newFrom: #( 1 2 3 1 4 )) = #(1 2 3 4 3 2 1) asSet    ⟶    true
```

asSet nos ofrece una forma cómoda de eliminar duplicados de una colección:

```
{ Color black. Color white. (Color red + Color blue + Color green) } asSet size    ⟶    2
```

Notar que red + blue + green = white.

Un Bag es muy similar a un Set excepto que permite duplicados:

```
{ Color black. Color white. (Color red + Color blue + Color green) } asBag size    ⟶    3
```

Las operaciones de conjuntos *unión*, *intersección* e *inclusión* son implementados por los mensajes de Collection union:, intersection: y includes:. El receptor es convertido primero a un Set, por lo que éstas operaciones funcionan para todos los tipos de colecciones!

```
(1 to: 6) union: (4 to: 10)    ⟶    a Set(1 2 3 4 5 6 7 8 9 10)
'hello' intersection: 'there'    ⟶    'he'
#Smalltalk includes: $k    ⟶    true
```

Como explicamos más abajo, los elementos de un conjunto son accedidos usando iteradores (ver Section 9.5).

SortedCollection

En contraste a una OrderedCollection, una SortedCollection mantiene sus elementos en un orden particular.

Por defecto, una sorted collection usa el mensaje <= para establecer el orden de los elementos, por lo cual puede ordenar instancias de subclases de la clase abstracta Magnitude, que define el protocolo de objetos comparables (<, =, >, >=, between:and:...) (Vea Chapter 8).

Puede crear una SortedCollection creando una nueva instancia y agregando elementos a ella:

```
SortedCollection new add: 5; add: 2; add: 50; add: −10; yourself.  ⟶  a
    SortedCollection(−10 2 5 50)
```

Por lo general, sin embargo, uno querrá enviar el mensaje de conversión asSortedCollection a una colección existente:

```
#(5 2 50 −10) asSortedCollection  ⟶  a SortedCollection(−10 2 5 50)
```

Este ejemplo responde la siguiente pregunta frecuente (FAQ):

> FAQ: ¿Cómo se ordena una colección?
> RESPUESTA: Envíe el mensaje asSortedCollection a la colección.

```
'hello' asSortedCollection  ⟶  a SortedCollection($e $h $l $l $o)
```

¿Cómo puede obtener una cadena de caracteres (String) nuevamente a partir de este resultado? asString desafortunadamente retorna la representación dada por printString, que no es lo que queremos:

```
'hello' asSortedCollection asString  ⟶  'a SortedCollection($e $h $l $l $o)'
```

La respuesta correcta es usar String class»newFrom:, String class»withAll: ó Object» as::

```
'hello' asSortedCollection as: String          ⟶  'ehllo'
String newFrom: ('hello' asSortedCollection)   ⟶  'ehllo'
String withAll: ('hello' asSortedCollection)   ⟶  'ehllo'
```

Es posible tener diferentes tipos de elementos en una SortedCollection en tanto sean coparables entre ellos. Por ejemplo podemos tener diferentes tipos de números tales como enteros, de coma flotante y fracciones:

```
{ 5. 2/−3. 5.21 } asSortedCollection  ⟶  a SortedCollection((−2/3) 5 5.21)
```

Imagine que quiere ordenar objetos que no definen el método <= o que quisiera tener un criterio diferente de ordenamiento. Usted puede hacer esto suministrando un bloque de dos argumentos, denominado bloque de ordenamiento o sortblock, a la sorted collection. Por ejemplo, la clase Color no es una subclase de Magnitude y no implementa el método <=, pero podemos especificar un bloque que estableza que los colores deben ser ordenados de acuerdo a la luminiscencia (una medida de brillo).

```
col := SortedCollection sortBlock: [:c1 :c2 | c1 luminance <= c2 luminance].
col addAll: { Color red. Color yellow. Color white. Color black }.
col  ⟶  a SortedCollection(Color black Color red Color yellow Color white)
```

String

Una cadena de caracteres (String) en Smalltalk representa una colección de instancias de Character. Es secuenciable, indexable, mutable y homogénea, y contiene sólo instancias de Character. De la misma forma que los Arrays, los Strings tienen una sintaxis particular, y son creados normalmente especificando directamente un literal entre comillas simples, aunque los mensajes de creación comunes de colecciones también funcionarán.

```
'Hello'                              ⟶  'Hello'
String with: $A                      ⟶  'A'
String with: $h with: $i with: $!    ⟶  'hi!'
String newFrom: #($h $e $l $l $o)    ⟶  'hello'
```

En realidad, String es abstracta. Cuando instanciamos un String podemos obtener un ByteString de 8 bits ó un WideString de 32 bits. Para mantener las cosas simples, nosotros ignoraremos usualmente la diferencia y simplemente hablaremos de instancias de String.

Dos instancias de String pueden ser concatenadas con una coma.

```
s := 'no', ' ', 'worries'.
s   ⟶   'no worries'
```

Dado que un cadena de caracteres es una colección mutable podemos además cambiarla usando el método at:put:.

```
s at: 4 put: $h; at: 5 put: $u.
s   ⟶   'no hurries'
```

Notar que el método coma se define en Collection, por lo cual funcionará para todo tipo de colecciones!

```
(1 to: 3) , '45'   ⟶   #(1 2 3 $4 $5)
```

Podemos tambien modificar un string existente usando replaceAll:with: ó replaceFrom:to:with: como se muestra a continuación. Notar que el número de caracteres y el intervalo deberían tener el mismo tamaño.

```
s replaceAll: $n with: $N.
s   ⟶   'No hurries'
s replaceFrom: 4 to: 5 with: 'wo'.
s   ⟶   'No worries'
```

En contraste a los métodos descriptos anteriormente, el método copyReplaceAll: crea una nueva cadema de caracteres. (Curiosamente, aquí los argumentos son subcadenas y no caracteres individuales, y sus tamaños no deben coincidir necesariamente.)

```
s copyReplaceAll: 'rries' with: 'mbats'    ⟶    'No wombats'
```

Una vista rápida por la implementación de estos métodos revela que están definidos no solamente para los Strings, sino para todos los tipos de SequenceableCollection, por lo que el siguiente código también funciona:

```
(1 to: 6) copyReplaceAll: (3 to: 5) with: { 'three'. 'etc.' }    ⟶    #(1 2 'three' 'etc.' 6)
```

Correspondencia de cadenas de caracteres. Es posible preguntar si un patrón corresponde con una cadena de caracteres enviando el mensaje match:. El patrón puede especificar * para indicar una serie arbitraria de caracteres y # para indicar un caracter simple. Notar que match: es enviado al patrón y no a la cadena de caracteres.

```
'Linux *' match: 'Linux mag'                    ⟶    true
'GNU/Linux #ag' match: 'GNU/Linux tag'    ⟶    true
```

Otro método útil es findString:.

```
'GNU/Linux mag' findString: 'Linux'                                    ⟶    5
'GNU/Linux mag' findString: 'linux' startingAt: 1 caseSensitive: false    ⟶    5
```

Más facilidades avanzadas de correspondencia que ofrecen las capacidades de Perl están también disponibles en el paquete *Regex*.

Algunas pruebas sobre las cadenas de caracteres. Los siguientes ejemplos ilustran el uso de isEmpty, includes: y anySatisfy: que son mensajes adicionales definidos no sólo para las cadenas de caracteres sino también para las colecciones en general.

```
'Hello' isEmpty    ⟶    false
'Hello' includes: $a    ⟶    false
'JOE' anySatisfy: [:c | c isLowercase]    ⟶    false
'Joe' anySatisfy: [:c | c isLowercase]    ⟶    true
```

Plantillas de cadenas de caracteres. Hay tres mensajes que son útiles para manejar cadenas de caracteres templating: format:, expandMacros y expandMacrosWith:.

```
'{1} is {2}' format: {'Pharo' . 'cool'}    ⟶    'Pharo is cool'
```

Los mensajes de la familia de expandMacros ofrecen substitución de variables, usando <n> para el retorno de carro, <t> para la tabulación, <1s>, <2s>, <3s> para los argumentos (<1p>, <2p>, rodean la cadena de caracteres con comillas simples), y <1?valor1:valor2> para condicionales.

```
'look−<t>−here' expandMacros                                    ⟶   'look− −here'
'<1s> is <2s>' expandMacrosWith: 'Pharo' with: 'cool'    ⟶   'Pharo is cool'
'<2s> is <1s>' expandMacrosWith: 'Pharo' with: 'cool'    ⟶   'cool is Pharo'
'<1p> or <1s>' expandMacrosWith: 'Pharo' with: 'cool'    ⟶   '"Pharo" or Pharo'
'<1?Quentin:Thibaut> plays' expandMacrosWith: true       ⟶   'Quentin plays'
'<1?Quentin:Thibaut> plays' expandMacrosWith: false      ⟶   'Thibaut plays'
```

Otros métodos útiles. La clase String ofrece otras numerosas utilidades incluyendo el mensaje asLowercase, asUppercase y capitalized.

```
'XYZ' asLowercase    ⟶   'xyz'
'xyz' asUppercase    ⟶   'XYZ'
'hilaire' capitalized    ⟶   'Hilaire'
'1.54' asNumber      ⟶   1.54
'esta sentencia es sin lugar a dudas demasiado larga' contractTo: 20    ⟶   'esta sent
    ...do larga'
```

Notar que hay una diferencia entre preguntar a un objeto por su representación como cadena de caracteres enviando el mensaje printString y convertirlo en una cadena de caracteres enviando el mensaje asString. Aquí hay un ejemplo de la diferencia.

```
#ASymbol printString    ⟶   '#ASymbol'
#ASymbol asString       ⟶   'ASymbol'
```

Un símbolo es similar a una cadena de caracteres pero garantiza que es único a nivel global. Por ésta razón los símbolos son preferidos por sobre las cadenas de caracteres como claves de diccionarios, en particular para instancias de IdentityDictionary. Vea también Chapter 8 para más información sobre cadenas de caracteres (String) y símbolos (Symbol).

9.5 Iteradores de colecciones

En Smalltalk los bucles y condicionales son simplemente mensajes enviados a las colecciones u otros objetos tales como enteros o bloques (ver también Chapter 3). Además de los mensajes de bajo nivel tales como to:do: que evalúa un bloque con un argumento que van desde un número inicial a uno final, la jerarquía de colecciones de Smalltalk ofrece iteradores de alto nivel. Usando estos iteradores su código será más robusto y compacto.

Iterando (do:)

El método do: es el iterador básico de colecciones. Aplica su argumento (un bloque que toma un sólo argumento) a cada uno de los elementos del receptor. El siguiente ejemplo imprime en el Transcript todas las cadenas de caracteres contenidas en el receptor.

```
#('bob' 'joe' 'toto') do: [:each | Transcript show: each; cr].
```

Variantes. Hay un montón de variantes de do:, tales como do:without:, doWithIndex: y reverseDo:. Para las colecciones indexadas (Array, OrderedCollection, SortedCollection) el método doWithIndex: también da acceso al índice actual. éste método está relacionado a to:do: que está definido en la clase Number.

```
#('bob' 'joe' 'toto') doWithIndex: [:each :i | (each = 'joe') ifTrue: [ ↑ i ] ]   ⟶   2
```

Para colecciones ordenadas, reverseDo: recorre la colección en el orden inverso.

El siguiente código muestra un mensaje interesante: do:separatedBy: que ejecuta el segundo bloque sólo entre dos elementos.

```
res := ''.
#('bob' 'joe' 'toto') do: [:e | res := res, e ] separatedBy: [res := res, '.'].
res   ⟶   'bob.joe.toto'
```

Notar que este código no es especialmente eficiente dado que crea cadenas de caracteres intermedias y sería mejor usar un write stream para almacenar el resultado (ver Chapter 10):

```
String streamContents: [:stream | #('bob' 'joe' 'toto') asStringOn: stream delimiter: '.' ]
        ⟶   'bob.joe.toto'
```

Diccionarios. Cuando el mensaje do: es enviado a un diccionario, los elementos tomados en cuenta son valores, no las asociaciones. Los métodos adecuados para usar son keysDo:, valuesDo:, y associationsDo:, que iteran en las claves, valores o asociaciones respectivamente.

```
colors := Dictionary newFrom: { #yellow −> Color yellow. #blue −> Color blue. #red −>
      Color red }.
colors keysDo: [:key | Transcript show: key; cr].                "muestra las claves"
colors valuesDo: [:value | Transcript show: value;cr].           "muestra los valores"
colors associationsDo: [:value | Transcript show: value;cr].     "muestra las asociaciones"
```

Colectando resultados (collect:)

Si quiere procesar los elementos de una colección y producir una nueva como resultado, en vez de usar do:, es mejor usar collect:, o uno de los otros métodos iteradores. La mayoría de éstos pueden ser encontrados en el protocolo *enumerating* de Collection y sus subclases.

Imagine que queremos una colección que contenga los dobles de los elementos en otra colección. Usando el método do: debemos escribir lo siguiente:

```
double := OrderedCollection new.
#(1 2 3 4 5 6) do: [:e | double add: 2 * e].
double    ⟶    an OrderedCollection(2 4 6 8 10 12)
```

El método collect: ejecuta su bloque argumento para cada elemento y retorna una nueva colección que contiene los resultados. Usando collect: en cambio, el código es mucho más simple:

```
#(1 2 3 4 5 6) collect: [:e | 2 * e]    ⟶    #(2 4 6 8 10 12)
```

Las ventajas de collect: por sobre do: son incluso más dramáticas en el siguiente ejemplo, donde tomamos una colección de enteros y generamos como resultado una colecció de los valores absolutos de esos enteros:

```
aCol := #( 2 −3 4 −35 4 −11).
result := aCol species new: aCol size.
1 to: aCol size do: [ :each | result at: each put: (aCol at: each) abs].
result    ⟶    #(2 3 4 35 4 11)
```

Contraste el código anterior con la expresión mucho más simple a continuación:

```
#( 2 −3 4 −35 4 −11) collect: [:each | each abs ]    ⟶    #(2 3 4 35 4 11)
```

Una ventaja adicional de la segunda solución es que va a funcionar también para conjuntos y bolsas (bags).

En general usted debería evitar el uso del do:, a menos que quiera enviar mensajes a cada uno de los elementos de la colección.

Notar que el envío del mensaje collect: retorna el mismo tipo de colección que el receptor. Por esta razón el siguiente código falla. (Un String no puede contener valores enteros.)

```
'abc' collect: [:ea | ea asciiValue ]    "error!"
```

Debemos primero convertir la cadena de caracteres a un Array o a una OrderedCollection:

```
'abc' asArray collect: [:ea | ea asciiValue ]    ⟶    #(97 98 99)
```

En realidad collect: no garantiza retornar una colección de exactamente la misma clase que el receptor, pero sí de la misma *"especie"*. En el caso de un Interval, la especie es en realidad un Array!

```
(1 to: 5) collect: [ :ea | ea * 2 ]    ⟶    #(2 4 6 8 10)
```

Selección y rechazo de elementos

select: retorna los elementos del receptos que satisfacen una condición particular:

```
(2 to: 20) select: [:each | each isPrime]    ⟶    #(2 3 5 7 11 13 17 19)
```

reject: hace lo contrario:

```
(2 to: 20) reject: [:each | each isPrime]    ⟶    #(4 6 8 9 10 12 14 15 16 18 20)
```

Identificación de un elemento con detect:

El método detect: retorna el primer elemento del receptor que satisface el bloque argumento (el bloque retorna true al evaluarse).

```
'through' detect: [:each | each isVowel]    ⟶    $o
```

El método detect:ifNone: es una variante del método detect:. El segundo bloque es evaluado cuando no hay elementos que satisfacen el bloque.

```
Smalltalk allClasses detect: [:each | '*cobol*' match: each asString] ifNone: [ nil ]
        ⟶    nil
```

Acumulación de resultados con inject:into:

Los lenguajes de programación funcionales ofrecen frecuentemente una función de alto orden llamada *fold* ó *reduce* para acumular el resultado de aplicar algún operador binario iterativamente sobre todos los elementos de una colección. En Pharo ésto se hace con Collection»inject:into:.

El primer argumento es el valor inicial, y el segundo argumento es un bloque de dos argumentos que es aplicado al resultado parcial acumulado, y cada elemento de la colección a su vez.

Una aplicación trivial de inject:into: es producir la suma de una colección de números. Siguiendo a Gauss, en Pharo podemos escribir esta expresión para sumar los primeros 100 enteros:

```
(1 to: 100) inject: 0 into: [:sum :each | sum + each ]    ⟶    5050
```

Otro ejemplo es el siguiente bloque de un argumento que computa factoriales:

```
factorial := [:n | (1 to: n) inject: 1 into: [:product :each | product * each ] ].
factorial value: 10    ⟶    3628800
```

Otros mensajes

count: El mensaje count: retorna el número de elementos que satisfacen una condición. La condición está representada como un bloque booleano.

```
Smalltalk allClasses count: [:each | 'Collection*' match: each asString ]    ⟶    3
```

includes: El mensaje includes: verifica si el argumento dado está en la colección.

```
colors := {Color white . Color yellow. Color red . Color blue . Color orange}.
colors includes: Color blue.    ⟶    true
```

anySatisfy: El mensaje anySatisfy: responde true si al menos un elemento de la colección satisface la condición representada por el argumento (un bloque).

```
colors anySatisfy: [:c | c red > 0.5]    ⟶    true
```

9.6 Algunas sugerencias para usar colecciones

Un error común con add: . El siguiente error es uno de los más frecuentes en Smalltalk.

```
collection := OrderedCollection new add: 1; add: 2.
collection    ⟶    2
```

Aquí la variable collection no contiene la colección recientemente creada sino el último número agregado. Esto se debe a que el método add: retorna el elemento agregado y no el receptor.

El siguiente código produce el resultado esperado:

```
collection := OrderedCollection new.
collection add: 1; add: 2.
collection    ⟶    an OrderedCollection(1 2)
```

Puede también usar el mensaje yourself para retornar el receptor de una cascada de mensajes:

```
collection := OrderedCollection new add: 1; add: 2; yourself    ⟶    an
    OrderedCollection(1 2)
```

Remoción de un elemento de la colección que se está recorriendo. Otro error que puede hacer es remover un elemento de una colección la cual usted está recorriendo. remove:

```
range := (2 to: 20) asOrderedCollection.
range do: [:aNumber | aNumber isPrime ifFalse: [ range remove: aNumber ] ].
range    ⟶    an OrderedCollection(2 3 5 7 9 11 13 15 17 19)
```

Este resultado es claramente incorrecto dado que 9 y 15 deberían haber sido filtrados!

La solución es copiar la colección antes de recorrerla.

```
range := (2 to: 20) asOrderedCollection.
range copy do: [:aNumber | aNumber isPrime ifFalse: [ range remove: aNumber ] ].
range    ⟶    an OrderedCollection(2 3 5 7 11 13 17 19)
```

Redefinición de = y hash. Un error difícil de detectar es cuando redefine = pero no hash. Los síntomas son que perderá los elementos cuando los agrega a conjuntos u otros comportamientos extraños. Una solución propuesa por Kent Beck es usar xor: para redefinir hash. Suponga que queremos que dos libros sean considerados iguales si sus títulos y autores son los mismos. Entonces redefiniríamos no sólo = sino también hash de la siguiente manera:

Method 9.1: *Redefining* = *and* hash.

```
Book»= aBook
  self class = aBook class ifFalse: [↑ false].
  ↑ title = aBook title and: [ authors = aBook authors]

Book»hash
  ↑ title hash xor: authors hash
```

Otro problema grave surge si usa objetos mutables, *i.e.*, un objeto que puede cambiar su valor de hash a través del tiempo, como un elemento de

un Set o como clave de un Dictionary. No haga esto a no ser que le guste mucho la depuración de código!

9.7 Resumen del capítulo

La jerarquía de colecciones de Smalltalk provee un vocabulario común para manipular uniformemente una variedad de diferentes tipos de colecciones.

- Una distinción clave es entre colecciones secuenciables (subclases de SequenceableCollection), que mantienen sus elementos en un orden dado, diccionarios (Dictionary) y sus subclases, que mantienen asociaciones clave-valor, y conjuntos (Set) y Bags, que no tienen orden.
- Puede convertir la mayoría de las colecciones en otro tipo de colección enviándoles los mensajes asArray, asOrderedCollection etc..
- Para ordenar una colección, enviar el mensaje asSortedCollection.
- Los Arrays literales son creados con una sintaxis especial #(...). Los Array s dinámicos son creador con la sintaxis { ... }.
- Un Dictionary compara las claves por igualdad. Es más útil cuando las claves sin instancias de String. Un IdentityDictionary sin embargo usa la identidad de los objetos para comparar las claves. Es más adecuado cuando los símbolos (Symbols) son usados como claves, o cuando mapping object references to values.
- Las cadenas de caracteres (Strings) también entienden los mensajes usuales de colecciones. Además, una cadena de caracteres soporta una forma simple de *pattern-matching*. Para una aplicación más avanzada, buscar en el paquete RegEx.
- El mensaje básico de iteración es do:. Es útil para código imperativo, tal como la modificación de cada elemento de una colección, o el envío de un mensaje a cada elemento.
- En vez de usar do:, es más usual usar collect:, select:, reject:, includes:, inject:into: y otros mensajes de alto nivel para procesar colecciones de manera uniforme.
- Nunca remover un elemento de una colección que se está recorriendo. Si se debe modificar, oterar sobre una copia de ella.
- Si redefine el método =, recuerde redefinir también el método hash!

Chapter 10

Streams

Los Streams son usados para iterar sobre secuencias de elementos como colecciones secuenciadas, archivos y streams de red. Los Streams pueden ser de lectura, de escritura o de ambos. Leer o escribir es siempre relativo a la posición en el stream. Los Streams pueden ser fácilmente convertidos en colecciones y vice versa.

10.1 Dos secuencias de elementos

Una buena metáfora para entender un stream es las siguiente: Un stream puede ser representado como dos secuencias de elementos: una secuencia de elementos pasados y una secuencias de elementos futuros. El stream es posicionado entre las dos secuencias. Entender este modelo es importante ya que toda operación en Smalltalk se basa en el.

Por esta razón, la mayoria de las classes Stream son subclases de PositionableStream. Figure 10.1 presenta un stream con cinco caracteres. Este stream está es su posición original, *i.e.*, no hay elementos anteriores. Puedes volver a esta posición usando el mensaje reset.

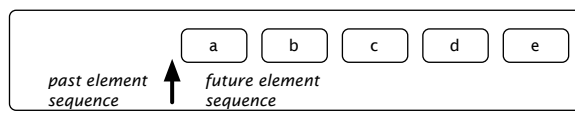


Figure 10.1: Un stream posicionado en su comienzo.

Leer un elemento conceptualmente significa sacar el primer elemento de la secuencia de elementos futuros y ponerlo despues del último en la se-

cuencia de elementos pasados. Luego de haber leido un elemento usando el mensaje next, el estado de tu stream es el mostrado en Figure 10.2.

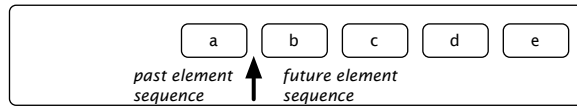


Figure 10.2: El mismo stream despues de la ejecución del método next: ell caracter a está "en el pasado" mientras b, c, d y e estan "en el futuro".

Escribir un elemento significa reemplazar el primer elemento de la secuencia futura por el nuevo y moverlo hacia atras. Figure 10.3 muestra el estado del mismo stream despues de haber escrito a x usando el mensaje nextPut: anElement.

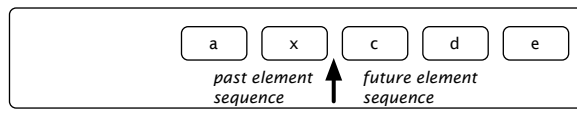


Figure 10.3: El mismo stream despues de haber escrito una x.

10.2 Streams vs. colecciones

El protocolo de colección soporte el almacenamiento, la remoción y enumeración de los elementos de una colección, but no permite que estas operaciones sean entremezcladas. Por ejemplo, si los elementos de una OrderedCollection son procesados por un método do:, no es posible agregar o remover elementos desde el interior del bloque do:. El protocolo de colección tampoco ofrece formas de interar sobre dos colecciones al mismo tiempo, eligiendo cual colección avanza y cual no. Procedimientos como estos requieren que un índice transversal o una posición de referencia sea mantenida fuera de la propia colección: este es exactamente el rol de ReadStream, WriteStream y ReadWriteStream.

Estas tres clases estan definidas para trabajar sobre alguna colección. Por ejemplo, el siguiente fragmento de código crea un stream sobre un intérvalo y luego lee dos elementos

```
r := ReadStream on: (1 to: 1000).
r next.     ⟶   1
r next.     ⟶   2
r atEnd.    ⟶   false
```

WriteStreams pueden escribir datos en una colección:

```
w := WriteStream on: (String new: 5).
w nextPut: $a.
w nextPut: $b.
w contents.    ⟶    'ab'
```

También es posible crear ReadWriteStreams que soporte ambos protocolos de escritura y lectura.

El principal problema con WriteStream y ReadWriteStream es que solo soportan arreglos y cadenas de caracteres en Pharo. Esto está siendo cambiado actualmente con el desarrollo de una nueva biblioteca llamada Nile, pero por ahora si intentas trabajar sobre otra clase de colección, obtendrás un error:

```
w := WriteStream on: (OrderedCollection new: 20).
w nextPut: 12.    ⟶    raises an error
```

Los Streams no estan solo pensados para colecciones, también pueden ser usados con archivos o sockets. El siguiente ejemplo crea un archivo llamado test.txt, escribe dos cadenas de caracteres separadas por un retorno de carro y cierra el archivo.

```
StandardFileStream
 fileNamed: 'test.txt'
 do: [:str | str
            nextPutAll: '123';
            cr;
            nextPutAll: 'abcd'].
```

Las siguientes secciones presentan los protocoles en mayor profundidad.

10.3 Streaming over collections

Los streams son realmente útiles para trabajar con colecciones de elementos. Pueden ser usados para leer y escribir elementos en colecciones Ahora exploraremos las caracteristicas de los streams para las colecciones.

Leyendo colecciones

Esta sección presenta caracteristicas usadas para leer colecciones. Usar un stream para leer una colección provee un puntero a la colección. Dicho puntero se moverá hacia adelante leyendo y podrás ubicarlo donde gustes.

La clase ReadStream deberia usarse para leer elementos de la colección.

Los métodos next ay next: son utilizados para obtener uno o más elementos de una colección.

```
stream := ReadStream on: #(1 (a b c) false).
stream next.    ⟶    1
stream next.    ⟶    #(#a #b #c)
stream next.    ⟶    false
```

```
stream := ReadStream on: 'abcdef'.
stream next: 0.    ⟶    ''
stream next: 1.    ⟶    'a'
stream next: 3.    ⟶    'bcd'
stream next: 2.    ⟶    'ef'
```

El mensaje peek es usado cuando quieres conocer el siguiente elemento en el stream sin avanzar.

```
stream := ReadStream on: '−143'.
negative := (stream peek = $−).    "look at the first element without reading it"
negative.    ⟶    true
negative ifTrue: [stream next].    "ignores the minus character"
number := stream upToEnd.
number.    ⟶    '143'
```

Este código establece la variable booleana negative de acuerdo al signo del número en el stream y number a su valor absoluto. El método upToEnd devuelve todo desde la posición actual hasta el final del stream y establece el stream hasta su final. Este código puede ser simplificado usando peekFor:, que adelanta si el siguiente elemento es igual al parámetro y no se mueve en otro caso.

```
stream := '−143' readStream.
(stream peekFor: $−)    ⟶    true
stream upToEnd          ⟶    '143'
```

peekFor: también devuelve un booleano indicanto si el parámetro es igual al elemento.

Puede que hayas notado un nuevo modo de construir un stream en el ejemplo anterior: uno puede simplemente enviar readStream a una colección

secuenciable para obtener un stream de lectura de una colección particular. collection.

Posicionamiento. Hay métodos para posicionar el puntero del stream. Si tienes el índice, puedes usarlo directamente usando position:. Puedes pedir la posición usando position. Por favor recuerda que un stream no está posicionado en un elemento, sino entre dos elementos. El índice correspondiente al comienzo del stream es 0.

Puedes obtener el estado del stream representado en Figure 10.4 con el siguiente código

```
stream := 'abcde' readStream.
stream position: 2.
stream peek    ⟶    $c
```

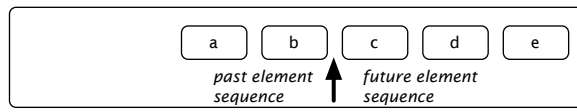


Figure 10.4: Un stream en la posición 2

Para posicionar el stream al principio o al final, puedes usar reset ó setToEnd. skip: y skipTo: son usados para adelantar hasta una posición relativa a la posición actual: skip: toma un número como parámetro y saltea ese número de elementos, mientras que skipTo: saltea todos los elementos en el stream hasta que encuentra un elemento igual a su parámetro. Notar que posiciona el stream después del elemento encontrado.

```
stream := 'abcdef' readStream.
stream next.          ⟶    $a    "stream is now positioned just after the a"
stream skip: 3.                   "stream is now after the d"
stream position.      ⟶    4
stream skip: −2.                  "stream is after the b"
stream position.      ⟶    2
stream reset.
stream position.      ⟶    0
stream skipTo: $e.                "stream is just after the e now"
stream next.          ⟶    $f
stream contents.      ⟶    'abcdef'
```

Como puedes ver, la letra e ha sido salteada.

El método contents siempre devuelve una copia completa del stream.

Testing. Algunos métodos te permiten probar el estado del stream actual: atEnd devuelve verdadero si y solo si no hay más elementos que puedan ser leídos, mientras que isEmpty devuelve verdadero si y solo si no hay más elementos en la colección.

Aquí hay una posible implementación de un algoritmo usando {atEnd que toma dos colecciones ordenadas como parámetros y las mezcla, creando otra colección ordenada.

```
stream1 := #(1 4 9 11 12 13) readStream.
stream2 := #(1 2 3 4 5 10 13 14 15) readStream.

"The variable result will contain the sorted collection."
result := OrderedCollection new.
[stream1 atEnd not & stream2 atEnd not]
  whileTrue: [stream1 peek < stream2 peek
    "Remove the smallest element from either stream and add it to the result."
    ifTrue: [result add: stream1 next]
    ifFalse: [result add: stream2 next]].

"One of the two streams might not be at its end. Copy whatever remains."
result
  addAll: stream1 upToEnd;
  addAll: stream2 upToEnd.

result.   ⟶   an OrderedCollection(1 1 2 3 4 4 5 9 10 11 12 13 13 14 15)
```

Escribiendo colecciones

Ya hemos visto como leer una colección iterando sobre sus elementos, usando un ReadStream. Ahora aprenderemos como crear colecciones usando WriteStreams.

Los WriteStreams son útiles para agregar muchos datos a una colección en varias posiciones. A menudo son usados para construir cadenas de caracteres basadas en partes estáticas o dinámicas, como en este ejemplo:

```
stream := String new writeStream.
stream
  nextPutAll: 'This Smalltalk image contains: ';
  print: Smalltalk allClasses size;
  nextPutAll: ' classes.';
  cr;
  nextPutAll: 'This is really a lot.'.

stream contents.   ⟶   'This Smalltalk image contains: 2322 classes.
This is really a lot.'
```

Esta técnica es usada en diferentes implementaciones del método printOn: por ejemplo. Hay una manera más simple y eficiente de crear streams, si solo estas interesado en el contenido del stream:

```
string := String streamContents:
      [:stream |
         stream
            print: #(1 2 3);
            space;
            nextPutAll: 'size';
            space;
            nextPut: $=;
            space;
            print: 3.  ].
string.    ⟶      '#(1 2 3) size = 3'
```

El método streamContents: crea una colección y un stream en esa colección por ti. Luego ejecuta el bloque que le has indicado, pasandoo el stream como parámetro. Cuado el bloque termina, streamContents: devuelve el contenido de la colección.

Los siguientes métodos WriteStream son especialmente útiles en este contexto:

nextPut: agrega el parámetro al stream;

nextPutAll: agrega cada elemento de la colección, pasado como parámetro, al stream;

print: agrega la representación textual del parámetro al stream.

También hay métodos útiles para imprimir diferentes tipos de caracteres en el stream, como space, tab y cr (carriage return). Otro método de utilidad es ensureASpace, que asegura que el último caracter en el stream es un espacio; si el último caracter no es un espacio, agrega uno.

Sobre la Concatenación. Usar nextPut: y nextPutAll: en un WriteStream es a menudo la mejora forma de concatenar caracteres. Usar el operador de concatenación por comas (,) es mucho menos eficiente.

```
[| temp |
  temp := String new.
  (1 to: 100000)
    do: [:i | temp := temp, i asString, ' ']] timeToRun    ⟶    115176 "(milliseconds)"

[| temp |
  temp := WriteStream on: String new.
  (1 to: 100000)
```

```
  do: [:i | temp nextPutAll: i asString; space].
 temp contents] timeToRun    ⟶    1262 "(milliseconds)"
```

to help you do this: La razón por la que usar un stream puede ser mucho más eficiente es que la coma crea una nuva cadena de caracteres conteniendo la concatenación del receptor del mensaje y el argumento, por lo que tiene que copiar ambos. Cuando concatenas repetidamente en el mismo receptor, se hace cada vez más largo, por lo que el número de caracteres que hay que copiar crece exponencialmente. Esto también crea mucha basura, que debe ser recolectada. Usar un stream en lugar de concatenación de cadenas es una optimización conocida.

```
String streamContents: [ :tempStream |
 (1 to: 100000)
    do: [:i | tempStream nextPutAll: i asString; space]]
```

Leyendo y escribiendo al mismo tiempo

Es posible usar un stream para acceder a una colección para leer y escribir al mismo tiempo. Imagina que quieres crear una clase History que manejará los botones adelante y atrás en un navegador web. El historial reaccionará como en las figuras 10.5 a 10.11.

Figure 10.5: El nuevo historial está vacío. No se muestra nada en el navegador web.

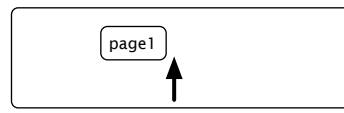


Figure 10.6: El usuario abre la página 1.

Este comportamiento puede ser implementado usando un ReadWriteStream.

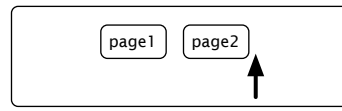


Figure 10.7: El usuario hace clic en un enlace a la página 2.

Figure 10.8: El usuario hace clic en un enlace a la página 3.

Figure 10.9: El usuario hace clic en el botón atrás. Ahora está viendo la página 2 nuevamente.

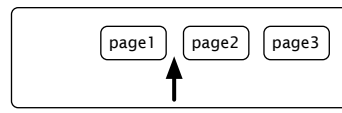


Figure 10.10: El usuario hace clic nuevamente en el botón atrás. Ahora se muestra la página 1.

Figure 10.11: Desde la página 1, el usuario hace clic en un enlace a la página 4. El historial olvida las páginas 2 y 3.

```
Object subclass: #History
  instanceVariableNames: 'stream'
  classVariableNames: ''
```

```
 poolDictionaries: ''
 category: 'PBE−Streams'

History>>initialize
   super initialize.
   stream := ReadWriteStream on: Array new.
```

Nada realmente difícil, definimos una nueva clase que contiene un stream. El stream es creado durante el método initialize. Necesitamos métodos para ir hacia adelante y hacia atrás.

```
History>>goBackward
 self canGoBackward ifFalse: [self error: 'Already on the first element'].
 stream skip: −2.
 ↑ stream next.

History>>goForward
 self canGoForward ifFalse: [self error: 'Already on the last element'].
 ↑ stream next
```

Hasta acá, el código era bastante directo. Ahora tenemos que tratar con el método goTo:, que debería ser activado cuando el usuario hace clic en un enlace. Una posible solución es:

```
History>>goTo: aPage
   stream nextPut: aPage.
```

Sin embargo esta versíon está incompleta. Esto es porque cuando el usuario hace clic en el enlace, no debería haber más páginas donde ir hacia adelante, *i.e.*, el botón adelante debe ser desactivado. Para esto, la solución más simple es escribir nil justo después para indicar el final del historial.

```
History>>goTo: anObject
 stream nextPut: anObject.
 stream nextPut: nil.
 stream back.
```

Ahora, solo queda implementar los métodos canGoBackward y canGoForward.

Un stream siempre se posiciona entre dos elementos. Para ir hacia atrás, debe haber dos páginas antes de la posición actual: una es la página actual, y la otra es la página a la que queremos ir.

```
History>>canGoBackward
 ↑ stream position > 1

History>>canGoForward
 ↑ stream atEnd not and: [stream peek notNil]
```

Agreguemos un método para ver los cotenidos del stream:

```
History>>contents
 ↑ stream contents
```

Y el historial funciona como anunciamos:

```
History new
   goTo: #page1;
   goTo: #page2;
   goTo: #page3;
   goBackward;
   goBackward;
   goTo: #page4;
   contents    ⟶    #(#page1 #page4 nil nil)
```

10.4 Usando streams para acceso a archivos

Ya has visto como usar streams con colecciones de elementos. También es posible usar streams con archivos en tu disco. Una vez creado, un stream en un archivo es como un stream en una colección: podrás usar el mismo protocolo para leer, escribir o posicionar el stream. La diferencia más importante aparece en la creación del stream. Hay muchas formas diferentes de crear streams de archivos, como veremos a continuación.

Creating file streams

Creando streams de archivos

Para crear streams de archivos, tendrás que usar uno de los métodos de instanciación que ofrece la clase FileStream:

fileNamed: Abre un archivo con el nombre dado para lectura y escritura. Si el archivo existe, su contenido previo puede ser modificado o reemplazado, pero el archivo no será truncado al cerrarse. Si el nombre no contiene la ruta, el archivo se creará en el directorio por defecto.

newFileNamed: Crea un nuevo archivo con el nombre dado, y devuelve un stream abierto para escritura en ese archivo. Si el archivo ya existe pregunta al usuario que hacer.

forceNewFileNamed: Crea un nuevo archivo con el nombre dado, y devuelve un stream abierto para escritura en ese archivo. Si el archivo ya existe lo borra sin preguntar.

oldFileNamed: Abre un archivo existente con el nombre dado para lectura y escritura. Si el archivo existe, su contenido previo puede ser modificado o reemplazado, pero el archivo no será truncado al cerrarse. Si el nombre no especifica la ruta, el archivo se creará en el directorio por defecto.

readOnlyFileNamed: Abre un archivo existente con el nombre dado solo para lectura.

Recuerda que cada vez que abres un stream sobre un archivo, también debes cerrarlo. Esto se hace con el método close.

```
stream := FileStream forceNewFileNamed: 'test.txt'.
stream
   nextPutAll: 'This text is written in a file named ';
   print: stream localName.
stream close.

stream := FileStream readOnlyFileNamed: 'test.txt'.
stream contents.    ⟶    'This text is written in a file named "test.txt"'
stream close.
```

El método localName devuelve el último componente del nombre del archivo. Puedes acceder al nombre completo con el método fullName.

Pronto notarás que cerrar los stream de archivos manualmente es doloroso y propenso a errores. Es por eso que ctFileStream ofrece un mensaje llamado forceNewFileNamed:do: para cerrar automáticamente un nuevo stream luego de evaluar un bloque que modifica sus contenidos.

```
FileStream
   forceNewFileNamed: 'test.txt'
   do: [:stream |
      stream
         nextPutAll: 'This text is written in a file named ';
         print: stream localName].
string := FileStream
         readOnlyFileNamed: 'test.txt'
         do: [:stream | stream contents].
string    ⟶    'This text is written in a file named "test.txt"'
```

Los métodos de creación de streams que toman un bloque como argumento primero crean un stream sobre un archivo, luego ejecutan el bloque con el stream como argumento, y finalmente cierran el stream. Estos métodos devuelven lo que es devuelto por el bloque, o sea, el valor de la última expresión del bloque. Esto es usado en el ejemplo anterior para obtener el contenido del archivo y colocarlo en la variable string.

Binary streams

Streams binarios

Por defecto, los streams se crean en texto, lo que significa que leerás y escribirás caracteres. Si tu stream debe ser binario, tienes que enviar el mensaje binary a tu stream.

Cuando tu stream está en modo binario, solo puedes escribir números desde 0 a 255 (1 byte). Si quieres usar nextPutAll: para escribir más de un número al mismo tiempo, tienes que pasar un ByteArray como argumento.

```
FileStream
 forceNewFileNamed: 'test.bin'
 do: [:stream |
       stream
         binary;
         nextPutAll: #(145 250 139 98) asByteArray].

FileStream
 readOnlyFileNamed: 'test.bin'
 do: [:stream |
       stream binary.
       stream size.          ⟶   4
       stream next.          ⟶   145
       stream upToEnd.       ⟶   #[250 139 98]
   ].
```

Aquí hay otro ejemplo que crea una imágen en un archivo llamado "test.pgm" (formato de archivo Portable Graymap). Puedes abrir este archivo con tu programa de dibujo favorito.

```
FileStream
 forceNewFileNamed: 'test.pgm'
 do: [:stream |
   stream
      nextPutAll: 'P5'; cr;
      nextPutAll: '4 4'; cr;
      nextPutAll: '255'; cr;
      binary;
      nextPutAll: #(255 0 255 0) asByteArray;
      nextPutAll: #(0 255 0 255) asByteArray;
      nextPutAll: #(255 0 255 0) asByteArray;
      nextPutAll: #(0 255 0 255) asByteArray
   ]
```

Esto crea un checkerboard de 4x4, como se ve en Figure 10.12.

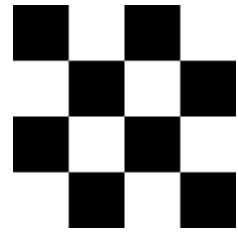


Figure 10.12: Un checkerboard de 4x4 que puedes dibujar usando streams binarios.

10.5 Resúmen del capítulo

Los streams ofrecen una mejor manera que las colecciones para leer y escribir incrementalmente una secuencia de elementos. Hay maneras sencillas de convertir streams en colecciones y viceversa.

- Los streams pueden ser de escritura, de lectura o ambos al mismo tiempo.
- Para convertir una colección en un stream, define un stream "en" una colección, *e.g.*, `ReadStream on: (1 to: 1000)`, o envía los mensajes `readStream`, etc. a la colección.
- Para convertir un stream en una colección, envía el mensaje `contents`.
- para concatenar colecciones largas, en lugar de usar el operador coma, es más eficiente crear un stream, anexar las colecciones al stream con `nextPutAll:`, y extraer el resultado enviando `contents`.
- Los streams de archivos son por defecto basados en texto. Envía `binary` para hacerlos binarios explícitamente.

Chapter 11

Morphic

Morphic es el nombre de la interface gráfica de Pharo. Morphic está escrito en Smalltalk, por lo que es complementamente portable entre distintos sistemas operativos; como consecuencia de ello, Pharo se ve exactamente igual en Unix, MacOS y Windows. Lo que distingue a Morphic de la mayoria de los otros toolkits de interface de usuario es que no tiene modos separados para "composición' y "ejecución" de la interface: todos los elementos gráficos pueden ser ensamblados y desensamblados por el usuario en cualquier momento.[1]

11.1 La historia de Morphic

Morphic fue desarrollado por John Maloney and Randy Smith para el lenguaje de programación Self , allá por 1993. Posteriormente Maloney escribió una nueva versión de Morphic para Squeak, pero las ideas básicas detras de la versión de Self son aún vigentes en el Morphic de Pharo: *directness* and *liveness*. Directness significa que las formas en la pantalla son objetos que pueden ser examinados o modificados directamente apuntandolos con el mouse. Liveness significa que la interface de usuario siempre es capaz de responder a las acciones del usuario: la información en la pantalla es continuamente actualizada a medida que cambia el mundo que esta describe. Un simple ejemplo de esto es que uno puede desacoplar un ítem de menú item y mantenerlo como un botón.

 Abre el menú World. Meta-click para traer su halo morphic[2]*, luego meta-click*

[1] Le agradecemos a Hilaire Fernandes por permitir basar este capítulo en su artículo original en francés.

[2] Recuerda que debes setear halosEnabled en el explorador de Preferencias. Alternativamente, puedes evaluar Preferences enable: #halosEnabled en un workspace.

otra vez el el ítem de menú que deseas desacoplar para traer su halo. Ahora arrastra el ítem a algún otro lugar de la pantalla agarrándolo del manejador negro , como se muestra en Figure 11.1.

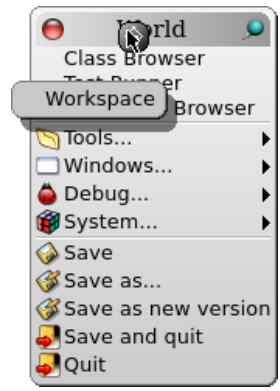


Figure 11.1: Desacoplando un morph, aquí el ítem de menú `Workspace`, para hacerlo un botón independiente.

Todos los objetos que ves en la pantalla cuando ejecutas Pharo son *Morphs*, eso es, son instancias de subclases de la clase `Morph`. `Morph` en si mismo es una clase grande con muchos métodos; esto hace posible para las subclases implementar comportamiento interesante con poco código. Tu puedes crear un morph para representar cualquier objeto, aunque cuan buena sea esa representanción, depende del objeto!

Para crear un morph que represente a un objeto cadena de caracteres, ejecuta el siguiente código en un workspace.

```
'Morph' asMorph openInWorld
```

Esto crea un Morph para representar la cadena de caracteres `'Morph'`, y luego lo abre (esto es, lo muestra) en el "mundo", que es el nombre que Pharo le da a la pantalla. Deberias obtener un elemento gráfico — un Morph — al que podrias manipular con meta-click.

Por supuesto, que es posible definir morphs que sean representaciones gráficas más interesantes que la que hemos visto. El método `asMorph` tiene una implementación por defecto en la clase `Object` que simplemente crea un StringMorph. Entonces, por ejemplo, `Color tan asMorph` devuelve un StringMorph etiquetado con el resultado de `Color tan printString`. Cambiemos esto para obtener un rectángulo coloreado en su lugar.

Abre un browser en la clase `Color` *y agrega el siguiente método:*

Method 11.1: *Getting a morph for an instance of* Color.

```
Color»asMorph
  ↑ Morph new color: self
```

Ahora ejecuta Color orange asMorph openInWorld en un workspace. En lugar de morph tipo string, obtienes ¡un rectángulo naranja!

11.2 Manipulando morphs

Los morphs son objetos, por lo tanto podemos manipularlos como cualquier otro objeto en Smalltalk: enviándoles mensajes, podemos cambiar sus propiedades, crear nuevas subclases de Morph y demás.

Cada morph, incluso si no está actualmente abierto en la pantalla, tiene una posición y un tamaño. Para conveniencia, todo morph es considerado ocupando un región rectangular de la pantalla; si son de forma irregular, su posición y tamaño son aquellos de la "caja" rectangular de menor tamaño que los rodea, que es conocida como caja de delimitación del morph, o simplemente su "delimitador".

El método position devuelve un Point que describe la ubicación de la esquina superior izquierda del morph (o la esquina superior izquierda de su caja delimitadora). El origen del sistema de coordenadas es la esquina superior izquierda de la pantalla, con la coordenada y incrementándose hacia abajo en la pantalla y con las coordenada x incrementándose hacia la derecha.

El método extent también devuelve un punto, pero este punto especifica el ancho y alto del morph en vez de su posición.

 *Tipea lo siguiente en un workspace y* do it*:*

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red .
bill openInWorld.
```

Luego tipea joe position y print it. Para mover joe, ejecuta joe position: (joe position + (10@3)) repetidamente.

Es posible hacer algo similar con el tamaño.

joe extent devuelve el tamanio de joe; para agrandar a joe, ejecuta joe extent: (joe extent * 1.1). Para cambiar el color de un morph, envíale el mensaje color: con el objecto Color deseado como argumento, por ejemplo, joe color: Color orange. Para agregar transparencia, prueba joe color: (Color orange alpha: 0.5).

Para hacer que bill siga a joe, puedes ejecutar repetidamente este código:

```
bill position: (joe position + (100@0))
```

Si mueves a joe usando el ratón y luego ejecutas este código, bill se moverá de manera de colocarse a 100 pixels a la derecha de joe. .

11.3 Componiendo morphs

Una forma de crear nuevas representaciones gráficas es colocando un morph dentro de otro. Esto es llamado *composición*; los morphs pueden ser compuesto en cualquier nivel de profundidad. Puedes colocar un morph dentro de otro enviando en mensaje addMorph: al morph contenedor.

Prueba agregar un morph dentro de otro:

```
star := StarMorph new color: Color yellow.
joe addMorph: star.
star position: joe position.
```

La última línea posiciona la estrella en las mismas coordenadas que joe. Nota que las coorodenadas del morph contenido son aún relativas a la pantalla, no al morph contenedor.

Hay muchos métodos disponibles para posicionar un morph, navega el protocolo *geometry* de la clase Morph para verlos por ti mismo. Por ejemplo, para centrar la estrella dentro de joe, ejecuta star center: joe center.

Figure 11.2: La estrella es contenida dentro de joe, el morph azul translúcido.

Si ahora intentas agarrar la estrella con el ratón, encontrarás que en realidad agarras a joe, y los dos morphs se mueven juntos: la estrella está *embebida* en joe. Es posible embeber más morphs dentro de joe. Además de poder hacer esto programáticamente, también puedes embeber morphs manipulandolos directamente.

11.4 Creando y dibujando tus propios morphs

Mientras que es posible hacer muchas representaciones gráficas útilies e interesantes componiendo morphs, en ocasiones necesitarás crear algo completamente diferente. Para hacer esto define una subclase de Morph y sobreescribe el método drawOn: para cambiar su apariencia.

El framework morphic envia el mensaje drawOn: a un morph cuando necesita volver a mostrar el morph en la pantalla. El parámetro pasado a drawOn: es de tipo Canvas; el comportamiento esperado es que el morph se dibuje en ese canvas, dentro de sus límites.

Usemos este conocimiento para crear un morph con forma de cruz.

Usando el browser, define una nueva clase CrossMorph *heredando de* Morph*:*

Class 11.2: *Definiendo* CrossMorph

```
Morph subclass: #CrossMorph
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE−Morphic'
```

Podemos definir el método drawOn: de esta forma:

Method 11.3: *Dibujando un* CrossMorph.

```
drawOn: aCanvas
   | crossHeight crossWidth horizontalBar verticalBar |
   crossHeight := self height / 3.0 .
   crossWidth := self width / 3.0 .
   horizontalBar := self bounds insetBy: 0 @ crossHeight.
   verticalBar := self bounds insetBy: crossWidth @ 0.
   aCanvas fillRectangle: horizontalBar color: self color.
   aCanvas fillRectangle: verticalBar color: self color
```

Enviando el mensaje bounds a un morph, este contesta su caja delimitadora, que es una instancia de Rectangle. Los rectángulos entiende muchos mensajes que crean otros rectángulos de geometria relacionada; aqui usamos el mensaje insetBy: con un punto como argumento para crear primero un rectángulo con altura reducida, y luego otro con archo reducido.

Para probar tu nuevo morph, ejecuta CrossMorph new openInWorld.

El resultado debería verse similar a la Figure 11.3. De todas formas, notarás que la zona sentiva — donde puedes hacer click para agarrar el morph — es aún la caja delimitadora completa. Arreglemos esto.

Figure 11.3: Un CrossMorph con su halo; puedes cambiarle el tamaño a tu gusto.

Cuando el framework Morphic necesita encontrar qué morphs estan bajo el cursor, envía el mensaje containsPoint: a todos los morphs cuyas cajas delimitadoras se encuentran bajo el puntero del ratón. Entonces, para limitar la zona sensitiva del morph a la forma de la cruz, necesitamos sobreescribir el método containsPoint:.

Define el siguiente método en la clase CrossMorph*:*

Method 11.4: *Ajustando la zona sensitiva de* CrossMorph.

```
containsPoint: aPoint
   | crossHeight crossWidth horizontalBar verticalBar |
   crossHeight := self height / 3.0.
   crossWidth := self width / 3.0.
   horizontalBar := self bounds insetBy: 0 @ crossHeight.
   verticalBar := self bounds insetBy: crossWidth @ 0.
   ↑ (horizontalBar containsPoint: aPoint)
      or: [verticalBar containsPoint: aPoint]
```

Este método usa la misma lógica que drawOn:, entonces podemos estar seguros que los puntos para los cuales containsPoint: contesta true son los mismos que serán coloreados por drawOn. Nota como aprovechamos el método containsPoint: en la clase Rectangle para hacer el trabajo difícil.

Hay dos problemas con el código en los métodos 11.3 y 11.4. El más obvio es que hemos duplicado código. Este es un error cardinal: si encontramos que necesitamos cambiar la forma en que horizonatalBar o verticalBar son calculadas, podriamos olvidarnos de cambiar una de las dos ocurrencias. La solución es factorizar estos cálculos en dos nuevos métodos, que pondremos en el protocolo private:

Method 11.5: horizontalBar.

```
horizontalBar
   | crossHeight |
```

```
    crossHeight := self height / 3.0.
    ↑ self bounds insetBy: 0 @ crossHeight
```

Method 11.6: verticalBar.

```
verticalBar
    | crossWidth |
    crossWidth := self width / 3.0.
    ↑ self bounds insetBy: crossWidth @ 0
```

Podemos entonces definir ambos drawOn: y containsPoint: usando estos métodos:

Method 11.7: *Refactored* CrossMorph»drawOn:.

```
drawOn: aCanvas
    aCanvas fillRectangle: self horizontalBar color: self color.
    aCanvas fillRectangle: self verticalBar color: self color
```

r

Method 11.8: *Refactored* CrossMorph»containsPoint:.

```
containsPoint: aPoint
    ↑ (self horizontalBar containsPoint: aPoint)
        or: [self verticalBar containsPoint: aPoint]
```

Este código es mucho más simple de entender, en gran medida porque hemos dado nombres con significado a los métodos privados. De hecho, es tanto más simple que puedes haber notado el segundo problema: el área en el centro de la cruz, que está bajo la barra horizontal y la barra vertical, es dibujada dos veces. Esto no importa cuando llenamos la cruz con un color opaco, pero el defecto se vuelve evidente inmediatable si dibujamos una cruz semitransparente, como se muestra en la Figure 11.4.

Ejecuta el siguiente código en un workspace, línea por línea:

```
m := CrossMorph new bounds: (0@0 corner: 300@300).
m openInWorld.
m color: (Color blue alpha: 0.3).
```

La solución a esto es dividir la barra vertical en 3 partes, y llenar solo la parte superior y la inferior. Una vez más encontramos un método en la clase Rectangle que hace el trabajo difícil por nosotros: r1 areasOutside: r2 devuelve un arreglo de rectángulos con las partes de r1 fuera de r2. Aquí está el código revisado:

Method 11.9: *El método revisado* drawOn:, *que llena el centro de la una vez.*

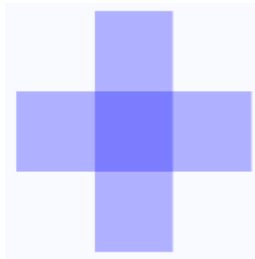


Figure 11.4: El centro de la cruz es llenado dos veces con el color.

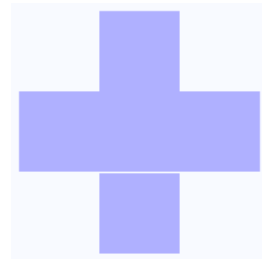


Figure 11.5: El morph con forma de cruz, mostrando una fila de pixeles sin llenar.

```
drawOn: aCanvas
    | topAndBottom |
    aCanvas fillRectangle: self horizontalBar color: self color.
    topAndBottom := self verticalBar areasOutside: self horizontalBar.
    topAndBottom do: [ :each | aCanvas fillRectangle: each color: self color]
```

Este código parece funcionar, pero si lo pruebas con algunas cruces y le cambias el tamaño, notarás que en ciertos tamaños, una línea de un pixel separa la parte inferior de la cruz, del resto, como se muestra en Figure 11.5. Esto es debido al redondeo: cuando el tamaño del rectángulo a ser llenado no es un entero, fillRectangle: color: parece redondear consistentemente, dejando una fila de pixeles sin llenar. Podemos trabajar sobre esto redondeando explícitamente cuando calculamos los tamanõs de las barras.

Method 11.10: CrossMorph»horizontalBar *con redondeo explícito.*

```
horizontalBar
    | crossHeight |
    crossHeight := (self height / 3.0) rounded.
    ↑ self bounds insetBy: 0 @ crossHeight
```

Method 11.11: CrossMorph»verticalBar *con redondeo explícito.*

```
verticalBar
    | crossWidth |
    crossWidth := (self width / 3.0) rounded.
    ↑ self bounds insetBy: crossWidth @ 0
```

11.5 Interacción y animación

Para construir interfaces de usuario vivas usando morphs, necesitamos poder interactuar con ellos usando el ratón y el teclado. Más aún, los morphs

necesitan ser capaces de responder a la entrada del usuario cambiando su apariencia y posición — esto es, animándose asimismos.

Eventos del ratón

Cuando un botón del ratón es presionado, Morphic envía el mensaje mouseDown: a cada morph ubicado bajo el puntero del ratón y también envia el mensaje mouseUp: cuando el usuario libera el botón del ratón. Si todos los morphs contestan false, entonces Morphic inicia una operación drap-and-drop. Como discutiremos más abajo, los mensajes mouseDown: y mouseUp: son enviados con el argumento — a MouseEvent — que codifica los detalles de la acción del ratón.

Extendamos el CrossMorph para manejar eventos del ratón, Comenzamos por asegurarnos que todos los crossMorphs contestan true al mensajehandlesMouseDown:.

Agrega este método a CrossMorph*:*

Method 11.12: *Declarando que* CrossMorph *reaccionará clics del ratón.*

```
CrossMorph»handlesMouseDown: anEvent
  ↑true
```

Supongamos que cuando hacemos clic en la cruz, queremos cambiar el color de la cruz a rojo, y que cuando action-click en esta, queremos cambiar el color a amarillo. Esto se puede hacer con method 11.13.

Method 11.13: *Reaccionando a clics del ratón para cambiar el color del morph.*

```
CrossMorph»mouseDown: anEvent
  anEvent redButtonPressed "click"
    ifTrue: [self color: Color red].
  anEvent yellowButtonPressed "action−click"
    ifTrue: [self color: Color yellow].
  self changed
```

Nota que adicionalmente al cambiar el color de un morph, este método también envia self changed. Esto asegura que el morphic envía drawOn: de manera oportuna.

Nota también que una vez que el morph maneja los eventos del ratón, ya no puedes agarrarlo con el ratón y moverlo. En lugar de ello tienes que utilizar el halo: meta-click en el morph para hacer aparecer el halo y agarrar el manejador marrón o el manejador negro en la parte superior del morph,

El argumento anEvent de mouseDown: es una instancia de MouseEvent, que es una subclase de MorphicEvent. MouseEvent define los métodos

redButtonPressed y yellowButtonPressed. Navega esta clase para ver qué otros métodos provee para interrogar los eventos del ratón.

Eventos del teclado

Para atrapar eventos del teclado, necesitamos hacer tres pasos.

1. Dar el "foco de teclado" a un morph específico: por ejemplo podemos dar el foco a nuestro morph cuando el ratón se mueve sobre él.

2. Manejar el evento del teclado con el handleKeystroke: method: este mensaje es enviado al morph que tiene el foco de teclado cuando el usuario presiona una tecla.

3. Liberar el foco de teclado cuando el ratón ya no está sobre nuestro morph.

Extendamos el CrossMorph para que reaccione a las pulsaciones de teclas. Primero necesitamos ser notificados cuando el ratón está sobre el morph. Esto sucederá si nuestro morph responde true al mensaje handlesMouseOver:

Declara que CrossMorph *reaccionará cuando este bajo el puntero del ratón.*

Method 11.14: *Queremos manejar los eventos de "mouse over".*

```
CrossMorph»handlesMouseOver: anEvent
  ↑true
```

Este mensaje es el equivalente a handlesMouseDown: para la posición del ratón. Cuando el puntero del ratón entra o sale del morph, los mensajes mouseEnter: y mouseLeave: son enviados a este.

Define dos métodos para que el CrossMorph *atrape y libere el foco del teclado, y un tercer método para manejar las pulsaciones de teclado.*

Method 11.15: *Obteniendo el foco de teclado cuando el ratón entra al morph.*

```
CrossMorph»mouseEnter: anEvent
  anEvent hand newKeyboardFocus: self
```

Method 11.16: *Devolviendo el foco cuando el puntero se va.*

```
CrossMorph»mouseLeave: anEvent
  anEvent hand newKeyboardFocus: nil
```

Method 11.17: *Recibiendo y manejando eventos de teclado.*

```
CrossMorph»handleKeystroke: anEvent
  | keyValue |
  keyValue := anEvent keyValue.
  keyValue = 30    "up arrow"
    ifTrue: [self position: self position − (0 @ 1)].
  keyValue = 31    "down arrow"
    ifTrue: [self position: self position + (0 @ 1)].
  keyValue = 29    "right arrow"
    ifTrue: [self position: self position + (1 @ 0)].
  keyValue = 28    "left arrow"
    ifTrue: [self position: self position − (1 @ 0)]
```

Hemos escrito este método para poder mover el morph usando las teclas de flechas. Nota que cuando el ratón ya no está sobre el morph, el mensaje handleKeystroke: no es enviado, por lo que el morph deja de responder a los comandos del teclado. Para descubrir los valores de las teclas, puedes abrir un Transcript y agregar Transcript show: anEvent keyValue al método method 11.17.

El argumento anEvent de handleKeystroke: es una instancia de KeyboardEvent, otra subclase de MorphicEvent. Navega esta clase para aprender más sobre los eventos de teclado.

Animaciones Morphic

Morphic provee un sistema simple de animación con dos métodos: step es enviado a un morph en intervalos regulares de tiempo, mientras que stepTime especifica el tiempo en milisegundos entre steps.[3]

Si preguntas por un stepTime de 1 ms, no te sorprendas si Pharo está muy ocupado para step tu morph tan seguido.

In addition, startStepping turns on the stepping mechanism, while stopStepping turns it off again; isStepping can be used to find out whether a morph is currently being stepped.

Make CrossMorph *blink by defining these methods as follows:*

Method 11.18: *Defining the animation time interval.*

```
CrossMorph»stepTime
  ↑ 100
```

Method 11.19: *Making a step in the animation.*

```
CrossMorph»step
```

[3] stepTime es en realidad el *mínimo* tiempo entre steps.

```
(self color diff: Color black) < 0.1
    ifTrue: [self color: Color red]
    ifFalse: [self color: self color darker]
```

Para comenzar, puedes abrir un inspector en un CrossMorph (usando el manejador de debug en una halo de morphic), tipea self startStepping en el pequeño workspace de abajo, y do it. Alternativamente, puedes modificar el médtodo handleKeystroke: para que puedas usar las teclas + and − para comenzar y deterner el paso a paso.

Agregar el siguiente código a method 11.17:

```
keyValue = $+ asciiValue
    ifTrue: [self startStepping].
keyValue = $− asciiValue
    ifTrue: [self stopStepping].
```

11.6 Interactores

Para pedirle al usuario una entrada, la clase UIManager provee una gran número de cajas de diálogo listas para usar. Por ejemplo, el método request:initialAnswer: revuelve un cadena ingresada por le usuario (Figure 11.6).

```
UIManager default request: 'What''s your name?' initialAnswer: 'no name'
```

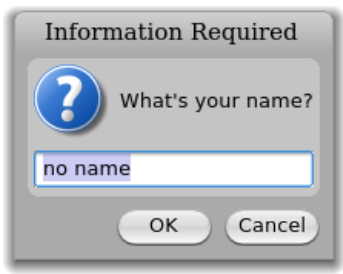


Figure 11.6: Un diálogo de entrada.

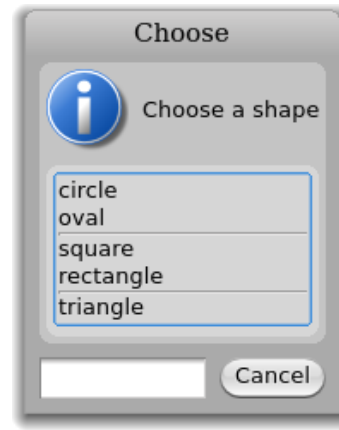


Figure 11.7: Menú emergente.

Para desplegar un menú emergente, usa uno de los varios métodos chooseFrom: (Figure 11.7):

```
UIManager default
  chooseFrom: #('circle' 'oval' 'square' 'rectangle' 'triangle')
  lines: #(2 4) message: 'Choose a shape'
```

Nageva la clase UIManager *y prueba algunos de los métodos de interacción ofrecidos.*

11.7 Drag-And-Drop

Morphic también soporta drag-and-drop. Examinemos un simple ejemplo con dos morphs, un morph receptor y un morph arrojado. El receptor aceptará un morph solo si el morph arrastrado coincide con una condición dada: en nuestro ejemplo, el morph debe ser azul. Si es rechazado el morph arrastrado decide que hacer.

Primero definamos el morph receptor:

Class 11.20: *Definiendo un morph en el que podamos arrastrar otros morphs*

```
Morph subclass: #ReceiverMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−Morphic'
```

Ahora define el método de inicialización de la manera usual:

Method 11.21: *Inicializando* ReceiverMorph.

```
ReceiverMorph»initialize
  super initialize.
  color := Color red.
  bounds := 0 @ 0 extent: 200 @ 200
```

¿Como dedicir si el receptor aceptará o rechazará el morph arrastrado? En general ambos morphs tendrán que acordar en la interacción.

El receptor hace esto respondiendo a wantsDroppedMorph:event:; el primer argumento es el morph arrastrado, y el segundo es el evento del ratón, de esta forma el receptor puede por ejemplo ver si alguna tecla modificadora fue presionada al momento del arrastre. El morph arrastrado también tiene la oportunidad de chequear y ver si le gusta el morph sobre el que

está siendo arrojado; se le envía el mensaje wantsToBeDroppedInto:. La implementación por defecto de este método (en la clase Morph) responde true.

Method 11.22: *Aceptar el morph arrojado en base a su color.*

```
ReceiverMorph»wantsDroppedMorph: aMorph event: anEvent
  ↑ aMorph color = Color blue
```

Que ocurre con el morph arrojado si el morph receptor no lo quiere? El comportamiento por defecto es no hacer nada, eso es, sentarse encima del morph receptor, pero sin interactuar con el. Un comportamiento más intuitivo es que el morph arrojado vuelva a su posición original. Esto puede lograrse en el receptor, contestando true al mensaje repelsMorph:event: cuando no quiere al morph arrojado:

Method 11.23: *Changing the behaviour of the dropped morph when it is rejected.*

```
ReceiverMorph»repelsMorph: aMorph event: ev
  ↑ (self wantsDroppedMorph: aMorph event: ev) not
```

Eso es todo lo que necesitamos en lo que respecta al receptor.

Crea instancias de ReceiverMorph *y* EllipseMorph *en un workspace:*

```
ReceiverMorph new openInWorld.
EllipseMorph new openInWorld.
```

Intenta hacer drag-and-drop el EllipseMorph amarillo sobre el receptor. Este será rechazado y enviado de vuelta a su posición original.

Para cambiar este comportamiento, cambia el color de la morph elipse a Color blue *usando un inspector. Los morphs azules deben ser aceptado por el* ReceiverMorph .

Creemos una subclase de Morph, llamada DroppedMorph, para que podamos experimentar un poco más:

Class 11.24: *Definiendo un morph que podamos arrastrar y soltar sobre el* ReceiverMorph

```
Morph subclass: #DroppedMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−Morphic'
```

Method 11.25: *Inicializando* DroppedMorph.

```
DroppedMorph»initialize
  super initialize.
  color := Color blue.
  self position: 250@100
```

Ahora podemos especificar qué debe hacer el morph arrastrado cuando es rechazado por el receptor; aqui este permanecerá adjunto al puntero del ratón:

Method 11.26: *Reaccionando cuando el morph fue arrojado pero rechazado.*

```
DroppedMorph»rejectDropMorphEvent: anEvent
  | h |
  h := anEvent hand.
  WorldState
    addDeferredUIMessage: [h grabMorph: self].
  anEvent wasHandled: true
```

Enviar el mensaje hand a un evento, responde una *hand*, que es una instancia de HandMorph que representa el puntero del ratón y lo que sea que sostenga.

Aqui le decimos al World que la mano debe agarrar self, el morph rechazado.

 *Crea dos intancias de* DroppedMorph*, y luego drag-and-drop sobre el receptor.*

```
ReceiverMorph new openInWorld.
(DroppedMorph new color: Color blue) openInWorld.
(DroppedMorph new color: Color green) openInWorld.
```

El morph verde es rechazado y por lo tanto se mantiene adjunto al puntero del ratón.

11.8 Un ejmeplo completo

Deseñemos un morph para tirar un dado. Haciendo clic en el, se mostrarán los valores de todas sus caras en un ciclo, y otro clic detendrá la animación.

 *Define un dado como una subclase de* BorderedMorph *en lugar de* Morph*, porque haremos uso del borde.*

Figure 11.8: El dado en Morphic.

Class 11.27: *Definiendo un dado morph*

```
BorderedMorph subclass: #DieMorph
   instanceVariableNames: 'faces dieValue isStopped'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE−Morphic'
```

La variable de instancia faces registra el número de caras del dado; ¡nos permitirá dados con hasta 9 caras! dieValue registra los valores de la cara que está actualmente mostrada, y isStopped es verdadero si la animación está detenida.

Para crear una instancia de dado, definimos el método caras en el lado *class* de DieMorph para crear un nuevo dado con n caras.

Method 11.28: *Creado un new dado con el número de caras que nos gusta.*

```
DieMorph class»faces: aNumber
   ↑ self new faces: aNumber
```

El método initialize está definido en la instancia como de costumbre; recuerda que new envía initialize a la nueva instancia creada.

Method 11.29: *Inicializando instancias de* DieMorph.

```
DieMorph»initialize
   super initialize.
   self extent: 50 @ 50.
   self useGradientFill; borderWidth: 2; useRoundedCorners.
   self setBorderStyle: #complexRaised.
   self fillStyle direction: self extent.
   self color: Color green.
   dieValue := 1.
   faces := 6.
   isStopped := false
```

Usamos unos pocos métodos de BorderedMorph para dar una apariencia agradable al dado: un borde grueso con una efecto elevado, esquinas redondeadas, y un color gradiente en la cara visible.

Definimos el método de instancia faces: para chequear parámetros válidos como sigue:

Method 11.30: *Seteando el número de caras del dado.*

```
DieMorph»faces: aNumber
   "Set the number of faces"
   (aNumber isInteger
         and: [aNumber > 0]
         and: [aNumber <= 9])
      ifTrue: [faces := aNumber]
```

Podría ser bueno reveer el orden en que se envian los mensajes cuando un dado es creado. Por ejemplo, si comenzamos por evaluar DieMorph faces: 9:

1. El método de clase DieMorph class»faces: envia new a DieMorph class.
2. El método para new (heredado por DieMorph class de Behavior) crea la nueva instacia y le envía el mensaje initialize.
3. El método initialize en DieMorph establece faces al valor inicial 6.
4. DieMorph class»new retorna el método de clase DieMorph class»faces:, que envía el mensaje faces: 9 a la nueva instancia.
5. El método de instancia DieMorph»faces: ahora se ejecuta, estableciendo la variable de instancia faces en 9.

Antes de definir drawOn:, necesitamos unos pocos métodos para poner los puntos en la cara mostrada:

Methods 11.31: *Nueve métodos para ubicar los puntos en las caras del dado.*

```
DieMorph»face1
   ↑{0.5@0.5}
DieMorph»face2
   ↑{0.25@0.25 . 0.75@0.75}
DieMorph»face3
   ↑{0.25@0.25 . 0.75@0.75 . 0.5@0.5}
DieMorph»face4
   ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}
DieMorph»face5
   ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}
DieMorph»face6
   ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5}
DieMorph»face7
   ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5}

DieMorph »face8
   ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5
       . 0.5@0.25}
```

```
DieMorph »face9
  ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5
     . 0.5@0.25 . 0.5@0.75}
```

Estos métodos definen colecciones de coordenadas de puntos para cada cara. Las coordenadas estan en un cuadrado de tamaño 1×1; simplemente necesitaremos escalarlas para ubicar los verdaderos puntos.

El método drawOn: hace dos cosas: dibuja el fondo del dado con super-send, y luego dibuja los puntos.

Method 11.32: *Dibujando el dado morph.*

```
DieMorph»drawOn: aCanvas
  super drawOn: aCanvas.
  (self perform: ('face' , dieValue asString) asSymbol)
    do: [:aPoint | self drawDotOn: aCanvas at: aPoint]
```

La segunda parte de este método usa las capacidades refletivas de Smalltalk. Dibujar los puntos de una cara es una simple cuestión de iterar sobre la colección dada por el método faceX para esa cara, enviando el mensaje drawDotOn:at: para cada coordenada. Para llamar al correcto m'etodo faceX , usamos el método perform: que envia un mensaje a partir de una cadena, aquí ('face', dieValue asString) asSymbol. Encotrarás este uso de perform: bastante seguido.

Method 11.33: *Dibujando un punto en una cara.*

```
DieMorph»drawDotOn: aCanvas at: aPoint
  aCanvas
    fillOval: (Rectangle
      center: self position + (self extent * aPoint)
      extent: self extent / 6)
    color: Color black
```

Como las coordenadas están normalizadas al intervalo $[0{:}1]$, podemos escalarlas a las dimensiones de nuestro dado: self extent * aPoint.

Ya podemos crear una instancia de dado desde un workspace:

```
(DieMorph faces: 6) openInWorld.
```

Para cambiar la cara mostrada, creamos un accesor que podemos usar como myDie dieValue: 4:

Method 11.34: *Estableciendo el valor del dado.*

```
DieMorph»dieValue: aNumber
  (aNumber isInteger
      and: [aNumber > 0]
```

```
        and: [aNumber <= faces])
    ifTrue:
        [dieValue := aNumber.
        self changed]
```

Ahora vamos usar el sistema de animación para mostrar rápidamente todas las caras:

Methods 11.35: *Animando el dado.*

```
DieMorph»stepTime
    ↑ 100

DieMorph»step
    isStopped ifFalse: [self dieValue: (1 to: faces) atRandom]
```

¡Ahora el dado está rodando!

Para comenzar o deterner la animación haciendo clic, usaremos lo que hemos aprendido previamente sobre los eventos del ratón. Primero, activamos la recepción de los eventos del ratón:

Methods 11.36: *Manejando clics del ratón para iniciar y detenar la animación.*

```
DieMorph»handlesMouseDown: anEvent
    ↑ true

DieMorph»mouseDown: anEvent
    anEvent redButtonPressed
        ifTrue: [isStopped := isStopped not]
```

Ahora el dado rodará o se detendrá hagamos clic en él.

11.9 Más acerca del canvas

El método drawOn: tiene una instancia de Canvas como su único argumento; el canvas es un área en donde el morph se dibuja asimismo. Usando los métodos gráficos del canvas, eres libre de dar la apariencia que quieras a un morph. Si navegas la jerarquía de herencia de la clase Canvas, verás que tiene varias variantes.

La variante por defecto de Canvas es FormCanvas; encontrarás los métodos gráficos clave en Canvas y FormCanvas. Estos métodos pueden dibujar puntos, líneas, polígonos, rectángulos, elipses, textos e imágenes con rotación y escalamiento.

También es posible usar otras clases de canvas, para obtener morphs transparentes, más métodos gráficos, antialising, y otros.

Para usar estas funcionalidades necesitarás un AlphaBlendingCanvas o un BalloonCanvas.

Pero, ¿como puedes obtener un canvas así en un método drawOn: cuando drawOn: recibe una instancia de FormCanvas como argumento? Afortunadamente, puedes transformar un tipo de canvas en otro.

Para usar un canvas con alpha-transparency 0.5 en DieMorph, *redefine* drawOn: *de esta forma:*

Method 11.37: *Dibujando un dado translucido.*

```
DieMorph»drawOn: aCanvas
   | theCanvas |
   theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.
   super drawOn: theCanvas.
   (self perform: ('face' , dieValue asString) asSymbol)
      do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

¡Eso es todo lo que necesitas hacer!

Figure 11.9: El dado mostrado con alpha-transparency.

11.10 Resumen del capítulo

Morphic es un marco de trabajo gráfico en que los elementos de la interface gráfica pueden ser compuestos dinámicamente.

- Puedes convertir un objeto en un morph y mostrarlo en la pantalla enviándole los mensajes asMorph openInWorld.
- Puedes manipular a un morph meta-clicking sobre el y usando los manejadores que aparecen. (Los manejadores tiene balones de ayuda que explican lo que hacen.)
- Puedes componer morphs embebiéndolos uno dentro de otro, o arrastrándo y soltando o enviando el mensaje addMorph:.

- Puedes subclasear un clase morph existente y redefinir los métodos clave, como initialize y drawOn:.
- Puedes controlar como un morph reacciona a los eventos de ratón y teclado redefiniendo los métodos handlesMouseDown:, handlesMouseOver:, etc.
- Puedes animar un morph definiendo los métodos step (qué hacer) y stepTime (el número de milisegundos entre pasos)
- Varios morphs predefinidos, como PopUpMenu y FillInTheBlank, están disponibles para interactuar con los usuarios.

Chapter 12

Seaside en Ejemplos

Seaside es un marco de trabajo para crear aplicaciones web en Smalltalk. Fue desarrollado originalmente por Avi Bryant en 2002; una vez dominado, Seaside hace las aplicaciones web tan fáciles de escribir como las aplicaciones de escritorio.

Dos de las aplicaciones creadas con Seaside más conocidas son SqueakSource[1] y Dabble DB[2]. Seaside es poco común ya que está rigurosamente orientado a objetos: no hay plantillas XHTML, ni complicados controles de flujo a través de páginas web, ni codificación del estado en las URLs. En su lugar, sólo envías mensajes a los objetos. ¡Qué buena idea!

12.1 ¿Por qué necesitamos Seaside?

Las aplicaciones web modernas tratan de interactuar con el usuario de la misma manera que las aplicaciones de escritorio: realizan al usuario preguntas y el usuario responde, normalmente rellenando un formulario o pulsando un botón. Pero la web funciona en sentido inverso: el navegador del usuario realiza una petición al servidor y el servidor responde con una nueva página web.

Por lo tanto, los marcos de desarrollo de aplicaciones web tienen que tratar con un montón de problemas, siendo el principal de ellos la gestión de este control de flujo "invertido". Por esto, muchas aplicaciones intentan prohibir el uso del botón "volver atrás" del navegador, debido a la dificultad de mantener el seguimiento del estado de una sesión. Expresar un control de flujo no trivial a través de varias páginas normalmente es engorroso y varios flujos de control pueden ser difíciles o imposibles de expresar.

[1] http://SqueakSource.com
[2] http://DabbleDB.com

Seaside es un marco de trabajo basado en componentes que hace el desarrollo web más fácil en varias formas. Primero, el control de flujo puede expresarse de forma natural usando el envío de mensajes. Seaside realiza el seguimiento de la página web que corresponde con cada punto de ejecución de la aplicación web. Esto significa que el botón "volver atrás" del navegador funciona correctamente.

Segundo, gestiona el estado en tu lugar. Como el desarrollador, puedes escoger habilitar el seguimiento del estado, para que la navegación "hacia atrás" en el tiempo desahaga los efectos laterales. Si no, puedes usar el soporte de transacciones incluído en Seaside para evitar que los usuarios deshagan los efectos laterales permanentes cuando usen el botón volver atrás. No tienes que codificar la información de estado en la URL — esto también es gestionado automáticamente por ti.

Tercero, las páginas web son construídas a partir de componentes anidados, cada uno de los cuales soporta su propio control de flujo independiente. No hay plantillas XHTML — sino que se genera XHTML válido programadamente, usando un sencillo protocolo Smalltalk. Seaside soporta Hojas de Estilo en Cascada (CSS), de forma que el contenido y disposición estén claramente separados.

Finalmente, Seaside proporciona una práctica interfaz de desarrollo basada en web, lo que hace fácil desarrollar aplicaciones iterativamente, depurar aplicaciones interactivamente y recompilar y extender aplicaciones miembras el servidor está ejecutándose.

12.2 Para empezar

La forma más sencilla de comenzar es descargar "Seaside Experiencia en Un Click" desde el sitio web de Seaside[3]. Esta es una versión preempaquetada de Seaside 2.8 para Mac OSX, Linux y Windows. El mismo sitio web lista muchas referencias a recursos adicionales, incluyendo documentación y tutoriales. Te advertimos, sin embargo, de que Seaside ha evolucionado considerablemente con los años y no todo el material disponible se refiere a la última versión de Seaside.

Seaside incluye un servidor web; puedes iniciar el servidor, diciéndole que escuche en el puerto 8080, evaluando `WAKom startOn: 8080`, y puedes detenerlo otra vez evaluando `WAKom stop`. En la instalación por defecto, el login de administrador por defecto es `admin` y la contraseña por defecto es `seaside`. Para cambiarlos, evalúa: `WADispatcherEditor initialize`. Esto te solicitará un nuevo nombre y contraseña.

 *Inicia el servidor Seaside y dirige un navegador web a* `http://localhost:8080/`

[3] http://seaside.st

Figure 12.1: Iniciando Seaside

seaside/.

Deberías ver una página web como la de la Figure 12.1.

Navega a la página examples ▷ counter *page. (Figure 12.2)*

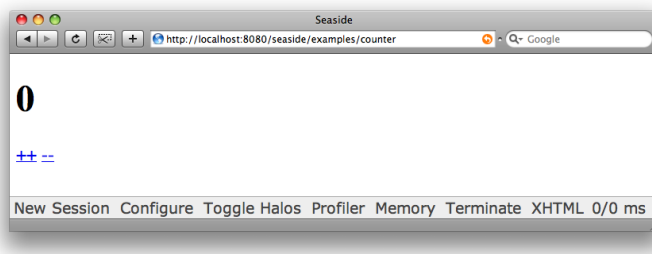


Figure 12.2: El contador.

Esta página es una pequeña aplicación Seaside: muestra un contador que puede ser incrementado o decrementado haciendo click en los enlaces ++ y --.

Juega con el contador haciendo click en estos enlaces. Usa el botón "atrás" de

tu navegador para volver a un estado previo y entonces haz cllick en ++ otra vez. Fíjate cómo el contador se incrementa correctamente con respecto al estado mostrado en ese momento, en lugar del estado en que estaba cuando usaste el botón "volver".

Fíjate en la barra de herramientas al final de la página web en la Figure 12.1. Seaside soporta una noción de "sesiones" para realizar el seguimiento del estado de la aplicación para diferentes usuarios. New Session comenzará una nueva sesión en la aplicación contador. Configure permite configurar los ajustes de tu aplicación a través de una cómoda interfaz web. (Para cerrar la vista Configure, pulsa en la x en la esquina superior derecha.) Toggle Halos proporciona una forma de explorar el estado de la aplicación ejecutándose en el servidor Seaside. Profiler y Memory proporcionan información detallada sobre el rendimiento de la aplicación en tiempo de ejecución. XHTML puede usarse para validar la página web generada, pero sólo funciona cuando la página web es públicamente accesible en Internet porque usa el servicio de validación del W3C.

Las aplicaciones Seaside se construyen a partir de "componentes" conectables. De hecho, los componentes son objetos normales de Smalltalk. Lo único que tienen de especial es que deben ser instancias de clases que heredan de la clase WAComponent del marco de trabajo Seaside. Podemos explorar los componentes y sus clases desde la imagen Pharo o directamente desde la interfaz web usando los halos.

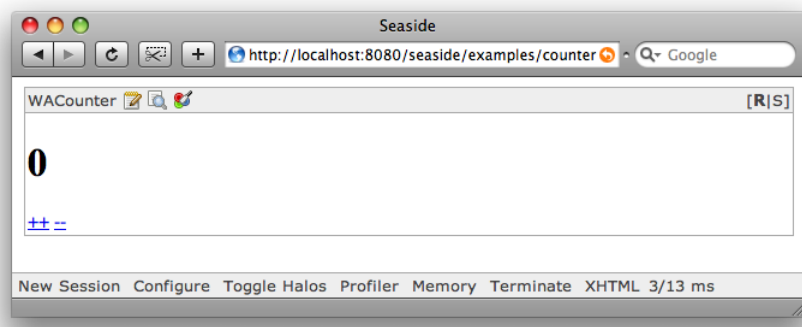


Figure 12.3: Halos

Selecciona Toggle Halos. *Deberías ver una página web como la de la Figure 12.3. En la esquina superior izquierda el texto* WACounter *indica la clase del componente Seaside que implementa el comportamiento de esta página web. Junto a este, hay tres iconos que puedes pulsar. El primero, con el lápiz, activa el navegador de clases de Seaside sobre esta clase. El segundo, con la lupa, abre un inspector de*

objetos sobre la instancia WACounter *actualmente activa. El tercero, con los circulos de color, muestra la hoja de estilos CSS para este componente. En la esquina superior derecha, la* R *y* S *te permiten conmutar entre las vistas renderizadas y fuente de la página web. Experimenta con todos estos enlaces. Ten en cuenta que los vínculos* ++ *y* – *también están activos en la vista fuente. Compara la vista fuente correctamente formateada que proporciona los Halos con la vista fuente sin formato ofrecida por tu navegador.*

El navegador de clases y el inspector de objetos de Seaside pueden ser muy útiles cuando el servidor está ejecutándose en otro ordenador, especialmente cuando el servidor no tiene pantalla o está en un lugar remoto. Sin embargo, cuando desarrollas una aplicación Seaside por vez primera, el servidor estará ejecutándose localmente y es fácil usar las herramientas de desarrollo normales de Pharo en la imagen del servidor.

Figure 12.4: Deteniendo el contador

Usando el enlace del inspector de objetos en el navegador web, abrir un inspector sobre el objeto contador de Smalltalk subyacente y evaluar self halt. *La página web detendrá la carga. Ahora cambia a la imagen de Seaside. Deberías ver una ventana de pre-depurador (Figure 12.4) mostrando un objeto* WACounter *ejecutando un* halt. *Examina la ejecución en el depurador y entonces* Proceed. *Vuelve al navegador web y fíjate en que la aplicación contador está ejecutándose otra vez.*

Los componentes de Seaside pueden instanciarse varias veces y en diferentes contextos.

Dirige tu navegador web a http://localhost:8080/seaside/examples/multicounter. *Verás una aplicación formada por un número de instancias independientes del componente contador. Incrementa y decrementa varios de los contadores. Verifica que se comportan correctamente incluso si usas el botón "atrás". Activa los halos para ver cómo la aplicación está formada por componentes anidados. Usa el navegador de clases de Seaside para ver la implementación de* WAMultiCounter. *Deberías ver tres métodos en el lado de clase (*canBeRoot, description *e* initialize*) y tres en el lado de instancia (*children, initialize *y* renderContentOn:*). Observa que una aplicación es simplemente un componente dispuesto a ser la raíz de la jerarquía de componentes;*

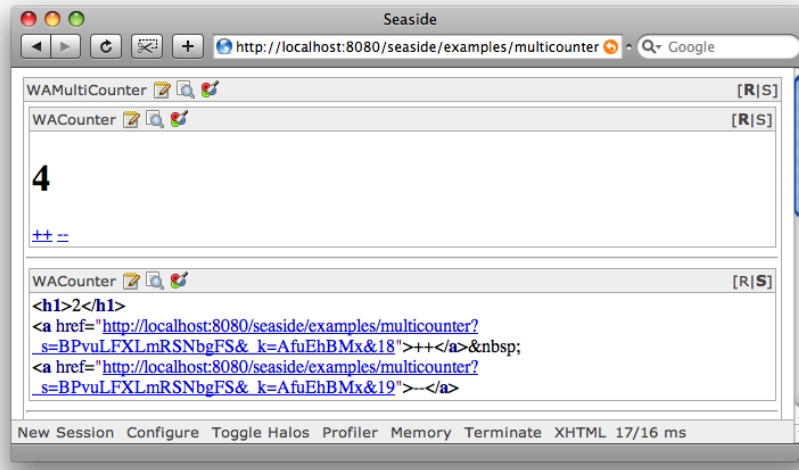


Figure 12.5: Subcomponentes independientes

esta predisposición se indica definiendo un método de clase canBeRoot *que responda* true.

Puedes usar la interfaz web de Seaside para configurar, copiar o eliminar aplicaciones individuales (es decir, componentes del nivel raíz). Inteta realizar el siguiente cambio de configuración.

Dirige tu navegador web a http://localhost:8080/seaside/config. *Introduce el login y contraseña (*admin *y* seaside *por defecto). Selecciona* Configure *junto a "examples." Bajo la cabecera "Add entry point", introduce el nuevo nombre "counter2" para el tipo* Application *y pulsa sobre* Add *(ver Figure 12.6). En la siguiente pantalla, cambia* Root Component *a* WACounter, *entonces haz click en* Save *y* Close. *Ahora tenemos un nuevo contador instalado en* http://localhost:8080/seaside/examples/counter2. *Usa la misma interfaz de configuración para eliminar este punto de entrada.*

Seaside opera en dos modos: el modo *desarrollo*, que es el que hemos visto hasta ahora, y el modo *despliegue*, en el que la barra de herramientas no está disponible. Puedes poner Seaside en el modo despliegue usando la página de configuración (navega a la entrada para la aplicación y haz click en el enlace Configure) o haciendo click en el botón Configure en la barra de herramientas. En cualquier caso, se pone el modo despliegue a *true*. Observa que esto sólo afecta a las nuevas sesiones. También puedes poner el modo globalmente evaluando WAGlobalConfiguration setDeploymentMode o

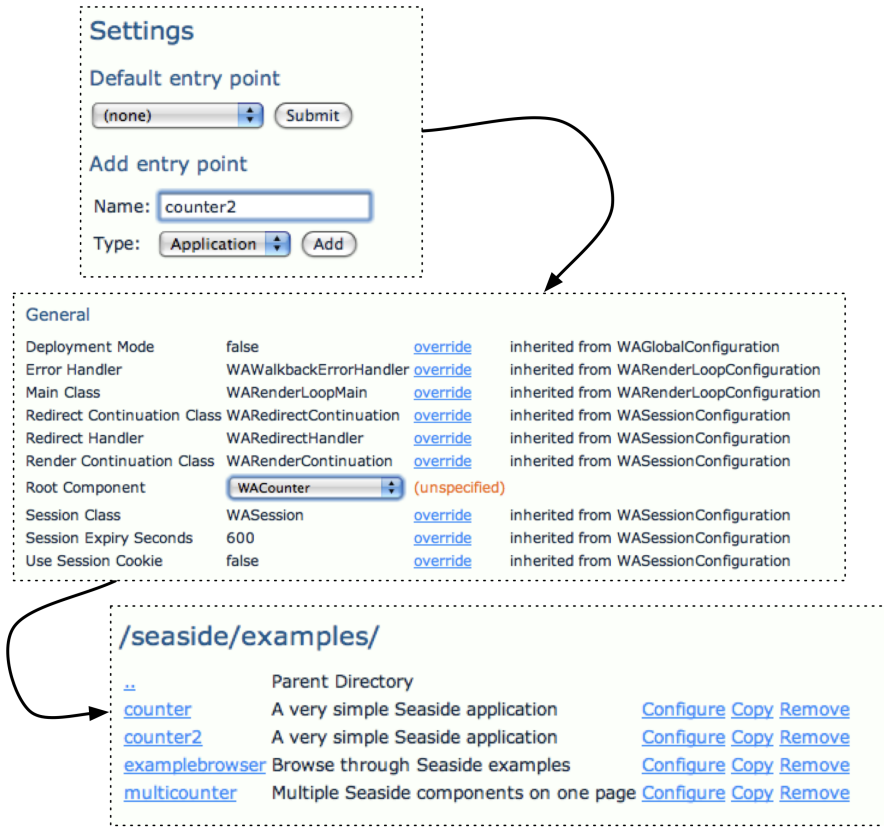


Figure 12.6: Configuración de una nueva aplicación

WAGlobalConfiguration setDevelopmentMode.

La página web de configuración sólo es otra aplicación Seaside por lo que también puede ser controlada desde la página de configuración. Si eliminas la aplicación "config", puedes recuperarla evaluando WADispatcherEditor initialize.

12.3 Componentes Seaside

Como hemos mencionado en la sección anterior, las aplicaciones Seaside se componentes construyen con *componentes*. Vamos a estudiar detenidamente cómo funciona Seaside implementando el componente *Hello World*.

Todo componente Seaside debe heredar directa o indirectamente de

WAComponent, como muestra la figura Figure 12.8.

Define una subclase de WAComponent *llamada* WAHelloWorld.

Los componentes deben saber cómo representarse a sí mismos. Normalmente esto se hace implementando el método renderContentOn:, que recibe como argumento una instancia de WAHtmlCanvas, que sabe cómo generar XHTML.

Implementa el siguiente método y ponlo en un protocolo llamado rendering:

```
WAHelloWorld»renderContentOn: html
  html text: 'hello world'
```

Ahora debemos informar a Seaside de que este componente está dispuesto a ser una aplicación independiente.

Implementa el siguiente método de clase de WAHelloWorld.

```
WAHelloWorld class»canBeRoot
  ↑ true
```

¡Estamos a punto de acabar!

Dirige tu navegador web a http://localhost:8080/seaside/config, *añade un nuevo punto de entrada llamado "hello" y pon como su componente raíz* WAHelloWorld. *Ahora dirige tu navegador a* http://localhost:8080/seaside/hello. *!Ya está! Deberías ver una página web como la de la figura Figure 12.7.*

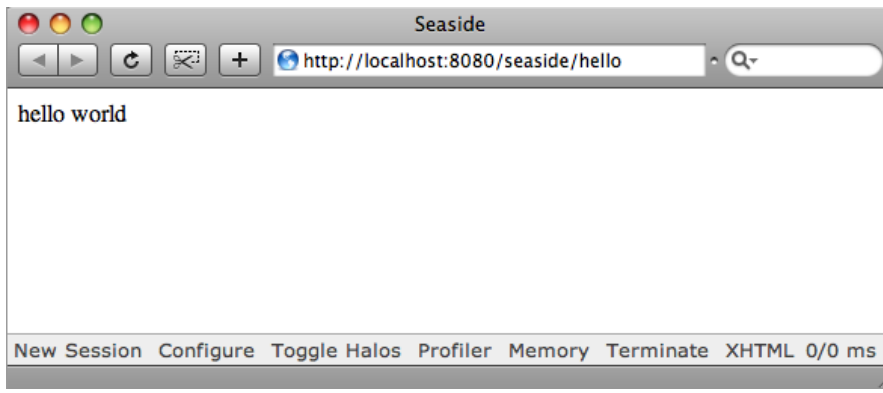


Figure 12.7: "Hello World" en Seaside

Marcha atrás de estado y la aplicación "Counter"

La aplicación "counter" es sólo ligeramente más compleja que la aplicación "hello world".

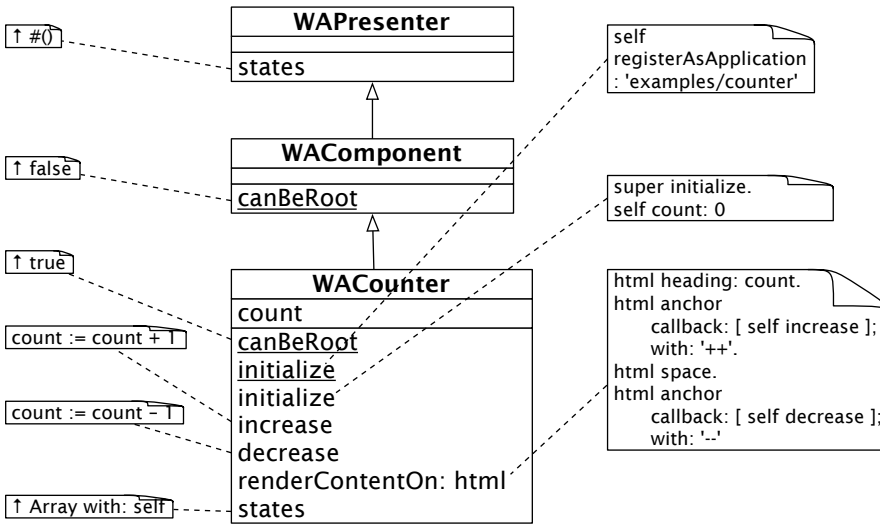


Figure 12.8: La clase WACounter, que implementa la aplicación *counter*. Los métodos con nombres subrayados son de clase; aquellos con nombres en texto normal son de instancia.

La clase WACounter es una aplicación independiente, con lo que la clase WACounter debe responder true al mensaje canBeRoot. También debe registrarse como una aplicación; esto se hace en su método de clase initialize, como muestra la figura Figure 12.8. WACounter define dos métodoso, increase y decrease, que serán disparados desde los vínculos ++ y -- de la página web. También define una variable de instancia count para registrar el estado del contador. Sin embargo, también queremos que Seaside sincronice el contador con la página del navegador: cuando el usuario hace click en el botón "atrás" del navegador, queremos que Seaside "retroceda" el estado del objeto WACounter. Seaside incluye un mecaniso general de vuelta a atrás, pero cada aplicación tiene que decirle a Seaside qué partes de su estado tiene que seguir.

Un componente habilita la vuelta a atrás implementando el método states en el lado de instancia: states debería devolver un array que contenga todos los objetos a seguir. En este caso, el objeto WACounter se añade él mismo a la tabla de objetos que pueden retrocederse de Seaside Seaside devolviendo Array with: self.

Advertencia. Hay un punto sutil pero importante al que estar atento cuando se declaran objetos para retroceder. Seaside sigue el estado haciendo una *copia* de todos los objetos declarados en el array states. Hace esto usando un objeto WASnapshot; WASnapshot es una subclase de IdentityDictionary que registra los objetos a seguir como claves y copias superficiales de su estado como valores.

Si el estado de una aplicación da marcha atrás a una instantánea particular, el estado de cada objeto que se añadió al diccionario de instantáneas se sobreescribe por la copia guardada en la instantánea.

El aspecto al que hay que estar atento es el siguiente: En el caso de WACounter, podrías pensar que el estado a seguir es un número — el valor de la variable de instancia count. Sin embargo, hacer que el método states retorne Array with: count no funcionará. Esto es así porque el objeto nombrado por count es un entero y los enteros son inmutables. Los métodos increase y decrease no cambian el estado del objeto 0 a 1 o el objeto 3 a 2. En cambio, count nombra un entero diferente: cada vez que count se incrementa o decrementa, el objeto nombrado por count es *reemplazado* por otro. Por esto es por lo que WACounter»states debe retornar Array with: self. Cuando el estado de un objeto WACounter es reemplazado por un estado previo, el *valor* de cada una de las variables de instancia en el objeto es reemplazado por un valor previo; este reemplaza correctamente el valor actual de count por un valor anterior.

12.4 Generando XHTML

El propósito de una aplicación web es crear o "presentar" páginas web. Como mencionamos en Section 12.3, cada componente Seaside es responsable de presentarse a sí mismo. Así pues, vamos a comenzar nuestra exploración de la presentación viendo cómo el componente contador se presenta a sí mismo.

Presentando el Contador

La presentación del contador es relativamente directa; el código se muestra en la figura Figure 12.8. El valor actual del contador se muestra como una cabecera XHTML, y las operaciones de incrementar y decrementar se implementan como anclas html (es decir, vínculos) con retrollamadas a bloques que enviarán increase y decrease al objeto contador.

Echaremos un vistazo más de cerca al protocolo de presentación en un momento. Pero antes de hacerlo, echemos un vistazo rápido al multicontador.

De Contador a Multicontador

WAMultiCounter que se muestra en la figura Figure 12.9 también es una aplicación independiente, con lo que sobreescribe canBeRoot para responder true . Además, es un componente *compuesto*, con lo que Seaside requiere que declare sus hijos implementando un método children que responda con un array con todos los componentes que contiene. Se presenta a sí mismo presentando cada uno de sus subcomponentes, separados por una línea horizontal. Además de los métodos de inicialización de instancia y estáticos, no hay nada más en el multicontador.

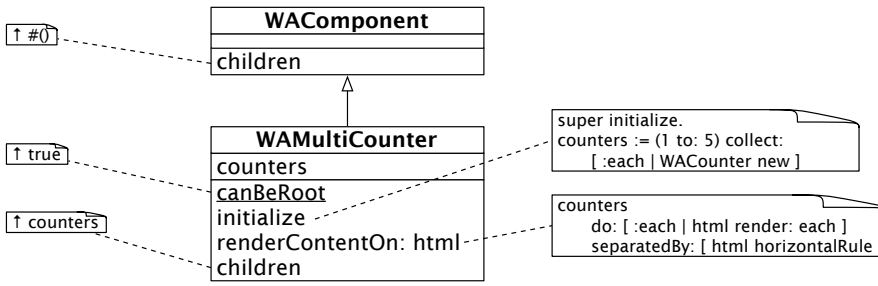


Figure 12.9: WAMultiCounter

Más sobre la Presentación XHTML

Como puedes ver en estos ejemplos, Seaside no usa plantillas para generar páginas web. En su lugar, genera XHTML programadamente. La idea básica es que cada componente Seaside debería sobreescribir el método renderContentOn:; este mensaje será enviado por el marco de trabajo a cada componente que necesita ser presentado. Este mensaje renderContentOn: tendrá un argumento que es un *lienzo html* sobre el que el componente debe presentarse a sí mismo. Por convenio, el parámetro del lienzo html se llama html.

Un lienzo html es análogo al lienzo gráfico usado por Morphic (y la mayoría de otros marcos de trabajo de dibujo) para abstraerse de los detalles dependientes del dispositivo de dibujo.

Estos son algunos de los métodos de presentación más básicos:

```
html text: 'hello world'.   "presenta una cada de texto plano"
html html: '&ndash;'.       "presenta un conjuro XHTML"
html render: 1.             "presenta cualquier objeto"
```

El mensaje render: anyObject se puede enviar a un lienzo html para presentar anyObject; se utiliza normalmente para presentar subcomponentes. A anyObject se le enviará el mensaje renderContentOn: esto es lo que ocurre en el multi-contador (ver la Figure 12.9).

Usando Pinceles

Un lienzo proporciona un número de *pinceles* que pueden usarse para presentar (*i.e.*, "pintar) contenido en el lienzo. Hay lienzos para cada tipo de elemento XHTML —párrafos, tablas, listas y demás. Para ver el protocolo completo de lienzos y métodos de utilidad, deberías navegar por la clase WACanvas y sus subclases. El argumento para renderContentOn: es realmente una instancia de la subclase WARenderCanvas.

Ya hemos visto el siguiente pincel utilizado en los ejemplos contador y multi-contador:

```
html horizontalRule.
```

En la figura Figure 12.10 podemo ver la salida de muchos de los pinceles básicos ofrecidos por Seaside.[4] El componente raíz SeasideDemo simplemente presenta sus subcomponentes, que son instancias de SeasideHtmlDemo, SeasideFormDemo, SeasideEditCallDemo y SeasideDialogDemo, como se muestra en method 12.1.

Method 12.1: SeasideDemo»renderContentOn:

```
SeasideDemo»renderContentOn: html
   html heading: 'Rendering Demo'.
   html heading
      level: 2;
      with: 'Rendering basic HTML: '.
   html div
      class: 'subcomponent';
      with: htmlDemo.
   "render the remaining components ..."
```

Recuerda que un componente raíz debe siempre declarar sus hijos o Seaside rechazará presentarlos.

```
SeasideDemo»children
   ↑ { htmlDemo . formDemo . editDemo . dialogDemo }
```

Ten en cuenta que hay dos formas distintas de instanciar el pincel heading . La primera forma es establecer el texto directamente enviando el men-

[4] El código fuente para method 12.1 está en el paquete PBE−SeasideDemo en el proyecto http://www.squeaksource.com/PharoByExample.

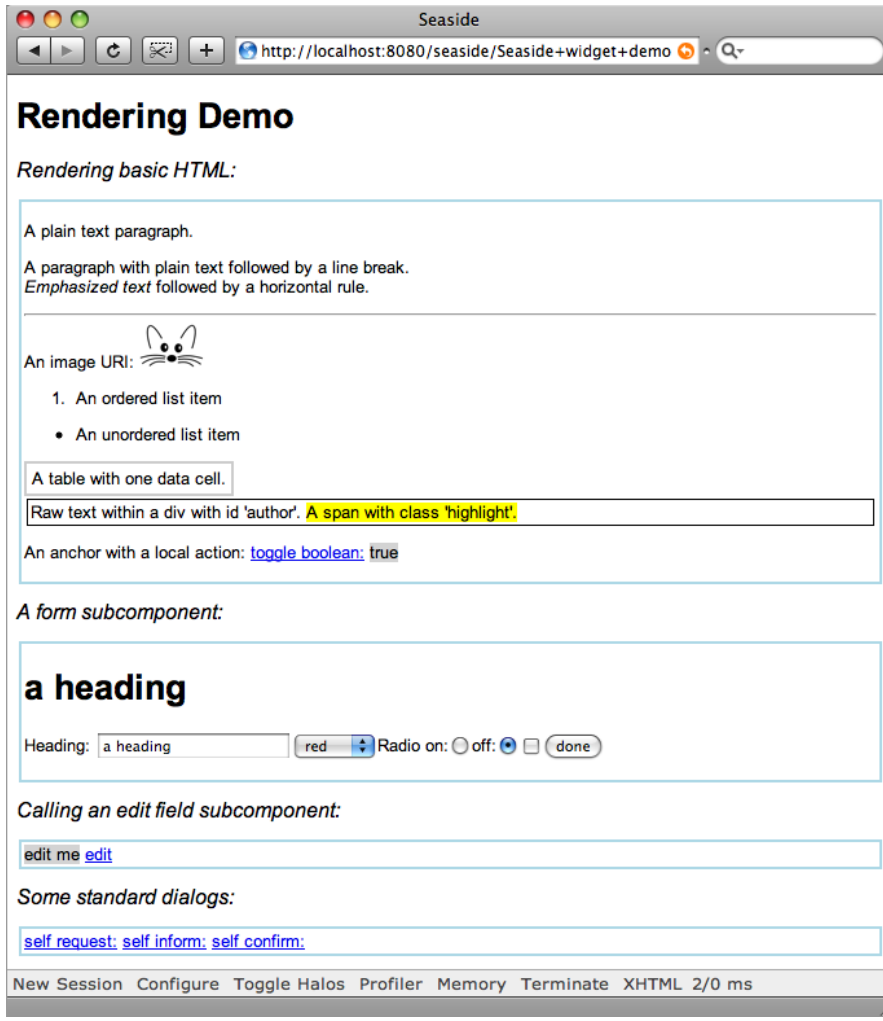


Figure 12.10: RenderingDemo

saje heading:. La segunda forma es instanciar el pincel enviando heading y entonces enviar una cascada de mensajes al pincel para establecer sus propiedades y presentarlo.

Muchos de los pinceles disponibles pueden usarse de estas dos formas.

> Si envías una cascada de mensajes al pincel, incluyendo el mensaje with:, entonces with: debería ser el mensaje *final*. with: establece el contenido y presenta el resultado.

En method 12.1, la primera cabecera está al nivel 1, ya que este es el valor por defecto. Explícitamente ponemos el nivel de la segunda cabecera a 2. El subcomponente es presentado como un *div* XHTML con la clase CSS "subcomponent". (Más sobre CSS en Section 12.5.) También ten en cuenta que es necesario que el argumento para el mensaje de palabra clave with: no sea una cadena literal: puede ser otro componente, o incluso — como en el siguiente ejemplo — un bloque conteniendo más acciones de presentación.

El componente SeasideHtmlDemo demuestra muchos de los pinceles más básicos. La mayoría del código debería explicarse por sí solo.

```
SeasideHtmlDemo»renderContentOn: html
  self renderParagraphsOn: html.
  self renderListsAndTablesOn: html.
  self renderDivsAndSpansOn: html.
  self renderLinkWithCallbackOn: html
```

Es una práctica común dividir métodos de presentación largos en muchos métodos auxiliares, como hemos hecho aquí.

> No pongas todo tu código de presentación en un único método. Divídelo en métodos auxiliares nombrados usando el patrón render*On:. Todos los métodos de presentación van en el protocolo *rendering*. No envíes renderContentOn: desde tu propio código, usa render: en su lugar.

Mira el siguiente código. El primer método auxiliar, SeasideHtmlDemo» renderParagraphsOn:, muestra cómo generar párrafos XHTML, texto plano y enfatizado e imágenes. Ten en cuenta que en Seaside los elementos sencillos son presentados especificando el texto que contienen directamente, mientras que los elementos complejos se especifican usando bloques. Este es un convenio sencillo para ayudatte a estructurar tu código de presentación.

```
SeasideHtmlDemo»renderParagraphsOn: html
  html paragraph: 'Un parrafo con texto plano.'.
  html paragraph: [
    html
      text: 'Un parrafo con texto plano seguido por una nueva linea. ';
      break;
      emphasis: 'Texto con enfasis ';
```

```
        text: 'seguido por una linea horizontal.';
        horizontalRule;
        text: 'Una URI de imagen: '.
      html image
        url: self squeakImageUrl;
        width: '50']
```

El siguiente método auxiliar, SeasideHtmlDemo»renderListsAndTablesOn:, muestra cómo generar listas y tablas. Una tabla usa dos niveles de bloques para mostrar cada una de sus filas y las celdas dentro de las filas.

```
SeasideHtmlDemo»renderListsAndTablesOn: html
  html orderedList: [
    html listItem: 'Un elemento de una lista ordenada'].
  html unorderedList: [
    html listItem: 'Un elemento de una lista desordenada'].
  html table: [
    html tableRow: [
      html tableData: 'Una tabla con una celda de datos.']]
```

El siguiente ejemplo muestra cómo podemos especificar *div*s y *span*s con atributos CSS *class* o *id*. Por supuesto, los mensajes class: e id: también pueden enviarse a los otros pinceles, no sólo a *div*s y *span*s. El método SeasideDemoWidget»style define cómo deberían ser mostrados estos elementos XHTML (ver Section 12.5).

```
SeasideHtmlDemo»renderDivsAndSpansOn: html
  html div
    id: 'author';
    with: [
      html text: 'Texto plano con un div con id "author". '.
      html span
        class: 'highlight';
        with: 'Un span con class "highlight".']
```

Finalmente vemos un sencillo ejemplo de un vínculo, creado asociando una sencilla retrollamada con un "ancla" (*i.e.*, un vínculo). Hacer click en el vínculo causará que el texto a continuación cambie entre "true" y "false" cambiando la variable de instancia toggleValue.

```
SeasideHtmlDemo»renderLinkWithCallbackOn: html
  html paragraph: [
    html text: 'Un ancla con una accion local: '.
    html span with: [
      html anchor
        callback: [toggleValue := toggleValue not];
        with: 'cambia el booleano:'].
    html space.
    html span
```

```
        class: 'boolean';
        with: toggleValue ]
```

> Ten en cuenta que las acciones deberían aparecer sólo en retrollamadas. ¡El código ejecutado durante la presentación no debería cambiar el estado de la aplicación!

Formularios

Los formularios se presentan igual que los otros ejemplos que ya hemos visto. Este es el código para el componente SeasideFormDemo de la figura Figure 12.10.

```
SeasideFormDemo»renderContentOn: html
   | radioGroup |
   html heading: heading.
   html form: [
      html span: 'Cabecera: '.
      html textInput on: #heading of: self.
      html select
         list: self colors;
         on: #color of: self.
      radioGroup := html radioGroup.
      html text: 'Radio on:'.
      radioGroup radioButton
         selected: radioOn;
         callback: [radioOn := true].
      html text: 'off:'.
      radioGroup radioButton
         selected: radioOn not;
         callback: [radioOn := false].
      html checkbox on: #checked of: self.
      html submitButton
         text: 'hecho' ]
```

Puesto que un formulario es una entidad compleja, se presenta usando un bloque. Ten en cuenta que todos los cambios de estado ocurren en las retrollamadas, no como parte de la presentación.

Hay una característica de Seaside utilizada aquí que merece una mención especial, que es el mensaje on:of:. En el ejemplo, este mensaje se usa para asociar un campo de entrada de texto con la variable heading. Anclas y botones también soportan este mensaje. El primer argumento es el nombre de una variable de instancia para la que se han definido métodos de acceso; el segundo argumento es el argumento al que esta variable de instancia pertenece. Ambos métodos de acceso, observador (heading) y modificador

(heading:) son comprendidos por el objeto con el convenio de nombre habitual. En este caso de un campo de entrada de texto, esto nos ahorra el problema de tener que definir una retrollamada que actualice el campo además de tener que asociar los contenidos por defecto del campo de entrada html con el valor actual de la variable de instancia. Usando on: #heading of: self, la variable heading se actualiza automáticamente cuando el usuario actualiza el campo de entrada de texto.

El mismo mensaje se usa dos veces más en este ejemplo, para causar que la selección de un color en el formulario html actualice la variable color, y para asociar el resultado del checkbox con la variable checked. Muchos otros ejemplos pueden encontrarse en los tests funcionales de Seaside. Echa un vistazo a la categoría *Seaside-Tests-Functional*, o simplemente dirige tu navegador a http://localhost:8080/seaside/tests/alltests. Selecciona WAInputTest y haz click en el botón Restart para ver la mayoría de las características de los formularios.

No olvides que si activas Toggle Halos, puedes navegar directamente por el código fuente de los ejemplos usando el navegador de clases de Seaside.

12.5 CSS: Hojas de estilo en cascada

Las hojas de estilo en cascada[5], o CSS para abreviar, han surgido como una forma estándar de separar el estilo del contenido para las aplicaciones web. Seaside se basa en CSS para evitar llenar tu código de presentación con consideraciones de diseño.

Puedes establecer la hoja de estilos CSS para tus componentes web definiendo el método style, que debería devolver una cadena conteniendo las reglas CSS para ese componente. Los estilos de todos los componentes mostrados en una página web se unen entre sí, con lo que cada componente puede tener su propio estilo. Una mejor aproximación puede ser definir una clase abstracta para tu aplicación web que defina un estilo común para todas sus subclases.

En realidad, para las aplicaciones desplegadas, es más habitual definir las hojas de estilos como archivos externos. De esta forma el aspecto del componente está completamente separado de su funcionalidad. (Echa un vistazo a WAFileLibrary, que proporciona un modo de servir archivos estáticos sin la necesidad de un servidor independiente.)

Si ya estás familiarizado con CSS, esto es todo lo que necesitas saber. En caso contrario, continúa leyendo una breve descripción de CSS.

En lugar de codificar directamente los atributos de presentación en los elementos de texto y párrafos de tus páginas web, con CSS definirás diferentes

[5] http://www.w3.org/Style/CSS/

```
SeasideDemoWidget»style
    ↑ '
body {
    font: 10pt Arial, Helvetica, sans−serif, Times New Roman;
}
h2 {
    font−size: 12pt;
    font−weight: normal;
    font−style: italic;
}
table { border−collapse: collapse; }
td {
    border: 2px solid #CCCCCC;
    padding: 4px;
}
#author {
    border: 1px solid black;
    padding: 2px;
    margin: 2px;
}
.subcomponent {
    border: 2px solid lightblue;
    padding: 2px;
    margin: 2px;
}
.highlight { background−color: yellow; }
.boolean { background−color: lightgrey; }
.field { background−color: lightgrey; }
'
```

Figure 12.11: Hoja de estilo común de SeasideDemoWidget.

clases de elementos y colocarás todas las consideraciones de presentación en una hoja de estilos separada. Las entidades de tipo párrafo se llaman *div*s y las entidades de tipo texto son *span*s. Entonces se definen nombres simbólicos, como "highlight" (ver ejemplo de abajo), para el texto resaltado y se especifica en la hoja de estilos cómo se va a presentar el texto resaltado.

Fundamentalmente una hoja de estilos CSS consiste en un conjunto de reglas que especifican cómo formatear los elementos XHTML dados. Cada regla consta de dos partes. Existe un *selector* que especifica a qué elementos XHTML aplica la regla y una *declaración* que define un número de atributos para esos elementos.

La Figure 12.11 representa una hoja de estilo sencilla para la demo de presentación mostrada anteriormente en la Figure 12.10. La primera regla especifica una preferencia para la tipografía a utilizar por el body de la página

web. Las siguientes reglas especifican propiedas de las cabeceras de segundo nivel (h2), tablas (table) y datos de las tablas (td).

El resto de reglas tienen selectores que corresponden con los elementos XHTML que tengan los atributos "class" o"id". Los selectores CSS para los atributos clase comienzan con un "." y aquellos para los atributos id con "#". La principal diferencia entre los atributos clase e id es que muchos elementos pueden tenir la misma clase, pero sólo un elemento muy tener un id dado (*i.e.*, un *identificador*). Así que, mientras un atributo clase, como highlight, puede aparecer varias veces en cualquier página, un id debe identificar un elemento *único* en la página, como un menú particular, la fecha de modificación o el autor. Ten en cuanta que un elemento XHTML concreto puede tener varias clases, en cuyo caso todos los atributos de presentación se aplicarán en secuencia.

Las condiciones de los selectores pueden combinarse, con lo que el selector div.subcomponent sólo se aplicará a aquellos elementos XHTML que sean un div *y* tengan una atributo clave igual a "subcomponent".

También es posible especificar elementos anidados, aunque rara vez es necesario. Por ejemplo, el selector "p span" aplicará a un span dentro de un párrafo, pero no dentro de un div.

Existen muchos libros y sitios web que pueden ayudarte a aprender CSS. Para una espectacular demostración del poder de CSS, te recomendamos que eches un vistazo al Jardín Zen CSS[6], que muestra cómo el mismo contenido puede presentarse de maneras radicalmente diferentes simplemente cambiando la hoja de estilos CSS.

12.6 Gestión del control del flujo

Seaside hace especialmente fácil diseñar aplicaciones web con un control de flujo no trivial. Fundamentalmente existen dos mecanismos que puedes utilizar:

1. Un componente puede *llamar* a otro componente enviando visitante call: visitado. El visitante es reemplazado temporalmente por el visitado, hasta que el visitado devuelve el control enviando answer:. El visitante normalmente es self, pero también podría ser cualquier otro componente visible actualmente.

2. Un flujo de trabajo puede definirse como una *tarea*. Esta es un caso especial de componente que hereda de WATask (en lugar de WAComponent). En lugar de definir renderContentOn:, no define ningún contenido, sino

[6] http://www.csszengarden.com/

que define un método go que envía una serie de mensajes call: para activar varios subcomponentes.

LLamada y respuesta

Llamada y respuesta se usan para producir diálogos sencillos.

Hay un ejemplo trivial de call: y answer: en la demo de presentación de la Figure 12.10. El componente SeasideEditCallDemo muestra un campo de texto y un vínculo *edit*. La retrollamada para el vínculo *edit* llama a una nueva instancia de SeasideEditAnswerDemo inicializada con el valor del campo de texto. La retrollamada también actualiza este campo de texto con el resultado que se envía como respuesta.

(Subrayamos los mensajes call: y answer: enviados para llamar la atención sobre ellos.)

```
SeasideEditCallDemo»renderContentOn: html
   html span
      class: 'field';
      with: self text.
   html space.
   html anchor
      callback: [self text: (self call: (SeasideEditAnswerDemo new text: self text))];
      with: 'edit'
```

Lo que resulta especialmente elegante es que el código no hace absolutamente ninguna referencia a la nueva página web que debe crearse. En tiempo de ejecución, se crea una nueva página en la que el componente SeasideEditCallDemo es reemplazado por un componente SeasideEditAnswerDemo; el componente padre y el resto de componentes no se ven modificados.

> call: y answer: nunca deben usarse durante la presentación. Pueden ser enviados de forma segura desde una retrollamada o desde el método go de una tarea.

El componente SeasideEditAnswerDemo también es extraordinariamente sencillo. Simplemente presenta un formulario con un campo de texto. El botón de envío está vinculado con una retrollamada que responderá el valor final del campo de texto.

```
SeasideEditAnswerDemo»renderContentOn: html
   html form: [
      html textInput
```

```
        on: #text of: self.
    html submitButton
        callback: [ self answer: self text ];
        text: 'ok'.
    ]
```

Eso es todo.

Seaside se ocupa del control del flujo y de la correcta presentación de todos los componentes. Curiosamente, el botón "volver" del navegador también funcionará correctamente (aunque los efectos colaterales no se deshacen a no ser que añadamos algunos pasos adicionales).

Métodos de utilidad

Puesto que ciertos diálogos llamada–respuesta son muy comunes, Seaside proporciona algunos métodos de utilidad para ahorrarte el problema de escribir componentes como SeasideEditAnswerDemo. Los diálogos generados se muestran en la Figure 12.12. Podemos ver los métodos de utilidad empleados en SeasideDialogDemo»renderContentOn:

El mensaje request: realiza una llamada a un componente que te permitirá editar un campo de texto. El componente responde con la cadena editada. También pueden especificarse opcionalmente una etiqueta y un valor por defecto.

```
SeasideDialogDemo»renderContentOn: html
    html anchor
        callback: [ self request: 'edit this' label: 'done' default: 'some text' ];
```

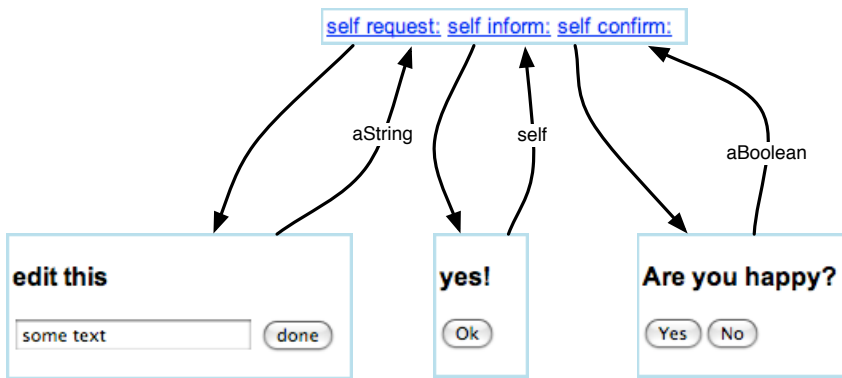


Figure 12.12: Algunos diálogos estándar

```
      with: 'self request:'.
...
```

El mensaje inform: llama a un componente que simplemente muestra el mensaje del argumento y espera a que el usuario pulse en "ok". El componente llamada simplemente retorna self.

```
...
   html space.
   html anchor
      callback: [ self inform: 'yes!' ];
      with: 'self inform:'.
...
```

El mensaje confirm: realiza una pregunta y espera a que el usuario seleccione "Yes" o "No". El componente responde con un booleano, que puede utilizarse para realizar otras acciones.

```
...
   html space.
   html anchor
      callback: [
         (self confirm: 'Are you happy?')
            ifTrue: [ self inform: ':-)' ]
            ifFalse: [ self inform: ':-(' ]
         ];
      with: 'self confirm:'.
```

Algunos otros métodos de utilidad, como chooseFrom:caption:, se definen en el protocolo *convenience* de WAComponent.

Tareas

Una tarea es un componente que hereda de WATask. No presenta nada por sí misma, sino que simplemente llama a otros componentes en un control de flujo definido implementando el método go.

WAConvenienceTest es un ejemplo sencillo de una tarea definida en la categoría *Seaside-Tests-Functional*. Para ver sus efectos, simplemente dirige tu navegador a http://localhost:8080/seaside/tests/alltests, selecciona WAConvenienceTest y haz click en Restart.

```
WAConvenienceTest»go
   [ self chooseCheese.
     self confirmCheese ] whileFalse.
   self informCheese
```

Esta tarea llamada a su vez a tres componentes. El primero, generado por el método de utilidad chooseFrom: caption:, es un WAChoiceDialog que pide al usuario que escoja un queso.

```
WAConvenienceTest»chooseCheese
  cheese := self
    chooseFrom: #('Greyerzer' 'Tilsiter' 'Sbrinz')
    caption: 'What''s your favorite Cheese?'.
  cheese isNil ifTrue: [ self chooseCheese ]
```

El segundo es un WAYesOrNoDialog para confirmar la elección (generado por el método de utilidad confirm:).

```
WAConvenienceTest»confirmCheese
  ↑self confirm: 'Is ', cheese,  ' your favorite cheese?'
```

Finalmente se llama a un WAFormDialog (por medio del método de utilidad inform:).

```
WAConvenienceTest»informCheese
  self inform: 'Your favorite cheese is ', cheese, '.'
```

Los diálogos generados se muestran en la Figure 12.13.

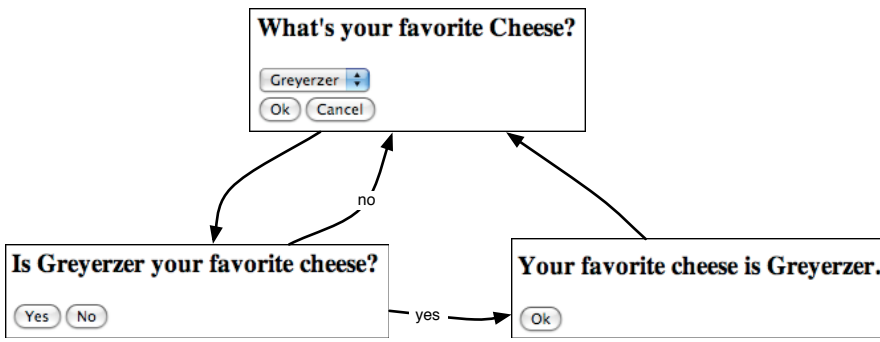


Figure 12.13: Una tarea sencilla

Transacciones

Como vimos en la Section 12.3, Seaside puede realizar un seguimiento de la correspondencia entre el estado de los componentes y las páginas web individuales por medio de componentes que registran sus estados para la vuelta atrás: todo lo que un componente necesita es implementar el método states para devolver un array con todos los objetos cuyo estado debe seguirse.

Sin embargo, algunas veces, no queremos volver atrás el estado: en su lugar queremos *evitar* que el usuario accidentalmente deshaga los efectos que deberían ser permanentes. A esto nos referimos frecuentemente como "el problema del carrito de la compra". Una vez que has validado tu carrito de la compra y pagado por los elementos que has comprado, no debería ser posible volver "atrás" con el navegador y añadir más elementos al carrito.

Seaside te permite evitar esto definiendo una tarea en la que ciertas acciones están agrupadas como *transacciones*. Puedes volver atrás dentro de una transacción, pero una vez que una transacción está completa, ya no puedes deshacerla. Las páginas correspondientes están *invalidadas* y cualquier intento de volver a ellas causará que Seaside genere un aviso y redirija al usuario a la página válida más reciente.

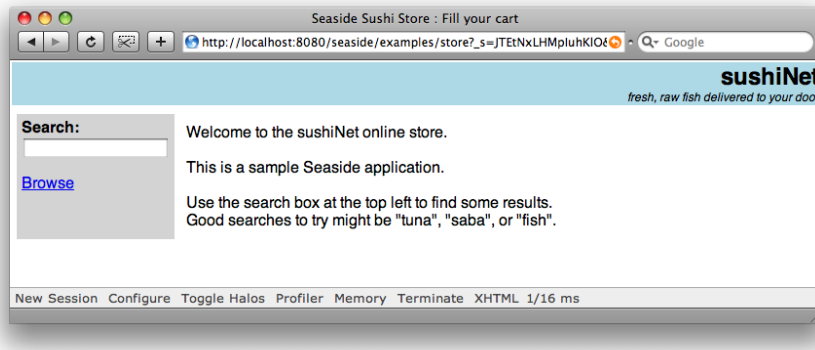


Figure 12.14: La tienda de Sushi

La aplicación de Seaside *Sushi Store* es una aplicación de ejemplo que ilustra muchas de las características de Seaside, incluyendo las transacciones. La aplicación está incluída con tu instalación de Seaside, así que puedes probarla dirigiendo tu navegador a http://localhost:8080/seaside/examples/store.[7]

La tienda de sushi soporta el siguiente flujo:

1. Visita la tienda.
2. Navega o busca sushi.
3. Añade sushi a tu carrito de la compra.
4. Lanza el pedido.
5. Verifica tu pedido.

[7]Si no puedes encontrarla en tu imagen, existe una versión de la tienda de sushi disponible en SqueakSource en http://www.squeaksource.com/SeasideExamples/.

6. Introduce la dirección de entrega.
7. Verifica la dirección de entrega.
8. Introduce la información del pago.
9. ¡Tu pescado está en camino!

Si activas los halos, verás que el componente de primer nivel de la tienda de sushi es una instancia de WAStore. No hace nada, salvo presentar la barra de título y entonces presenta task, una instancia de WAStoreTask.

```
WAStore»renderContentOn: html
   "... render the title bar ..."
   html div id: 'body'; with: task
```

WAStoreTask captura la secuencia de este flujo. En un par de puntos es crítico que el usuario no sea capaz de volver atrás y cambiar la información enviada.

 *"Compra" algo de sushi y entonces usa el botón "atrás" para intentar añadir más sushi al carrito. Obttendrás el mensaje "That page has expired."*

Seaside permite al programador declarar que cierta parte de un flujo de trabajo actúa como una transacción: una vez que la transacción se completa, el usuario no puede volver atrás y deshacerla. Declaras esto enviando isolate: a una tarea con el bloque transaccional como su argumento. Podemos ver esto en el flujo de trabajo de la tienda de sushi a continuación:

```
WAStoreTask»go
   | shipping billing creditCard |
   cart := WAStoreCart new.
   self isolate:
      [[self fillCart.
      self confirmContentsOfCart]
         whileFalse].

   self isolate:
      [shipping := self getShippingAddress.
      billing := (self useAsBillingAddress: shipping)
               ifFalse: [self getBillingAddress]
               ifTrue: [shipping].
      creditCard := self getPaymentInfo.
      self shipTo: shipping billTo: billing payWith: creditCard].

   self displayConfirmation.
```

Aquí vemos bastante claramente que existen dos transacciones. La primera rellena el carrito y cierra la fase de compra. (Los métodos auxiliares fillCart etc. se ocupan de instanciar y llamar a los componentes adecuados.)

Una vez que has confirmado los contenidos del carrito no puedes volver atrás sin iniciar una nueva sesión. La segunda transacción cumplimenta los datos de envío y pago. Puedes navegar atrás y adelante dentro de la segunda transacción hasta que confirmas el pago. Sin emgargo, una vez que ambas transacciones estás completas, cualquier intento de navegar atrás fallará.

Las transacciones también pueden anidarse. Una demostración sencilla de estos se encuentra en la clase WANestedTransaction. El primer isolate: recibe como argumento un bloque que contiene otro isolate: anidado.

```
WANestedTransaction»go
  self inform: 'Before parent txn'.
  self isolate:
      [self inform: 'Inside parent txn'.
      self isolate: [self inform: 'Inside child txn'].
      self inform: 'Outside child txn'].
  self inform: 'Outside parent txn'
```

Ve a http://localhost:8080/seaside/tests/alltests, selecciona WATransactionTest *y pulsa en* Restart. *Prueba a navegar atrás y adelante dentro de las transacciones padre e hija pulsando el botón* atrás *y después en* ok. *Observa que tan pronto como una transacción se completa, ya no puedes volver atrás dentro de la transacción sin generar un error al pulsar* ok.

12.7 Un ejemplo tutorial completo

Veamos como podemos construir una aplicación Seaside completa desde cero.[8] Construiremos una calculadora de notación polaca inversa como una aplicación Seaside que utiliza una simple pila como su modelo subyacente. Incluso más, la interface Seaside nos permitirá alternar entre dos vistas — una que solo nos muestre el valor actual en el tope de la pila, y la otra que nos muestre el estado completo de la pila.

La calculadora con las dos opciones de vistas se muestra en Figure 12.15.

Comenzamos por implementar la pila y sus pruebas.

Define una nueva clase llamada MyStackMachine *con una variable de instancia* contents *inicializada a una nueva* OrderedCollection.

```
MyStackMachine»initialize
  super initialize.
```

[8] El ejercicio debe llevar como mucho un par de horas. Si prefieres solo mirar el código fuente completo, puedes obtenerlo del proyecto SqueakSource http://www.squeaksource.com/PharoByExample. El paquete a cargar es *PBE-SeasideRPN*. El siguiente tutorial utiliza nombres de clases levemente diferentes para que puedas comparar tu implementación con las nuestra.

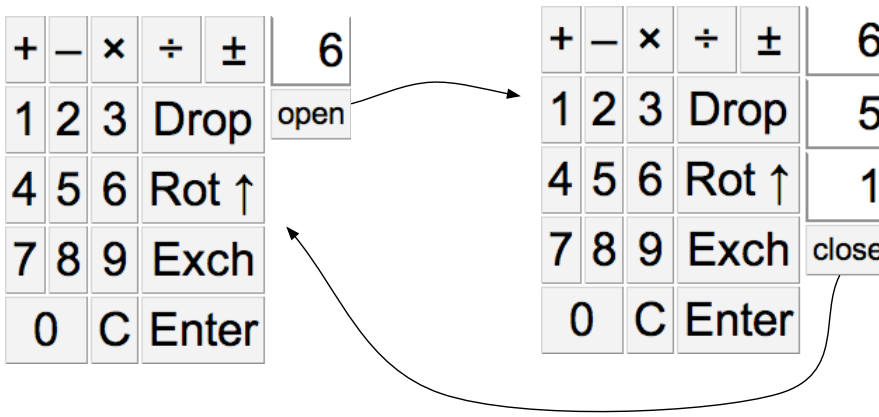


Figure 12.15: RPN calculator and its stack machine

```
contents := OrderedCollection new.
```

Esta pila debería proveer operationes para poner y sacar valores, ver el tope de la pila y realizar varias operaciones aritméticas para sumar restar, multiplicar y dividir valores en la pila.

Escribe algunas pruebas para las operaciones de la pila y luego implementalas. Aqui hay una prueba de muestra:

```
MyStackMachineTest»testDiv
   stack
      push: 3;
      push: 4;
      div.
   self assert: stack size = 1.
   self assert: stack top = (4/3).
```

Puedes considerar usar algunos métodos de ayuda para las operaciones aritmética para chequear que haya dos números en la pila antes de hacer cualquier cosa y levantar un error si esta precondición no es cumplida. [9] Si haces esto, la mayoría de tus métodos serán de una o dos lineas de longitud.

También pueden considerar implementar MyStackMachine»printOn: para facilitar la depuración de la implementación de tu pila con la ayuda de un inspector de objetos. (Pista: solo delega la impresión a la variable contents.)

Completa MyStackMachine *escribiendo las operaciones* dup *(apila un duplicado*

[9] Es una buena idea usar Object»assert: para especificar las precondiciones de una operación. Este método levantará una AssertionFailure si el usuario intenta usar la pila en un estado invalido.

del valor en el tope de la pila), exch *(intercambia los dos valores topes en la pila), y* rotUp *(rota la todos los contenidos de la pila — el tope será será movido al fondo de la pila).*

Ahora tenemos una implementación de una pila. Podemos comenzar a implementar la calculadora NPI Seaside.

Haremos uso de 5 clases:

- MyRPNWidget — esta deberia ser una clase abstracta que defina la hoja de estilo CSS común para la aplicación, y otro comportamiento común para los componentes de la calculadora NPI. Es una subclase de WAComponent y la superclase directa de las siguientes cuatro clases.
- MyCalculator — este es el componentes raíz. Debería registrar la aplicación (en la parte de la clase), instanciar y hacer render de sus subcomponentes, y registrar cualquier estado para backtracking.
- MyKeypad — esta muestra las teclas que usamos para interactuar con la calculadora.
- MyDisplay — este componente muetra el tope de la pila y provee un botón para llamar a otro componente para mostrar la vista detallada.
- MyDisplayStack — este componente, muestra la vista detallada de la pila y provee un botón para responder. Es una subclase de MyDisplay.

 *Define* MyRPNWidget *en la categoria* MyCalculator. *Define el* estilo *común para la aplicación.*

Esta es un CSS mínimo para la aplicación. Puedes hacerla mas atractiva si gustas.

```
MyRPNWidget»style
   ↑ 'table.keypad { float: left; }
td.key {
   border: 1px solid grey;
   background: lightgrey;
   padding: 4px;
   text-align: center;
}
table.stack { float: left; }
td.stackcell {
   border: 2px solid white;
   border-left-color: grey;
   border-right-color: grey;
   border-bottom-color: grey;
   padding: 4px;
   text-align: right;
```

```
}
td.small { font−size: 8pt; }'
```

Define MyCalculator *para ser el componente raíz y registralo asimismo como aplicaciónb (*i.e.*, implementa* canBeRoot *y* initialize *del lado de la clase). Implementa* MyCalculator»renderContentOn: *para hacer render de algo trivial (como su nombre), y verificar que la aplicación corre en un nagevador.*

MyCalculator es responsable de instanciar MyStackMachine, MyKeypad y MyDisplay.

Define MyKeypad *y* MyDisplay *como subclase de* MyRPNWidget. *Los tres componentes necesitarán acceso a una instancia común de la pila, entonces define una variable de instancia* stackMachine *y un método de inicialización* setMyStackMachine: *en su padre común,* MyRPNWidget. *Agrega variables de instancia* keypad *y* display *para* MyCalculator *e inicializarlos en* MyCalculator»initialize. *(No olvides de enviar* super initialize*!)*

Pasa una instancia compartida de la pila al keypad y al display en el mismo método initialize. Implementa MyCalculator»renderContentOn: *para simplemente hacer render cuando corresponda del keypad y el display. Para mostrar correctamente los subcomponentes, debes implementar* MyCalculator»children *para devolver un arreglo con el keypad y el display. Implement placeholder rendering methods for the keypad and the display and verify that the calculator now displays its two subcomponents.*

Ahora cambiaremos la implementación del display para mostrar el valor en el tope de la pila.

Usa una tabla con clase "keypad" conteniendo una fila con celda de datos con clase "stackcell". Cambia el método de rendering del keypad para asegurar que el número 0 es apilado en la pila en caso que este vacia. (Define y usa MyKeypad »ensureMyStackMachineNotEmpty.*) También hazlo mostrar una tabla vacia con la clase "keypad". Ahora la calculadora debería mostrar una celda conteniendo el valor 0. Si cambias los halos, deberías ver algo como esto:*

Ahora implementemos la interface para interactuar con la pila.

Primero define los siguientes métodos de ayuda, que harán más simple el guión de la interface:

```
MyKeypad»renderStackButton: text callback: aBlock colSpan: anInteger on: html
  html tableData
    class: 'key';
    colSpan: anInteger;
    with:
```

Figure 12.16: Mostrando el tope de la pila

```
            [html anchor
                callback: aBlock;
                with: [html html: text]]
```

```
MyKeypad»renderStackButton: text callback: aBlock on: html
   self
      renderStackButton: text
      callback: aBlock
      colSpan: 1
      on: html
```

Usaremos estos dos métodos para definir los botones en el keypad con las correspondientes métodos de callback. Ciertos botones pueden expandirse por múltiples columnas, pero por defecto ocuparán solo una.

Usa los dos métodos de ayuda para guionar el keypad como a continuación: (Pista: comienza por obtener el dígito e "introducir" teclas de trabajo y luego las operaciones aritméticas.)

```
MyKeypad»renderContentOn: html
 self ensureStackMachineNotEmpty.
 html table
   class: 'keypad';
   with: [
    html tableRow: [
       self renderStackButton: '+' callback: [self stackOp: #add] on: html.
       self renderStackButton: '&ndash;' callback: [self stackOp: #min] on: html.
       self renderStackButton: '&times;' callback: [self stackOp: #mul] on: html.
       self renderStackButton: '&divide;' callback: [self stackOp: #div] on: html.
       self renderStackButton: '&plusmn;' callback: [self stackOp: #neg] on: html ].
      html tableRow: [
       self renderStackButton: '1' callback: [self type: '1'] on: html.
       self renderStackButton: '2' callback: [self type: '2'] on: html.
       self renderStackButton: '3' callback: [self type: '3'] on: html.
       self renderStackButton: 'Drop' callback: [self stackOp: #pop]
         colSpan: 2 on: html ].
```

```
" and so on ... "
      html tableRow: [
        self renderStackButton: '0' callback: [self type: '0'] colSpan: 2 on: html.
        self renderStackButton: 'C' callback: [self stackClearTop] on: html.
        self renderStackButton: 'Enter'
          callback: [self stackOp: #dup. self setClearMode]
          colSpan: 2 on: html ]]
```

Chequea que el keypad se muestra correctamente. Si intentas hacer clic en las teclas, sin embargo, entontrarás que la calculadora aún no funciona …

Implementa MyKeypad»type: *para actualizar el tope de la pila agregando el dígito tipeado. Necesitarás convertir el valor del tope a una cadena de caracteres, actualizarlo y volver a convertirlo a entero, algo asi:*

```
MyKeypad»type: aString
   stackMachine push: (stackMachine pop asString, aString) asNumber.
```

Now when you click on the digit keys the display should be updated. (Be sure that MyStackMachine»pop returns the value popped, or this will not work!)

Now we must implement MyKeypad»stackOp: *Something like this will do the trick:*

```
MyKeypad»stackOp: op
   [ stackMachine perform: op ] on: AssertionFailure do: [ ].
```

El punto es que no estamos seguros de que todas las operacione sean exitosas, por ejemplo, la suma fallará si no tenemos dos números en la pila. Por el momento podemos ignorar estos errores. Si no sentimos más ambiciosos luego, podemos proveer algún feedback al usuario en el bloque de manejo de error.

Esta primera versión de la calculadora debería estar funcionando ahora. Intentar ingresar algunos números presionando las teclas de los dígitos, haciendo Enter *para apilar una copia del valor actual, e ingresando* + *para sumar los dos valores en el tope de la pila.*

Notarás que tipeo dígitos no se comporta de la manera esperada. En verdad la calculadora, debería estar al tanto de si se están tipeando un *nuevo* número, o agregando un número existente.

Adapta MyKeypad»type: *para comportarse de manera diferente dependiendo del modo actual de tipeo. Introduce una nueva variable de instancia* mode *que tome uno de los 3 valores* #typing *(cuando se esta tipeando),* #push *(luego de que se realizado una operación y tipear debería forzar al valor tope a ser apilado, o* #clear *(luego de*

haber realizar Enter *y el valor tope deba ser limpiado antes de tipear). El nuevo método* type: *se veria como:*

```
MyKeypad»type: aString
   self inPushMode ifTrue: [
      stackMachine push: stackMachine top.
      self stackClearTop ].
   self inClearMode ifTrue: [ self stackClearTop ].
   stackMachine push: (stackMachine pop asString, aString) asNumber.
```

"

El tipeo debería funcionar mejor ahora, pero aún es fustrante no se capaz de ver que están en la pila.

Define MyDisplayStack *una subclase de* MyDisplay. *Agrega un botón al método render de* MyDisplay *que llamará a una nueva instancia de* MyDisplayStack. *Necesitarás un anchor html que se parezca a lo siguiente:*

```
html anchor
   callback: [ self call: (MyDisplayStack new setMyStackMachine: stackMachine)];
   with: 'open'
```

El método de callback causará que la instancia actual de MyDisplay sea temporalmente reemplazada por una nueva instancia de MyDisplayStack cuyo trabajo is mostrar la pila completa. Cuando este componente señala que ha terminado (*i.e.*, enviando self answer), entonces el control retonará a la instancia original de MyDisplay.

Define el método de render de MyDisplayStack *para mostrar todos los valores de la pila. (Necesitarás definir un accesor para el* contenido *de la pila o puedes definir* MyStackMachine»do: *para iterar sobre los valores de la pila.) La vista de la pila debería tener también un boton etiquedado "close" cuyo método de callback simplemente ejecutará* self answer.

```
html anchor
   callback: [ self answer];
   with: 'close'
```

Ahora deberias ser capaz de *abrir* y *cerrar* la pila mientras estes usando la calculadora.

ESto es sin embargo una cosa que nos hemos olvidado. Intenta realizar algunas operaciones con la pila. Ahora usa el botón "back" de tu navegador e intenta realizar algunas operaciones más con la pila. (Por ejemplo, abrir la pila, tipear 1, Enter dos veces y +. La pila debería mostrar "2" y "1". Ahora presiona el botón "back". La pila ahora muestra tres veces 1 otra vez. Ahora si tipeas + la pila muestra "3". El backtracking no está funcionando aún.

Implementa MyCalculator»states *para devolver un arreglo con los contenidos de la pila. Chequea que el backtraking ahora funciona correctamente!.*

Siéntate y disfruta de un gran vaso de de algo cool!

12.8 Una mirada a AJAX

AJAX (Asynchronous JavaScript and XML) es una técnica para hacer aplicaciones web más interactivas explotando funcionalidad JavaScript del lado del cliente.

Dos librerias JavaScript bien conocidas son Prototype (http://www.prototypejs.org) y script.aculo.us (http://script.aculo.us). Prototype provee un marco de trabajo para facilitar la escritura de JavaScript. script.aculo.us provee algunas características adicionales para soportar animaciones y drag-and-drop por encima de Prototype. Ambos marcos de trabajo estan soportados en Seaside a traves del paquete "Scriptaculous".

Todas las imágenes prearmadas tienen el paquete Scriptaculous ya cargado. La última versión está disponible desde http://www.squeaksource.com/Seaside. Un demostración en linéa está disponible en http://scriptaculous.seasidehosting.st. Alternativamente, si tienes una imagen habilitada corriendo, simplemente ve a http://localhost:8080/seaside/tests/scriptaculous.

Las extensiones Scriptaculous extensions siguen el mismo enfoque que Seaside mismo — simplemente configuran objetos Smalltalk para modelar tu aplicación, y el código JavaScript necesario será generado por ti.

Veamos un simple ejemplo de como el soporte JavaScript en el cliente puede hacer a nuestra calculadora NPI compartarse más naturalmente. Actualmente cada presión de una tecla para ingresar un dígito genera un pedido para refrescar la página. Nos gustaría en su lugar, manejar la edición del display del lado del cliente, actualizando el display en la página existente.

Para manipular el display desde el código JavaScript debemos primero darle un id único. Actualizar el método de rendering de la calculadora como sigue:[10]

```
MyCalculator»renderContentOn: html
  html div id: 'keypad'; with: keypad.
  html div id: 'display'; with: display.
```

Para ser capaz de re-dibujar el display cuando un botón del teclado es presionado, el teclado necesita conocer al componente display. Agrega una variable de in-

[10]Si no has implementado el ejemplo tutorial, puedes cargar dicho ejemplo completo (PBE-SeasideRPN) desde http://www.squeaksource.com/PharoByExample y aplicar los cambios sugeridos a las clases RPN* en lugar de My*.

stancia display *al* MyKeypad, *un método inicializador* MyKeypad»setDisplay:, *y llamalo desde* MyCalculator>>initialize. *Ahora somo capaces de asignar código JavaScript a los botones, actualizando* MyKeypad»renderStackButton:callback:colSpan:on: *com sigue:*

```
MyKeypad»renderStackButton: text callback: aBlock colSpan: anInteger on: html
   html tableData
      class: 'key';
      colSpan: anInteger;
      with: [
         html anchor
            callback: aBlock;
            onClick:              "handle Javascript event"
               (html updater
                  id: 'display';
                  callback: [ :r |
                     aBlock value.
                     r render: display ];
                  return: false);
            with: [ html html: text ] ]
```

onClick: especifica un manejador de evento JavaScript. html updater retorna una instancia de SUUpdater, un objeto Smalltalk representando el objeto JavaScript Ajax.Updater (http://www.prototypejs.org/api/ajax/updater). Este objeto realiza un pedido AJAX y actualiza el contenido de un contenedor basado en el texto de una respuesta. id: indica al actualiozador que elemento XHTML DOM actualizar, en este caso los contenidos de un elmento div con id 'display'. callback: especifica un bloque que is disparado cuando el usuario presiona el botón. El argumento del bloque es un nuevo renderer r, que podemos usar para hacer render del componente display. (Nota: a pesar de que html está aún accesible, ya no es válido en el momento en que esta bloque de callback es evaluado).

Antes de hacer rendering del componente display evaluamos aBlock para realizar la acción deseada.

return: false indica al motor JavaScript no disparar el enlace de callback original, que causaría un refresco completo. Podriamos en su lugar, remover el anchor callback: original, pero asi esta calculadora funcionará aún si JavaScript está desabilitado.

Prueba la calculadora otra vez, y nota como un refresco completo de página es disparado cada vez que presionas una tecla de un dígito. (La URL de la página web is actualizada con cada presión de tecla.)

A pesar que hemos implementado el comportamiento del lado del cliente, aún no lo hemos activado. Ahora habilitaremos el manejo de eventos JavaScript.

Haz clic en el enlace Configure *en la barra de herramientas de la calculadora. Selecciona "Add Library:"* SULibrary, *haz clic en los botones* Add *y* Close.

En lugar de agregar manualmente la bibliteca, puedes también hacerlo programáticamente cuando registras la aplicación:

```
MyCalculator class»initialize
  (self registerAsApplication: 'rpn')
    addLibrary: SULibrary}}
```

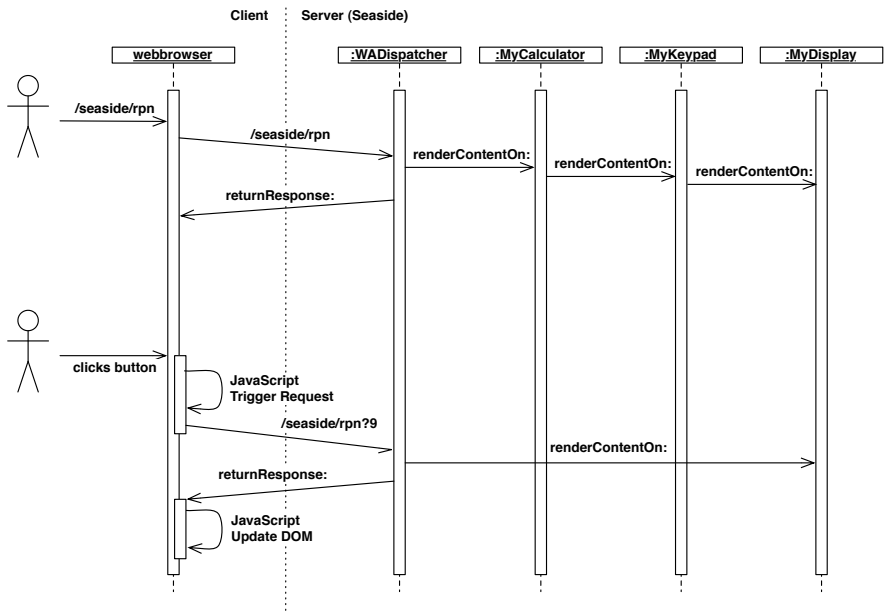


Figure 12.17: Seaside AJAX processing (simplified)

Prueba la aplicación revisada. Nota que el feedback es mucho más natural. En particular, no se genera una nueva URL con cada presión de tecla.

Bien puedes preguntar, *si, pero como funciona esto?* Figure 12.17 muestra como la aplicacion Caculadora NPI trabajaría con y sin AJAX. Básicamente, AJAX corto-circuita el rendering para actualizar *solamente* el componente display. Javascript es responsable por disparar el pedido y actualizar el correspondiente elemento DOM. Mira el código fuente generado, especialmente el código JavaScript:

```
new Ajax.Updater(
  'display',
  'http://localhost/seaside/RPN+Calculator',
```

```
  {'evalScripts': true,
   'parameters': ['_s=zcdqfonqwbeYzkza', '_k=jMORHtqr','9'].join('&')});
return false
```

Para ejemplos más avanzados, mira http://localhost:8080/seaside/tests/scriptaculous.

Pista. En caso de problemas del lado del servidor, utiliza el depurador de Smalltalk. En caso de problemas del lado del cliente utiliza FireFox (http://www.mozilla.com) con el agregado del depurador JavaScript FireBug (http://www.getfirebug.com/) habilitado.

12.9 Resumen del capítulo

- La forma más simple de comenzar es descargando el "Seaside One-Click Experience" desde http://seaside.st
- Enciende y apaga el servidor evaluando WAKom startOn: 8080 y WAKom stop.
- Restablce el usuario administrador y contraseña evaluando WADispatcherEditor initialize.
- Toggle Halos para directamente ver el código fuente de la aplicación, los objetos en ejecución, CSS y XHTML.
- Envía WAGlobalConfiguration setDeploymentMode para ocultar la barra de herramientas.
- Las aplicaciones web Seaside estan compuestas por componentes, cada uno de los cuales es una instancia de una subclase de WAComponent.
- Solo un componente raíz puede ser registrado como un componente. Este debe implementar canBeRoot del lado de la clase. Alternativamente puede registrarse asimismo como una aplicación en su método de clase initialize enviando self registerAsApplication: *application path*. Si sobreescribes description es posible retornar un nombre descriptivo de la aplicación que será mostrado en el editor de configuración.
- Para seguir el estado, un componente debe implementar el método states para retornar un arreglo de objetos cuyo estado será restaurado si el usuario hace clic en el botón "back" de su navegador.
- Un componente se renderiza asimismo implementando renderContentOn:. El argumento de este método es un *lienzo* de rendering XHTML (usualmente llamado html).

- Un componente puede hacer render de un subcomponente enviando self render: *subcomponent*.
- XHTML es generado programáticamente enviando mensajes a *pinceles*. Un pincel se obtiene enviando un mensaje, como paragraph o div, al lienzo html.
- Si envias un mensjae en cascada a un pincel que incluya el mensaje with: , entonces with: debería ser el último mensaje enviado. El mensaje with: establece los contenidos y *and* genera el resultado.
- Las acciones solo deberián aparecer en los métodos de callback. No deberias cambiar el estado de una aplicación mientras estas haciendo render de la misma.
- Puedes vincular varias formas de widgets y anclas a variables de instancia con métodos de acceso enviando el mensaje on: *variable de instancia* of: *objeto* de un pincel.
- Puedes definir el CSS para una jerarquía de componentes definiendo el método style, el cual debería retornar una cadena de caracteres conteniendo la hoja de estilos. (Para aplicaciones desplegadas, is más usual hacer referencia a una hoja de estilos localizada en una URL estática.)
- El flujo de control puede ser programado enviando x call: y, en cuyo caso el componente x será reemplazado por y hasta que y responda enviando answer: con el resultado en un mensaje de callback. El receptor de call: es usualmente self, pero puede ser en general cualquier componente visible.
- El flujo de control también puede ser especificado como una *tarea* — una instancia de una subclase de WATask. Esta debería implementar el método go, que debería llamar una seria de componentes en un workflow.
- Utiliza los métodos de WAComponents request:, inform:, confirm: y chooseFrom:caption: para interacciones básicas.
- Para prevenir que el usuario utilice el botón "back" del navegador para acceder a estados previos de ejecución de la aplicación web, puedes declarar porciones de un workflow para ser *transacciones* encerrándolas en un bloque isolate:.

Part III

Advanced Pharo

Chapter 13

Clases y metaclases

Tal como vimos en el Chapter 5, en Smalltalk todo es un objeto, y cada objeto es instancia de alguna clase. Las clases no son ninguna excepción: las clases son objetos, y dichos objetos son instancias de otras clases. Este modelo captura la esencia de la programación orientada a objetos: Es ligero, sencillo, elegante y uniforme. Sin embargo, las consecuencias de esta uniformidad pueden llegar a confundir a los neófitos. El objetivo de este capítulo es mostrar que no es complicado, "mágico" o especial: tan sólo son reglas sencillas aplicadas de manera uniforme. Siguiendo estas reglas uno siempre puede entender por qué el sistema se comporta de la manera observada.

13.1 Reglas para las clases y las metaclases

El modelo de objetos de Smalltalk está basado en un número limitado de conceptos aplicados de manera uniforme. Los diseñadores de Smalltalk aplicaron la navaja de Occam: descartar cualquier consideración que diera lugar a un sistema más complicado.

Recordemos las reglas del modelo de objetos que exploramos en Chapter 5.

Regla 1. Todo es un objeto.

Regla 2. Todo objeto es instancia de una clase.

Regla 3. Toda clase tiene una superclase.

Regla 4. Todo ocurre mediante el envío de mensajes.

Regla 5. La búsqueda de métodos sigue la cadena de herencia.

Tal y como mencionamos en la introducción a este capítulo, una consecuencia de la Rule 1 es que *las clases también son objetos*, de modo que la Rule 2 nos dice que las clases deben ser también instancias de otras clases. Llamaremos *metaclase* a la clase de una clase. Una metaclase se crea automáticamente cuando se crea una clase. La mayoría de las veces no hace falta preocuparse de las metaclases. Sin embargo, cada vez que se usa el navegador para explorar el "lado de clase" de una clase, es conveniente recordar que en realidad se está explorando una clase diferente. Una clase y su metaclase son dos clases distintas, aunque la primera sea una instancia de la segunda.

Para poder explicar las clases y las metaclases adecuadamente, necesitamos extender las reglas del Chapter 5 con las siguientes reglas adicionales.

Rule 6. Toda clase es instancia de una metaclase.

Rule 7. La jerarquía de metaclases tiene la misma estructura que la jerarquía de clases.

Rule 8. Toda metaclase hereda de Class y de Behavior.

Rule 9. Toda metaclase es instancia de Metaclass.

Rule 10. La metaclase de Metaclass es instancia de Metaclass.

Estas 10 reglas definen el modelo de objetos de Smalltalk.

Primero revisaremos brevemente las 5 reglas del Chapter 5 con un ejemplo pequeño. Después examinaremos en detalle las nuevas reglas, a partir del mismo ejemplo.

13.2 Repaso del modelo de objetos de Smalltalk

Debido a que todo es un objeto, el color azul (*blue*) también es un objeto.

```
Color blue  ⟶  Color blue
```

Cada objeto es instancia de una clase. La clase del color azul es la clase Color:

```
Color blue class  ⟶  Color
```

Es interesante observar que si asignamos un cierto valor *alpha* a un color, obtenemos una instancia de una clase diferente, a saber TranslucentColor:

```
(Color blue alpha: 0.4) class  ⟶  TranslucentColor
```

Podemos crear un morph y asignar este color translúcido como color del morph:

```
EllipseMorph new color: (Color blue alpha: 0.4); openInWorld
```

Podemos ver lo que sucede en la Figure 13.1.

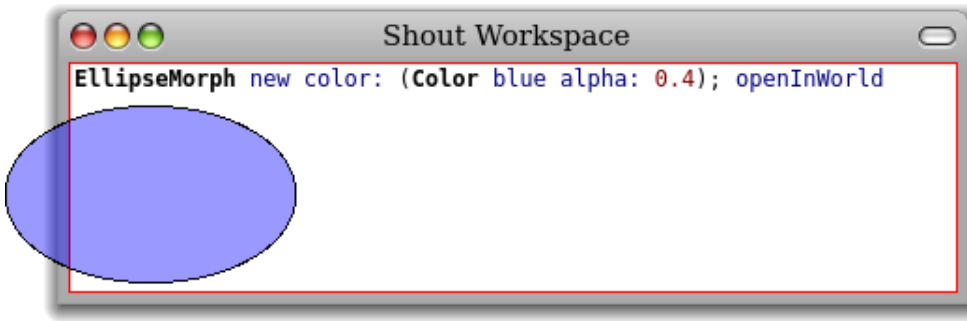


Figure 13.1: Una elípse translúcida

Por la Rule 3, cada clase tiene una superclase. La superclase de TranslucentColor es Color, y la superclase de Color es Object:

```
TranslucentColor superclass  ⟶  Color
Color superclass             ⟶  Object
```

Todo ocurre mediante el envío de mensajes (Rule 4), de modo que podemos deducir que blue es un mensaje para Color, class y alpha: son mensajes para el color azul (*blue*), openInWorld es un mensaje para un morph elipse, y superclass es un mensaje para TranslucentColor y para Color. En cada caso el receptor es un objeto, ya que todo es un objeto, pero algunos de estos objetos son también clases.

La búsqueda de método sigue la cadena de herencia (Rule 5), de modo que cuando se envía el mensaje class al resultado de Color blue alpha: 0.4, el mensaje es tratado cuando se encuentra el correspondiente método en la clase Object, como podemos ver en la Figure 13.2.

La figura captura la esencia de la relación *es-un*. Nuestro objeto azul translúcido *es-una* instancia de TranslucentColor, pero también podemos decir que *es-un* Color y que *es-un* Object, ya que responde a mensajes definidos en todas estas clases. De hecho, hay un mensaje, isKindOf:, que podemos enviar a cualquier objeto para saber si está en una relación *es-un* con una clase determinada:

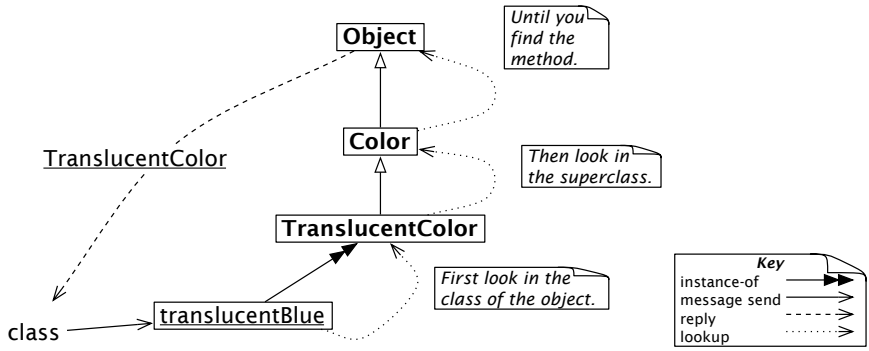


Figure 13.2: Envío de un mensaje a un color translúcido

```
translucentBlue := Color blue alpha: 0.4.
translucentBlue isKindOf: TranslucentColor  ⟶  true
translucentBlue isKindOf: Color             ⟶  true
translucentBlue isKindOf: Object            ⟶  true
```

13.3 Toda clase es instancia de una metaclase

Tal como mencionamos en Section 13.1, las clases cuyas instancias son clases son llamadas metaclases.

Las metaclases son implícitas. Las metaclases se crean automáticamente cuando se define una clase. Decimos que son *implícitas*, ya que el programador nunca tiene por qué preocuparse por ellas. Una metaclase implícita es creada por cada clase que se crea, de modo que cada metaclase tiene una sola instancia.

Mientras que los nombres de las clases normales son variables globales, las metaclases son anónimas. Sin embargo siempre podemos referirnos a las metaclases usando las clases que son sus instancias. La clase de Color, por ejemplo, es Color class, y la clase de Object es Object class:

```
Color class   ⟶  Color class
Object class  ⟶  Object class
```

La Figure 13.3 muestra que cada clase es instancia de su metaclase (anónima).

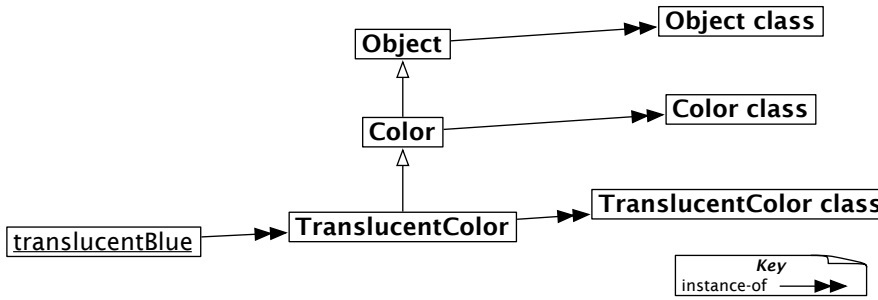


Figure 13.3: Las metaclases de Translucent y sus superclases

El hecho de que las clases sean también objetos nos facilita la obtención de información de dichas clases mediante el envío de mensajes. Veamos:

```
Color subclasses                    ⟶   {TranslucentColor}
TranslucentColor subclasses         ⟶   #()
TranslucentColor allSuperclasses    ⟶   an OrderedCollection(Color Object
    ProtoObject)
TranslucentColor instVarNames       ⟶   #('alpha')
TranslucentColor allInstVarNames    ⟶   #('rgb' 'cachedDepth' 'cachedBitPattern' '
    alpha')
TranslucentColor selectors          ⟶   an IdentitySet(#pixelValueForDepth:
    #pixelWord32 #convertToCurrentVersion:refStream: #isTransparent
    #scaledPixelValue32 #bitPatternForDepth: #storeArrayValuesOn: #setRgb:alpha:
    #alpha #isOpaque #pixelWordForDepth: #isTranslucentColor #hash #isTranslucent
    #alpha: #storeOn: #asNontranslucentColor #privateAlpha
    #balancedPatternForDepth:)
```

13.4 La jerarquía de metaclases tiene la misma estructura que la jerarquía de clases

La Rule 7 nos dice que la superclase de una metaclase no puede ser una clase arbitraria: Está constreñida por la metaclase de la superclase de la única instancia de la metaclase.

```
TranslucentColor class superclass   ⟶   Color class
TranslucentColor superclass class   ⟶   Color class
```

Esto es lo que queremos decir cuando decimos que la jerarquía de metaclases tiene la misma estructura que la jerarquía de clases; La Figure 13.4 muestra la estructura de la jerarquía de la clase TranslucentColor.

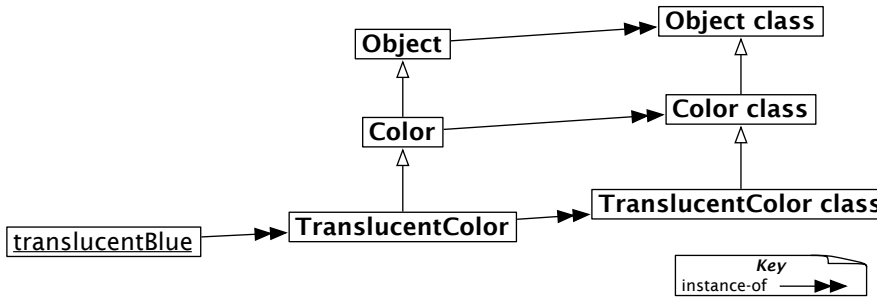


Figure 13.4: La jerarquía de metaclases tiene la misma estructura que la jerarquía de clases.

```
TranslucentColor class                          ⟶  TranslucentColor class
TranslucentColor class superclass               ⟶  Color class
TranslucentColor class superclass superclass    ⟶  Object class
```

Uniformidad entre Clases y Objetos. Es interesante tomarse un momento para ver las cosas con cierta perspectiva: No hay ninguna diferencia entre enviar un mensaje a un objeto y a una clase. En ambos casos la búsqueda del método correspondiente empieza en la clase del receptor, y posteriormente sigue la cadena de herencia.

Así pues, los mensajes enviados a clases deben seguir la cadena de herencia de las metaclases. Consideremos, por ejemplo, el método blue, que está implementado en el lado de clase de Color. Si enviamos el mensaje blue a TranslucentColor, el método correspondiente será buscado de la misma manera que cualquier otro mensaje. La búsqueda empieza en TranslucentColor class, y prosigue siguiendo la jerarquía hasta que dicho método es encontrado en Color class (ver la Figure 13.5).

```
TranslucentColor blue  ⟶  Color blue
```

Notemos que obtenemos como resultado un Color blue normal, y no uno translúcido— !'sin magia!

Así pues, vemos que hay un sólo tipo uniforme de búsqueda de métodos en Smalltalk. Las clases son objetos, y se comportan como cualquier otro objeto. Las clases tienen la capacidad de crear nuevas instancias tan sólo porque las clases pueden responder al mensaje new, y porque el método para new sabe cómo crear instancias nuevas. Normalmente, los objetos que no son

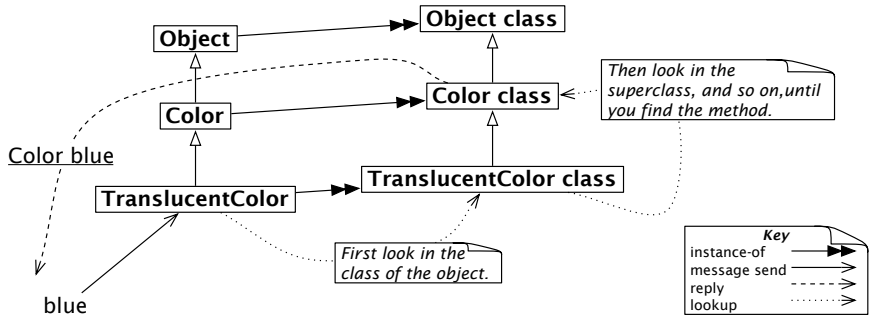


Figure 13.5: La búsqueda de los métodos correspondientes a mensajes enviados a clases es exactamente igual a la que ya conocíamos para los objetos normales

clases no comprenden este mensaje, pero si existe alguna buena razón para hacerlo no hay nada que impida al programador añadir un método new a una clase que no sea metaclase.

Como las clases son objetos, podemos inspeccionarlas.

 *Inspect* Color blue *and* Color.

Fijémonos que en un caso estamos inspeccionando una instancia de Color y en el otro caso la misma clase Color. Esto puede ser un poco confuso, porque la barra de título del inspector nombra la *clase* del objeto inspeccionado.

El inspector aplicado a Color nos permite ver la superclase, las variables de instancia, el diccionario de métodos, etc. de la clase Color, como podemos ver en la Figure 13.6.

13.5 Toda metaclase hereda de Class y de Behavior

Toda metaclase *es-una* clase, por lo tanto hereda de Class. Class a su vez hereda de sus superclases, ClassDescription y Behavior. Como todo en Smalltalk *es-un* objeto, todas estas clases eventualmente heredan de Object. Podemos ver el panorama completo en la Figure 13.7.

?'Dónde está definido new? Para entender la importancia del hecho que las metaclases hereden de Class y Behavior, ayuda preguntarse dónde está

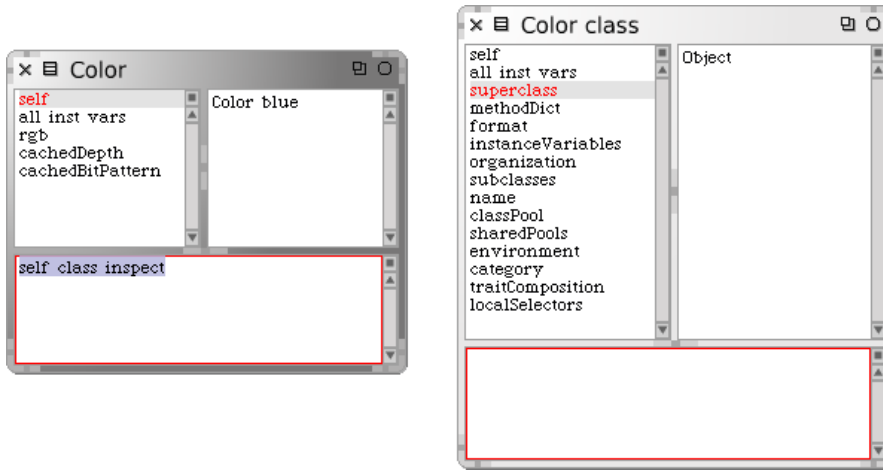


Figure 13.6: Las clases también son objetos.

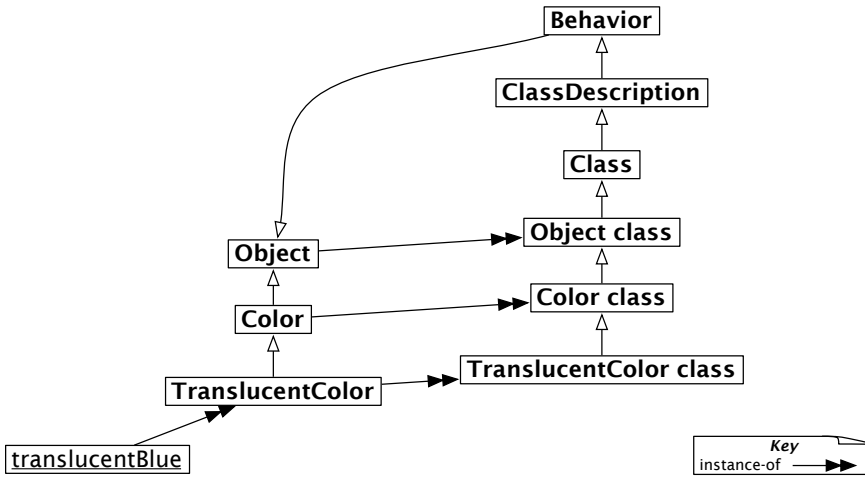


Figure 13.7: Las metaclases heredan de Class y Behavior

definido new y cómo encontrarlo. Cuando el mensaje new es enviado a una clase, este es buscado en su cadena de metaclases y finalmente en sus superclases Class, ClassDescription y Behavior como se puede ver en la Figure 13.8.

La pregunta *?'Dónde está definido* new? es crucial. new se define primero en la clase Behavior, y puede ser redefinido en sus subclases, incluyendo cualquiera de las metaclases de las clases que definimos si resulta necesario. Cuando se envía el mensaje new a una clase, este se busca, como siempre, en la metaclase de esta clase, subiendo por la cadena de superclases hasta la

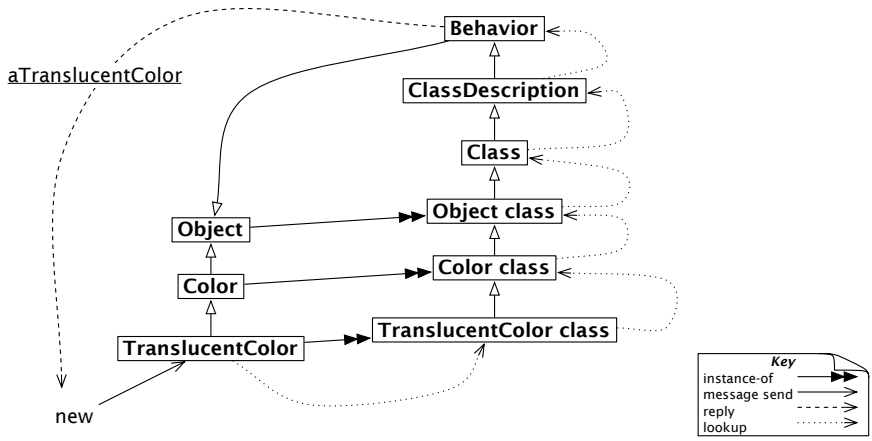


Figure 13.8: new es un mensaje normal que se busca en la cadena de metaclases.

clase Behavior, si no ha sido redefinido por el camino.

El resultado de enviar TranslucentColor new es una instancia de TranslucentColor y *no* de Behavior, !'aunque el metodo se haya encontrado en la clase Behavior! new siempre devuelve una instancia de self, la clase que recibe el mensaje, incluso si está implementado en otra clase.

```
TranslucentColor new class  ⟶  TranslucentColor  "no Behavior"
```

Un error común es buscar new en la superclase de la clase receptora del mensaje. Lo mismo sucede para new:, el mensaje estándar para crear objetos de un tamaño determinado. Por ejemplo, Array new: 4 crea un arreglo de 4 elementos. Este método no se encuentra definido en Array o ninguna de sus superclases. Hay que buscarlo en Array class y sus superclases, ya que es ahí donde la búsqueda va a empezar.

Responsabilidades de Behavior, ClassDescription y Class. Behavior proporciona el estado mínimo necesario para objetos que tienen instancias: Esto incluye un enlace a superclase, un diccionario de métodos y una descripción de las instancias (*i.e.*, representación y número). Behavior hereda de Object, así que la Behavior, y todas sus subclases, pueden comportarse como objetos.

Behavior es también la interfaz básica del compilador. Proporciona métodos para crear un diccionario de métodos, compilar métodos, crear instancias (*i.e.*, new, basicNew, new:, y basicNew:), gestionar la jerarquía de

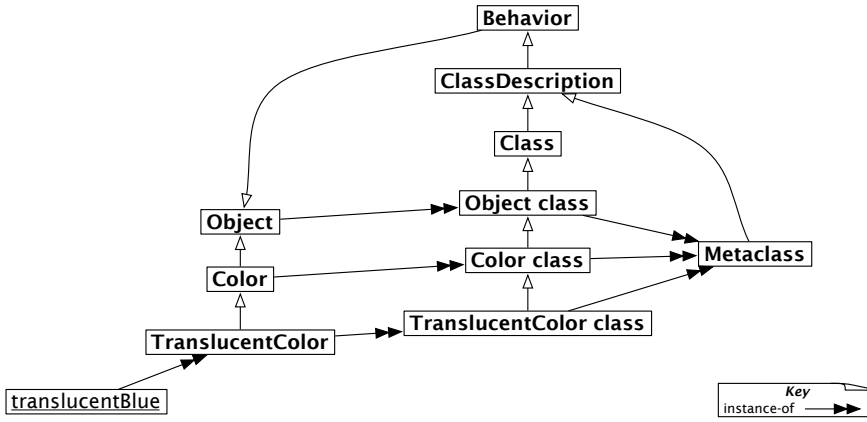


Figure 13.9: Toda metaclase es una Metaclass.

clases (*i.e.*, superclass:, addSubclass:), acceder a métodos (*i.e.*, selectors, allSelectors, compiledMethodAt:), acceder a instancias y variables (*i.e.*, allInstances, instVarNames ...), acceder a la jerarquía de clases (*i.e.*, superclass, subclasses) y obtener información diversa (*i.e.*, hasMethods, includesSelector, canUnderstand:, inheritsFrom:, isVariable).

ClassDescription es una clase abstracta que proporciona utilidades requeridas por sus dos subclases más directas, Class y Metaclass. ClassDescription añade ciertas utilidades a las utilidades básicas proporcionadas por Behavior: variables de instancia con nombre, categorización de métodos en protocolos, noción de nombre (abstracta), mantenimiento de conjuntos de cambios y el registro de los cambios, y la mayoría de los mecanismos para guardar los cambios en ficheros (*filing-out*)

Class representa el comportamiento común a todas las clases. Proporciona un nombre a la clase, métodos de compilación, almacenamiento de métodos y variables de instancia. Proporciona una representación concreta para los nombres de las variables de clase y las variables compartidas (*pool*) (addClassVarName:, addSharedPool:, initialize). Class sabe cómo crear instancias, de modo que todas las metaclases deberían heredar de Class.

13.6 Toda metaclase es instancia de Metaclass

Las metaclases también son objetos; son instancias de la clase Metaclass, como se muestra en la Figure 13.9. Las instancias de la clase Metaclass son metaclases anónimas, y cada una tiene exáctamente una instancia, que es una clase.

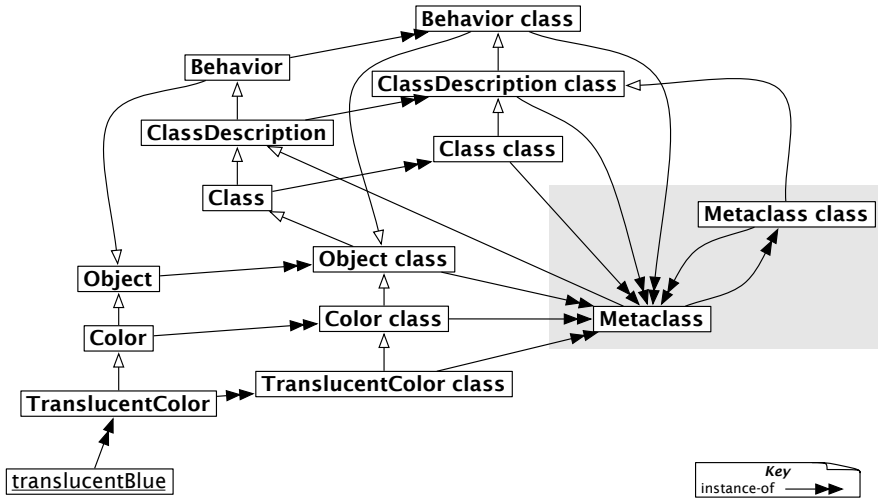


Figure 13.10: Todas las metaclases son instancias de la clase Metaclass, incluso la metaclase de Metaclass.

Metaclass representa el comportamiento común a las metclases. Proporciona métodos para la creación de instancias (subclassOf:), para crear instancias inicializadas de la única instancia de la metclase, inicialización de variables de clase, para la instancia de la metaclase, compilación de métodos e información de clase (enlaces de herencia, variables de instancia, etc.).

13.7 La metaclase de Metaclass es instancia de Metaclass

La pregunta final que queda por responder es: ?'Cual es la clase de Metaclass class?

La respuesta es sencilla: Es una metaclase, de modo que debe ser una instancia de Metaclass, como el resto de metaclases del sistema (ver la Figure 13.10).

La figura muestra como todas las metaclases son instancias de Metaclass , incluso la metaclase de Metaclass. Si comparamos las figuras 13.9 y 13.10 veremos como la jerarquía de metaclases perfectamente refleja la estructura de la jerarquía de clases, ascendiendo hasta Object class.

Los siguientes ejemplos muestran como podemos obtener información de la jerarquía de clases para demostrar que la Figure 13.10 es correcta (de hecho, veremos que hemos contado una pequeña mentira — Object class superclass ⟶ ProtoObject class, no Class. En Pharo debemos subir una superclase más para llegar a Class.)

Example 13.1: *La jerarquía de clases*

```
TranslucentColor superclass ⟶ Color
Color superclass            ⟶ Object
```

Example 13.2: *La jerarquía paralela de metaclases*

```
TranslucentColor class superclass         ⟶ Color class
Color class superclass                    ⟶ Object class
Object class superclass superclass        ⟶ Class    "NB: skip ProtoObject class"
Class superclass                          ⟶ ClassDescription
ClassDescription superclass               ⟶ Behavior
Behavior superclass                       ⟶ Object
```

Example 13.3: *Instancias de Metaclass*

```
TranslucentColor class class ⟶ Metaclass
Color class class            ⟶ Metaclass
Object class class           ⟶ Metaclass
Behavior class class         ⟶ Metaclass
```

Example 13.4: *Metaclass class es una Metaclass*

```
Metaclass class class ⟶ Metaclass
Metaclass superclass  ⟶ ClassDescription
```

13.8 Resumen del capítulo

El lector debería entender mejor como estan organizadas las clases y la relevancia de un modelo de objetos uniforme. Si resulta confuso siempre hay que recordar que la clave está en el envío de mensajes: Hay que buscar el método en la clase del receptor. Esto es así para *cualquier* receptor. Si el método no se encuentra en la clase del receptor, se busca en sus superclases.

- Toda clase es instancia de una metaclase. Las metaclases son implícitas. Una metaclase se crea automáticamente cuando se crea la clase que es su única instancia.

- La jerarquía de metaclases tiene la misma estructura que la jerarquía de clases. La búsqueda de métodos para las clases sigue el mismo procedimiento que sigue en los objetos corrientes, y asciende por la cadena de superclases de la metaclase.

- Toda metaclase hereda de Class y de Behavior. Toda clase *es una* Class . Como las metaclases también son clases, deben heredar de Class. Behavior proporciona comportamiento común a todas las entidades capaces de tener instancias.

- Toda metaclase es instancia de Metaclass. ClassDescription proporciona todo aquello que es común a Class y a Metaclass.

- La metaclase de Metaclass es instancia de Metaclass. La relación *instancia-de* forma un ciclo, de modo que Metaclass class class $\longrightarrow$ Metaclass.

Chapter 14

Reflexión

Smalltalk es un lenguaje de programación con reflexión computacional. En pocas palabras, quiere decir que los programas son capaces de «reflexionar» sobre su propia ejecución y estructura. En un sentido más técnico, esto significa que los *metaobjetos* del sistema en tiempo de ejecución pueden *reificarse* como objetos ordinarios, a los cuales se puede consultar e inspeccionar. Los metaobjetos en Smalltalk son las clases, metaclases, diccionarios de métodos, métodos compilados, la pila al momento de ejecución, y otros. Esta forma de reflexión también se denomina *introspección*, y muchos lenguajes modernos de programación la permiten.

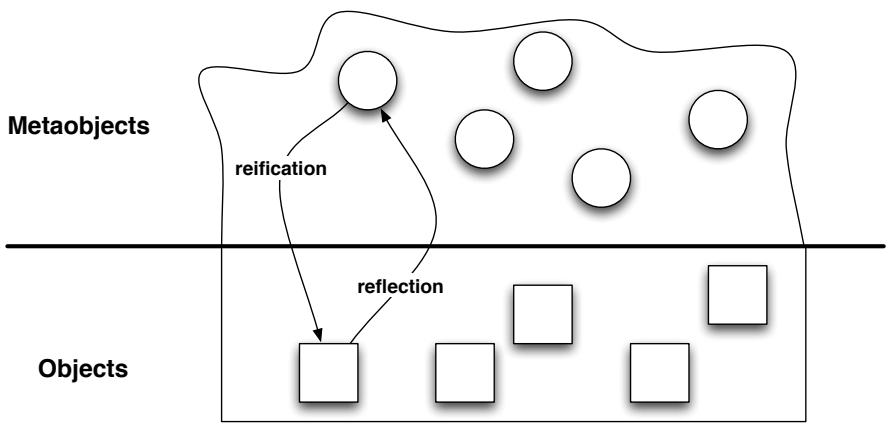


Figure 14.1: Reificación y reflexión.

En el sentido contrario, en Smalltalk resulta posible modificar los metaobjetos reificados y *reflejar* esos cambios de vuelta al sistema en ejecución (puedes verlo en la Figure 14.1). Esta actividad también se denomina *interce-*

sión. La intercesión está presente principalmente en los lenguajes de programación dinámicos mientras que los lenguajes estáticos sólo la admiten en un grado muy limitado.

Un programa que manipula otros programas (o se manipula a sí mismo) es un *metaprograma*. Para que un lenguaje de programación sea capaz de reflexión computacional debe permitir tanto la introspección como la intercesión. La introspección es la capacidad de *examinar* las estructuras de datos que definen el lenguaje, como los objetos, clases, métodos y la pila de ejecución. La intercesión es la capacidad de *modificar* dichas estructuras, o sea, de alterar la semántica del lenguaje y el comportamiento del programa desde el interior del propio programa. La *reflexión estructural* examina y modifica la estructura del sistema de tiempo de ejecución y la *reflexión de comportamiento* modifica la interpretación de esas estructuras.

En este capítulo nos enfocaremos principalmente en la reflexión estructural. Vamos a explorar muchos ejemplos prácticos que ilustrarán como Smalltalk permite la introspección y la metaprogramación.

14.1 Introspección

Mediante el inspector, puedes mirar dentro de un objeto, cambiar los valores de sus variables de instancia, e incluso enviarle mensajes.

 *Evalúa el siguiente código en un workspace:*

```
w := Workspace new.
w openLabel: 'My workspace'.
w inspect.
```

Esto va a abrir un segundo workspace y un inspector. El inspector muestra el estado interno de este nuevo espacio de trabajo, lista sus variables de instancia en la parte izquierda (dependents, contents, bindings, ...) y en la parte derecha se ve el valor de la variable de instancia seleccionada. La variable de instancia contents representa lo que se muestra en el área de texto del espacio de trabajo, así que si lo seleccionas, la parte derecha va a mostrar una cadena vacía.

 *Ahora escribe* 'hello' *en lugar de esa cadena vacía, y luego* acéptala.

El valor de la variable contents va a cambiar, pero la ventana del espacio de trabajo no se percata de ello, así que no se vuelve a regenerar lo que muestra. Para que se refresque el contenido en la ventana, evalúa self contentsChanged en la parte inferior del inspector.

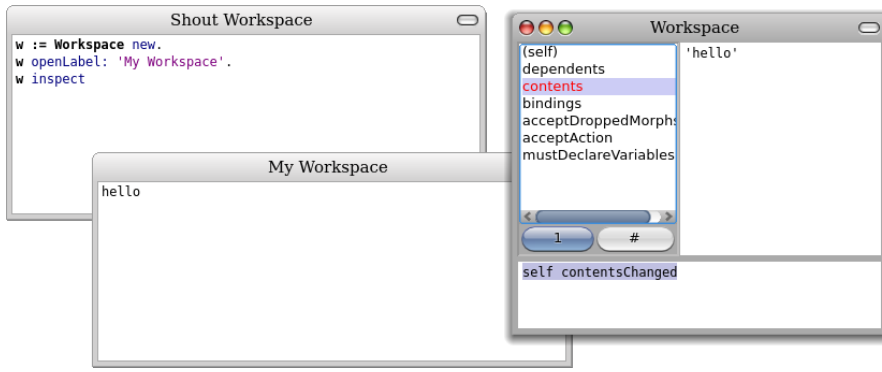


Figure 14.2: Inspección de un Workspace.

Cómo se accede a las variables de instancia

¿Cómo funciona el inspector? En Smalltalk, todas las variables de instancia son protegidas. En teoría, es imposible accederlas desde otro objeto si la clase no define sus accesores. En la práctica, el inspector puede acceder a las variables de instancia sin necesidad de accesores, porque usa las capacidades de reflexión computacional de Smalltalk. En Smalltalk, las clases definen variables de instancia o bien mediante un nombre o bien mediante un índice numérico. El inspector usa métodos definidos en la clase Object para accederlas: instVarAt: *index* y instVarNamed: *aString* se usan para acceder al valor de la variable de instancia en la posición *index* o aquella cuyo nombre es *aString*, respectivamente; para asignar nuevos valores a esas variables de instancia utiliza instVarAt:put: y instVarNamed:put:.

Por ejemplo, puedes cambiar el valor asociado a w en el primer espacio de trabajo mediante la evaluación de:

```
w instVarNamed: 'contents' put: 'howdy'; contentsChanged
```

> *Precaución:* Aunque estos métodos resultan útiles para construir herramientas de desarrollo, es una mala idea usarlos para desarrollar aplicaciones convencionales: estos métodos de reflexión computacional violan los límites de la encapsulación de los objetos y por lo tanto pueden hacer que tu código sea mucho más difícil de comprender y mantener.

Tanto instVarAt: como instVarAt:put: son métodos primitivos, lo que significa que están implementados como operaciones primitivas de la máquina vir-

tual de Pharo. Si revisas el código de esos métodos verás la sintaxis especial pragma <primitive: N> donde N es un entero.

```
Object»instVarAt: index
    "Primitive. Answer a fixed variable in an object. ..."
    <primitive: 73>
    "Access beyond fixed variables."
    ↑self basicAt: index − self class instSize
```

En el caso más usual no se ejecuta el código que sigue a la invocación de la primitiva. Sólo se ejecuta si la primitiva falla. En este caso específico, si tratamos de acceder una variable que no existe entonces se ejecuta el código que sigue a la primitiva. De este modo se permite también que el debugger se inicie sobre métodos primitivos. Aunque es posible modificar el código de los métodos primitivos, debes entender que es una actividad de alto riesgo para la estabilidad de tu sistema Pharo.

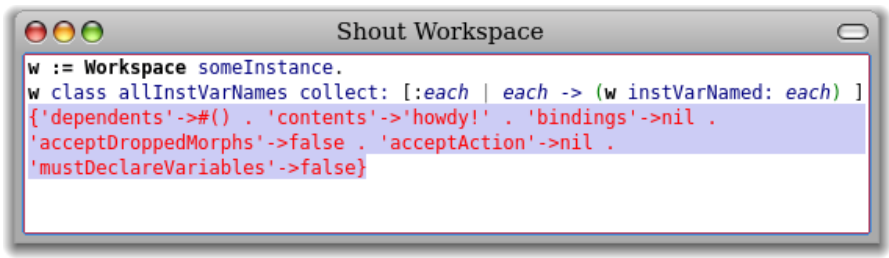


Figure 14.3: Vista de todas las variables de instancia de un Workspace.

La Figure 14.3 muestra cómo pueden verse los valores de las variables de instancia de una instancia arbitraria (w) de la clase Workspace. El método allInstVarNames retorna todos los nombres de las variables de instancia de una clase dada.

De manera similar, es posible conseguir las instancias que tienen ciertas propiedades específicas. Por ejemplo, para obtener todas las instancias de la clase SketchMorph cuya variable de instancia owner está asociada al morph mundo (o sea, las imágenes que se muestran en este momento), prueba la siguiente expresión:

```
SketchMorph allInstances select: [:c | (c instVarNamed: 'owner') isWorldMorph]
```

Cómo recorrer las variables de instancia

Consideremos el mensaje instanceVariableValues, que devuelve una colección de todos los valores de variables de instancia definidos en dicha clase, sin tener en cuenta las variables de instancia heredadas. Por ejemplo:

```
(1@2) instanceVariableValues  ⟶  an OrderedCollection(1 2)
```

El método está implementado en Object como se muestra a continuación:

```
Object»instanceVariableValues
    "Answer a collection whose elements are the values of those instance variables of
    the receiver which were added by the receiver's class."
    | c |
    c := OrderedCollection new.
    self class superclass instSize + 1
        to: self class instSize
        do: [ :i | c add: (self instVarAt: i)].
    ↑ c
```

Este método itera sobre los índices de las variables de instancia que están definidas en la clase; el primer valor corresponde al primer índice no usado por la superclase. (El método instSize retorna la cantidad de variables de instancia nominadas que define una clase)

Cómo consultar clases e interfaces

Tods las herramientas de desarrollo de Pharo (code browser, debugger, inspector...) utilizan las características de reflexión computacional que hemos visto hasta ahora.

Se muestran a continuación otros mensajes que pueden resultar útiles para construir herramientas de desarrollo:

isKindOf: *aClass* retorna true si el receptor es una instancia de *aClass* o una de sus superclases. Por ejemplo:

```
1.5 class                 ⟶  Float
1.5 isKindOf: Number      ⟶  true
1.5 isKindOf: Integer     ⟶  false
```

respondsTo: *aSymbol* returna true si el receptor tiene un método cuyo selector es *aSymbol*. Por ejemplo:

```
1.5 respondsTo: #floor          ⟶  true   "pues Number implementa floor"
1.5 floor                       ⟶  1
Exception respondsTo: #,        ⟶  true   "las clases de exception se pueden agrupar"
```

> *Precaución:* Aunque estas características resultan especialmente útiles para escribir herramientas de desarrollo, normalmente no son apropiadas para escribir aplicaciones típicas. Cuando uno consulta a un objeto para saber su clase o para saber a cuáles mensajes responde, estamos en presencia de señales típicas que apuntan a problemas de diseño, pues estas consultas violan el principio de encapsulación. Las herramientas de desarrollo, sin embargo, no son aplicaciones normales, pues su dominio es el del software en sí mismo. Por ello, estas herramientas tienen derecho a indagar profundamente en los detalles internos del código.

Métricas de código

Veamos cómo podemos usar las características de introspección de Smalltalk para extraer rápidamente algunas métricas de código. Las métricas de código miden aspectos tales como la profundidad de la jerarquía de herencia, la cantidad directa e indirecta de subclases, la cantidad de métodos o de variables de instancia en cada clase, o la cantidad de métodos o variables de instancia definidos localmente. A continuación se muestran unas métricas para la clase Morph, que es la superclase de todos los objetos gráficos en Pharo, y que revelan que se trata de una clase enorme, y que es la raíz de una enorme jerarquía. ¡Tal vez está necesitada de un poco de refactoring!

```
Morph allSuperclasses size.      ⟶     2 "profundidad de la herencia"
Morph allSelectors size.         ⟶  1378 "cantidad de metodos"
Morph allInstVarNames size.      ⟶     6 "cantidad de variables de instancia"
Morph selectors size.            ⟶   998 "cantidad de nuevos metodos"
Morph instVarNames size.         ⟶     6 "cantidad de nuevas variables"
Morph subclasses size.           ⟶    45 "cantidad de subclases directas"
Morph allSubclasses size.        ⟶   326 "total de subclases"
Morph linesOfCode.               ⟶  5968 "cantidad total de lineas de codigo"
```

Una de las métricas más interesantes en el dominio de los lenguajes orientados a objetos es la cantidad de métodos que extienden métodos heredados desde la superclase. Ella nos informa de la relación entre la clase y sus superclases. En las siguientes secciones veremos cómo aprovechar nuestro conocimiento de la estructura de tiempo de ejecución para responder estas cuestiones.

14.2 Cómo navegar por el código

En Smalltalk, todo es un objeto. En particular, las clases son objetos que nos proporcionan mecanismos útiles para navegar entre sus instancias. La mayoría de los mensajes que vamos a estudiar están implementados en Behavior y por lo tanto todas las clases responden a ellos.

Como ya vimos antes, puedes obtener una instancia de una clase dada si le envías el mensaje #someInstance.

```
Point someInstance    ⟶    0@0
```

Además, puedes obtener todas las instancias con #allInstances, y la cantidad de instancias activas en memoria con #instanceCount.

```
ByteString allInstances          ⟶    #('collection' 'position'  ...)
ByteString instanceCount         ⟶    104565
String allSubInstances size      ⟶    101675
```

Estas características pueden resultar muy útiles al depurar una aplicación, porque puedes solicitar a una clase que enumere aquellos de sus métodos que exhiben ciertas propiedades.

- whichSelectorsAccess: retorna la lista de todos los selectores de métodos que leen o escriben la variable de instancia nombrada por el argumento
- whichSelectorsStoreInto: retorna los selectores de métodos que modifican el valor de una variable de instancia dada
- whichSelectorsReferTo: retorna los selectores de métodos que envían un dado mensaje
- crossReference asocia cada mensaje con el conjunto de métodos que lo envían.

```
Point whichSelectorsAccess: 'x'      ⟶    an IdentitySet(#'\\' #= #scaleBy: ...)
Point whichSelectorsStoreInto: 'x'   ⟶    an IdentitySet(#setX:setY: ...)
Point whichSelectorsReferTo: #+      ⟶    an IdentitySet(#rotateBy:about: ...)
Point crossReference    ⟶    an Array(
        an Array('*' an IdentitySet(#rotateBy:about: ...))
        an Array('+' an IdentitySet(#rotateBy:about: ...))
        ...)
```

Los siguientes mensajes tienen en cuenta a la herencia:

- whichClassIncludesSelector: retorna la superclase que implementa el mensaje dado

- unreferencedInstanceVariables retorna la lista de variables de instancia que no se usan en la clase del receptor ni en sus subclases.

```
Rectangle whichClassIncludesSelector: #inspect  ⟶  Object
Rectangle unreferencedInstanceVariables         ⟶  #()
```

SystemNavigation es una fachada [facade] que incluye varios métodos útiles para consultar y navegar a través del código fuente del sistema. SystemNavigation default retorna una instancia que puedes usar para navegar por el sistema. Por ejemplo:

```
SystemNavigation default allClassesImplementing: #yourself  ⟶  {Object}
```

Los siguientes mensajes se explican solos:

```
SystemNavigation default allSentMessages size          ⟶  24930
SystemNavigation default allUnsentMessages size        ⟶  6431
SystemNavigation default allUnimplementedCalls size    ⟶  270
```

Que un mensaje esté implementado y que nadie lo envíe no es necesariamente una señal de un mensaje inútil, pues puede ser que se lo envíe implícitamente (por ejemplo, mediante perform:). Los mensajes que se envían pero no están implementados son más problemáticos, porque los métodos que envían esos mensajes van a fallar en tiempo de ejecución. Son la señal de una implementación sin terminar, APIs obsoletas o bibliotecas faltantes.

SystemNavigation default allCallsOn: #Point retorna todos los mensajes enviados explícitamente a Point como receptor.

Todas estas características están integradas en el entorno de programación de Pharo, en particular en los navegadores de código. Como ya conocerás, hay atajos de teclado convenientes para navegar por todos los implementors (CMD−m) y senders (CMD−n) de un mensaje dado. Lo que tal vez no resulte tan conocido es que hay muchas consultas preparadas que están implementadas como métodos de la clase SystemNavigation en el protocolo *browsing*. Por ejemplo, puedes encontrar programáticamente todos los implementadores del mensaje ifTrue: mediante la evaluación de:

```
SystemNavigation default browseAllImplementorsOf: #ifTrue:
```

Los métodos browseAllSelect: y browseMethodsWithSourceString: son particularmente útiles. A continuación se muestran dos manera diferentes de revisar todos los métodos en el sistema que envían mensajes a super (la primera es más bien por fuerza bruta; la segunda es mejor y elimina algunos falsos positivos):

```
SystemNavigation default browseMethodsWithSourceString: 'super'.
SystemNavigation default browseAllSelect: [:method | method sendsToSuper ].
```

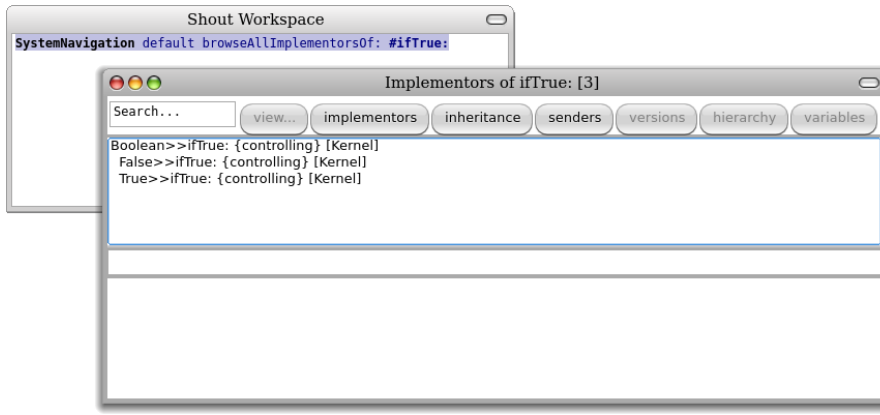


Figure 14.4: Implementadores de #ifTrue:.

14.3 Clases, dicionarios de métodos y métodos

Como las clases son objetos, podemos inspeccionarlas o explorarlas como a cualquier otro objeto.

Evalúa Point explore.

En Figure 14.5, el explorer muestra la estructura de la clase Point. Puedes ver que la clase almacena sus métodos en un diccionario, indexados por su selector. El selector #* apunta al bytecode descompilado de Point»*.

Vamos a considerar la relación entre clases y métodos. En Figure 14.6 vemos que las clases y metaclases tienen en común a la superclase Behavior. Aquí es donde se define a new, entre otros métodos para las clases. Cada clase tiene un diccionario de métodos, el cual mapea selectores de métodos en métodos compilados. Cada método compilado conoce la clase en la cual está instalado. En Figure 14.5 podemos ver incluso que esto se almacena en una asociacion en literal5.

Podemos aprovechar las relaciones entre clases y métodos para realizar consultas sobre el sistema. Por ejemplo, para descubrir cuáles métodos han sido recientemente incorporados por una dada clase (o sea, métodos que no sobrecargan métodos de las superclases) podemos navegar desde la clase al dicionario de métodos como se muestra a continuación:

```
[:aClass| aClass methodDict keys select: [:aMethod |
  (aClass superclass canUnderstand: aMethod) not ]] value: SmallInteger
    ⟶    an IdentitySet(#threeDigitName #printStringBase:nDigits: ...)
```

El método compilado no almacena solamente los bytecodes del método.

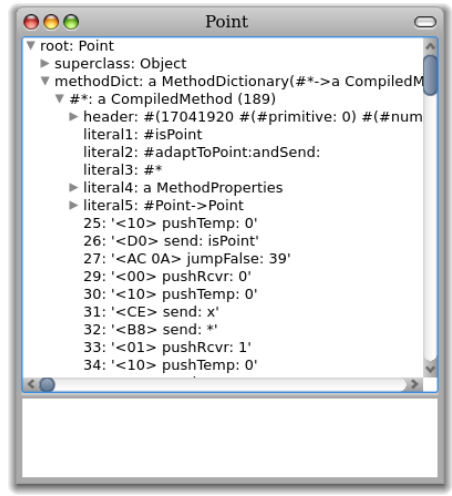


Figure 14.5: Explorer:la clase Point y el bytecode de su método #*.

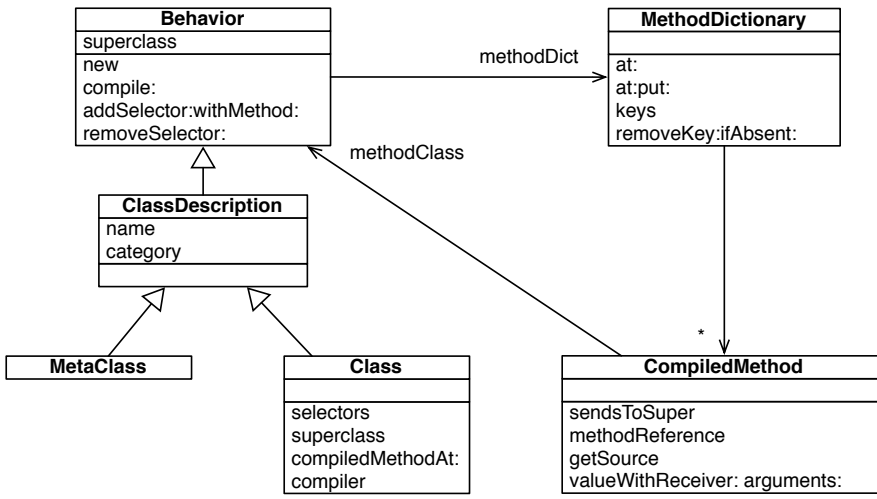


Figure 14.6: Clases, diccionarios de métodos y métodos compilados

Es un objeto que proporciona muchos métodos útiles para consultar el sistema. Uno de tales métodos es isAbstract (que nos indica si el método envía subclassResponsibility). Podemos usarlo para identificar todos los métodos abstractos de una clase abstracta.

```
[:aClass| aClass methodDict keys select: [:aMethod |
 (aClass>>aMethod) isAbstract ]] value: Number
    ⟶    an IdentitySet(#storeOn:base: #printOn:base: #+ #− #* #/ ...)
```

Advierte como este código envía el mensaje >> a una clase para obtener el método compilado de un selector dado.

Para pasear por los sends a super dentro de una jerarquía dada, por ejemplo dentro de la jerarquía de Collection, podemos realizar una consulta más sofisticada:

```
class := Collection.
SystemNavigation default
 browseMessageList: (class withAllSubclasses gather: [:each |
  each methodDict associations
   select: [:assoc | assoc value sendsToSuper]
   thenCollect: [:assoc | MethodReference class: each selector: assoc key]])
 name: 'Supersends of ' , class name , ' and its subclasses'
```

Nota como navegamos desde las clases a los diccionarios de métodos, hasta los métodos compilados para identificar los métodos en los cuales estamos interesados. El MethodReference es un proxy liviano para un método compilado; muchas herramientas utilizan estos proxys. Hay un método de facilitación llamado CompiledMethod»methodReference que retorna la referencia de método de un método compilado.

```
(Object>>#=) methodReference methodSymbol    ⟶    #=
```

14.4 Entornos de navegación

Aún cuando SystemNavigation ofrece algunas maneras programáticas que resultan útiles a la hora de consultar y navegar por el código del sistema mediante, hay otro método mejor. El Refactoring Browser, que está integrado en Pharo, proporciona tanto una vía interactiva como una programática para realizar consultas complejas.

Supongamos que queremos averiguar cuáles de los métodos dentro de la jerarquía Collection envían un mensaje a super que sea diferente del selector del método. Esto se considera un mal olor de código [code smell], porque el send a super normalmente debería poder reemplazarse por un send a self. (Piénsalo por un momento: la *necesidad* de super se debe a que lo utilizamos para extender el método que estamos sobrecargando; ¡todos los otros métodos heredados se pueden acceder mediante mensajes a self!

El navegador de refactoreo [refactoring browser] nos proporciona una manera elegante de restringir la consulta sólo a las clases y métodos en los

cuales estamos interesados.

En Figure 14.7 podemos ver que hay 19 métodos con esas caracerísticas dentro de la jerarquía Collection, entre los cuales está Collection»printNameOn:, que envía el mensaje super printOn:.

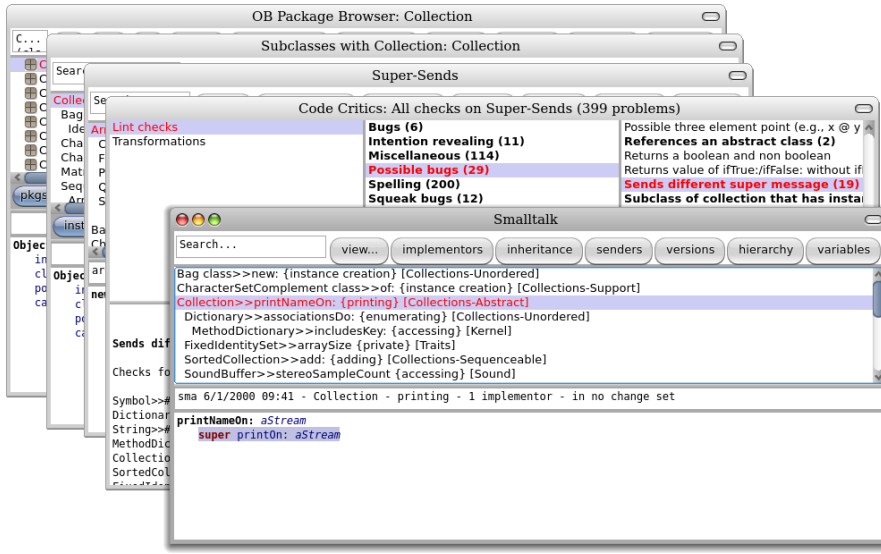


Figure 14.7: Cómo encontrar métodos que envían un mensaje a super diferente.

Los entornos de navegación también se pueden crear programáticamente. A continuación, por ejemplo, vamos a crear un nuevo BrowserEnvironment para Collection y sus subclases, seleccionaremos los métodos que envían mensajes a super y abriremos el entorno resultante.

```
((BrowserEnvironment new forClasses: (Collection withAllSubclasses))
      selectMethods: [:method | method sendsToSuper])
      label: 'Collection methods sending super';
      open.
```

Puedes notar cómo esta forma es considerablemente más compacta que la anterior, cuando usamos SystemNavigation.

Por último, podemos buscar aquellos métodos que envían un super diferente en forma programática como se muestra a continuación:

```
((BrowserEnvironment new forClasses: (Collection withAllSubclasses))
    selectMethods: [:method |
        method sendsToSuper
        and: [(method parseTree superMessages includes: method selector) not]])
    label: 'Collection methods sending different super';
    open
```

En este caso solicitamos a cada método compilado su árbol sintáctico (del Refactoring Browser) para ver cuáles de los mensajes a super difieren del selector del método. Para ver qué se puede hacer con los árboles sintácticos, echa una mirada al protocolo *querying* de la clase RBProgramNode.

14.5 Cómo acceder al contexto de tiempo de ejecución

Ya hemos visto como las capacidades de reflexión de Smalltalk nos permiten consultar y explorar objetos, clases y métodos. ¿Y qué podemos hacer con el entorno de ejecución?

Contexto de los métodos

De hecho, el contexto de tiempo de ejecución de un método que está corriendo reside en la máquina virtual — ¡ni siquiera está en la imagen! Por otro lado, el debugger obviamente tiene acceso a esta información, y podemos explorar con facilidad el contexto de tiempo de ejecución, al igual que hacemos con cualquier otro objeto. ¿Cómo puede ser esto posible?

En realidad, no hay nada de magia en el depurador. El secreto es la seudovariable thisContext, que ya hemos visto antes, aunque sólo de pasada. Cuando se referencia a thisContext dentro de un método en ejecución, se reifica el contexto completo de ejecución de dicho método y a ese objeto se lo deja accesible a la imagen en la forma de una serie de objetos MethodContext encadenados.

Podemos hacer unos experimentos on este mecanismo.

 *Cambia la definición de* Integer»factorial*, para ello inserta la expresión subrayada que se muestra más abajo:*

```
Integer»factorial
    "Answer the factorial of the receiver."
    self = 0 ifTrue: [thisContext explore. self halt. ↑ 1].
    self > 0 ifTrue: [↑ self * (self − 1) factorial].
    self error: 'Not valid for negative integers'
```

Ahora evalúa 3 factorial *en un espacio de trabajo. Deberías ver tanto una ventana de depurador como un explorador, tal como se muestra en Figure 14.8.*

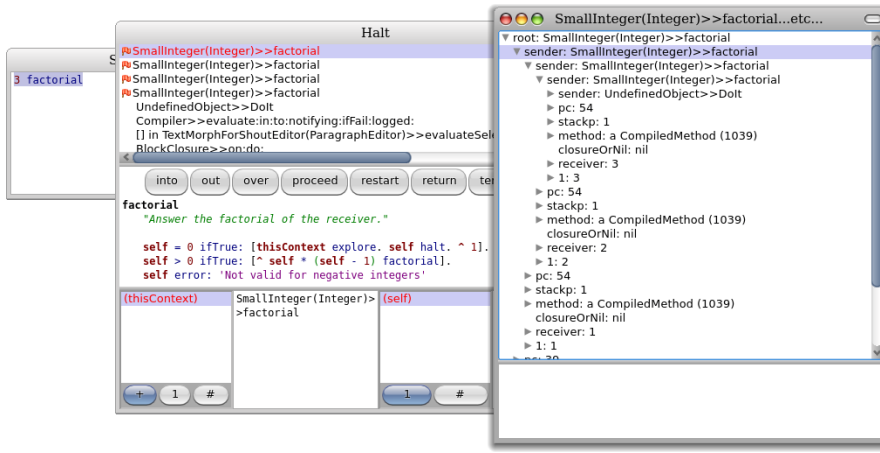


Figure 14.8: Cómo explorar thisContext.

¡Bienvenido al depurador del pobre! Si navegas ahora por la clase del objeto explorado (evalúa self browse en el panel inferior del explorador) vas a descubrir que es una instancia de la clase MethodContext, tal como cada uno de los sender en la cadena.

thisContext no está pensado para usarse en la programación cotidiana, pero es esencial en la implementación de herramientas tales como los depuradores, y para acceder a la información de la pila de llamadas. Puedes evaluar la siguiente expresión para descubrir cuáles métodos utilizan a thisContext :

```
SystemNavigation default browseMethodsWithSourceString: 'thisContext'
```

Como puedes ver, una de las aplicaciones más comunes es para descubrir el remitente de un mensaje. A continuación se muestra una aplicación típica:

```
Object»subclassResponsibility
    "This message sets up a framework for the behavior of the class' subclasses.
    Announce that the subclass should have implemented this message."

    self error: 'My subclass should have overridden ', thisContext sender selector
    printString
```

Por convención, los métodos en Smalltalk que envían self subclassResponsibility se consideran abstractos. Ahora bien, ¿cómo hace Object»subclassResponsibility para proporcionar un mensaje de error útil que

indique cuál de los métodos abstractos ha sido invocado? Muy simple, preguntando cuál es el thisContext del remitente.

Puntos de interrupción inteligentes

La manera a la Smalltalk de crear un punto de interrupción [breakpoint] consiste en evaluar self halt en una posición interesante de un método. Esto hace que thisContext se reifique y que se abra una ventana de debugger sobre el punto de interrupción. Desafortunadamente, esto plantea problemas para los métodos intensamente usados en el sistema.

Supongamos, por ejemplo, que deseamos explorar la ejecución de OrderedCollection»add:. Resulta problemático configurar un punto de interrupción en este método.

 *Toma una imagen* fresca *y configura los siguientes puntos de interrupción:*

```
OrderedCollection»add: newObject
    self halt.
    ↑self addLast: newObject
```

¡La imagen se congela de inmediato! Ni siquiera aparece una ventana del navegador. El problema se aclara cuando comprendemos que (I) el sistema usa a OrderedCollection»add: en muchas partes, y por lo tanto el punto de interrupción se dispara un momento después de aceptar el cambio, y además (II) *el propio depurador* envía add: a una instancia de OrderedCollection, lo cual impide que se abra la ventana del depurador. Lo que necesitamos es una forma de *hacer un halt condicional* sólo si nos encontramos en un contexto que sea de nuestro interés. Esto es exactamente lo que ofrece Object»haltIf:.

Supongamos que sólo queremos un halt si add: es enviado desde, digamos, el contexto de OrderedCollectionTest»testAdd.

 *Levanta una imagen fresca de nuevo, y configura el siguiente punto de interrupción:*

```
OrderedCollection»add: newObject
    self haltIf: #testAdd.
    ↑self addLast: newObject
```

Esta vez la imagen no se congela. Prueba de correr el OrderedCollectionTest. (Lo puedes encontrar en la categoría *CollectionsTests-Sequenceable*.)

¿Cómo funciona esto? Echemos una mirada a Object»haltIf::

```
Object»haltIf: condition
    | cntxt |
    condition isSymbol ifTrue: [
```

```
        "only halt if a method with selector symbol is in callchain"
        cntxt := thisContext.
        [cntxt sender isNil] whileFalse: [
            cntxt := cntxt sender.
            (cntxt selector = condition) ifTrue: [Halt signal]. ].
        ↑self.
    ].
    ...
```

Para cada contexto, desde thisContext, haltIf: hacia arriba en la pila de ejecución, se comprueba si el nombre del método que hace la llamada es el mismo símbolo que el recibido como parámetro. Si se da el caso, se levanta una excepción, que de manera predeterminada apremia al depurador.

También se puede proporcionar un valor booleano o un bloque booleano como argumento de haltIf:, pero esos casos son sencillos y no utilizan a thisContext.

14.6 Cómo interceptar los mensajes no definidos

Hasta ahora utilizamos las capacidades de reflexión computacional de Smalltalk principalmente para consultar y explorar objetos, clases, métodos y la pila de tiempo de ejecución. Ahora veremos cómo usar nuestro conocimiento de la estructua del sistema Smalltalk para interceptar mensajes y modificar comportamiento al momento de ejecución.

Cuando un objeto recibe un mensaje, revisa primero en el diccionario de métodos de su clase para encontrar el método correspondiente que responda al mensaje. Si no existe un método tal, continúa revisando hacia arriba en la jeraquía de clases, hasta que llega a Object. Si no se encuentra ningún método para ese mensaje, el objeto *se envía a sí mismo* el mensaje doesNotUnderstand: con el selector del mensaje como su argumento. El proceso se inicia de nuevo, hasta que se encuentra a Object»doesNotUnderstand: y se lanza el depurador.

¿Pero qué sucede si doesNotUnderstand: está sobrecargado en una de las subclases de Object en la ruta que se revisa? Como veremos, ésta es una manera conveniente de realizar ciertas clases de comportamiento muy dinámico. Un objeto que no responde [does not understand] a un mensaje puede sobrecargar doesNotUnderstand: para proporcionar una estrategia alternativa para responder al mensaje de marras.

Hay dos aplicaciones muy comunes de esta técnica: (1) para implementar proxies livianos [lightweight proxies] para los objetos, y (2) para compilar o cargar código faltante.

Proxies livianos

En el primer caso, vamos a a introducir un «objeto mínimo » [minimal object] para que actúe como proxy a favor de un objeto existente. Como el proxy prácticamente no implementa métodos propios, cualquier mensaje que se le envíe será interceptado por doesNotUnderstand:. Al implementar este mensaje, el proxy puede realizar una acción especial antes de delegar el mensaje al objeto real del cual es un proxy.

Veamos ahora cómo puede implementarse esta idea[1].

Definimos el LoggingProxy como se muestra a continuación:

```
ProtoObject subclass: #LoggingProxy
    instanceVariableNames: 'subject invocationCount'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE−Reflection'
```

Creamos la subclase de ProtoObject, en lugar de hacerlo desde Object porque no queremos que nuestro proxy herede ¡más de 400 métodos! de Object.

```
Object methodDict size    ⟶    408
```

Nuestro proxy tiene dos variables de instancia: subject es el sujeto de quien es proxy, y count con la cantidad de mensajes que ha interceptado. Inicializamos las dos variables de instancia y proporcionamos un accesor para la cuenta de mensajes. Inicialmente la variable subject apunta al proxy mismo.

```
LoggingProxy»initialize
    invocationCount := 0.
    subject := self.
```

```
LoggingProxy»invocationCount
    ↑ invocationCount
```

Ahora interceptamos todos los mensajes no definidos, los escribimos en el Transcript, actualizamos la cuenta de mensajes, y reenviamos el mensaje al sujeto real.

```
LoggingProxy»doesNotUnderstand: aMessage
    Transcript show: 'performing ', aMessage printString; cr.
    invocationCount := invocationCount + 1.
    ↑ aMessage sendTo: subject
```

[1] También puedes cargar *PBE-Reflection* desde http://www.squeaksource.com/PharoByExample/

Aquí va un poquito de magia. Creamos un nuevo objeto Point y un nuevo LoggingProxy, y luego hacemos que el proxy se transforme (become:) en el objeto punto:

```
point := 1@2.
LoggingProxy new become: point.
```

Esto tiene el efecto de intercambiar todas las referencias en la imagen que apuntaban al punto y ahora apuntan al proxy, y viceversa. Lo más importante: ¡ahora la variable de instancia subject va a referirse al punto!

```
point invocationCount   ⟶   0
point + (3@4)           ⟶   4@6
point invocationCount   ⟶   1
```

Esto funciona muy bien en la mayoría de los casos, pero hay algunas desventajas:

```
point class   ⟶   LoggingProxy
```

Resulta curioso que el método class no está implementado ni siquiera en ProtoObject sino en Object, ¡del cual no hereda LoggingProxy! La respuesta a este acertijo es que class nunca se envía como mensaje, sino que la respuesta la da directamente la máquina virtual.[2]

Aún si podemos ignorar tales envíos especiales de mensajes, exite otro problema fundamental que no se puede sobrellevar mediante este enfoque: los envíos a self no pueden interceptarse:

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount   ⟶   0
point rect: (3@4)       ⟶   1@2 corner: 3@4
point invocationCount   ⟶   1
```

Nuestro proxy ha sido estafado por dos envíos a self en el método rect:

```
Point»rect: aPoint
    ↑ Rectangle  origin: (self min: aPoint) corner: (self max: aPoint)
```

[2] yourself tampoco se envía, verdaderamente. Hay otros mensajes que pueden ser interpretados directamente por la máquina virtual, que dependen del receptor, entre ellos están: + − < > <= >= = ~= * / == @ bitShift: // bitAnd: bitOr: at: at:put: size next nextPut: atEnd blockCopy: value value: do: new new: x y. Hay selectores que nunca se envían, porque el compilador los compila en línea y los transforma a bytecodes de comparación y salto: ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue: and: or: whileFalse: whileTrue: whileFalse whileTrue to:do: to:by:do: caseOf: caseOf:otherwise: ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil: Si sobrecargamos mustBeBoolean en el receptor se pueden interceptar los intentos de enviar esos mensajes a objetos que no son booleanos y continuar la ejecución sobre un valor booleano válido. Otra forma de hacerlo es capturar la excepción NonBooleanReceiver

Aún cuando los proxies pueden interceptar mensajes mediante el uso de esta técnica, uno debe ser conciente de las limitaciones inherentes al utilizar un proxy. En Section 14.7 veremos otro enfoque, más general, para interceptar mensajes.

Cómo generar los métodos faltantes

La otra aplicación más popular de la intercepción de mensajes no definidos es la carga o generación dinámica de los métodos faltantes. Considere una biblioteca muy extensa de clases que tiene muchos métodos. En lugar de cargar la biblioteca completa, podemos cargar un empalme para cada clase de la biblioteca. Los empalmes saben dónde encontrar el código fuente de todos sus métodos. Los empalmes simplemente interceptan todos los mensajes no entendidos y cargan dinámicamente los métodos faltantes a medida que son necesarios. En algún momento posterior se puede desactivar este comportamiento, y grabar el código cargado como el mínimo subconjunto necesario para la aplicación cliente.

Vamos a estudiar una variante simple de esta técnica donde tenemos una clase que agrega automáticamente los accesores para sus variables de instancia cuando sean necesarios:

```
DynamicAcccessors»doesNotUnderstand: aMessage
    | messageName |
    messageName := aMessage selector asString.
    (self class instVarNames includes: messageName)
        ifTrue: [
            self class compile: messageName, String cr, '↑ ', messageName.
            ↑ aMessage sendTo: self ].
    ↑ super doesNotUnderstand: aMessage
```

Cualquier mensaje no definido se intercepta aquí. Si existe una variable de instancia con el mismo nombre del mensaje, entonces solicitamos a nuestra clase que compile un accesor para esa variable de instancia y reenviamos el mensaje.

Supongamos que la clase DynamicAcccessors tiene una variable de instancia (sin inicializar) llamada x pero la misma no tiene un accesor predefinido. Entonces el código siguiente va a generar el accesor dinámicamente y a recuperar su valor:

```
myDA := DynamicAccessors new.
myDA x    ⟶    nil
```

Vayamos paso a paso para ver lo que sucede la primera vez que se envía el mensaje x a nuestro objeto (mira Figure 14.9).

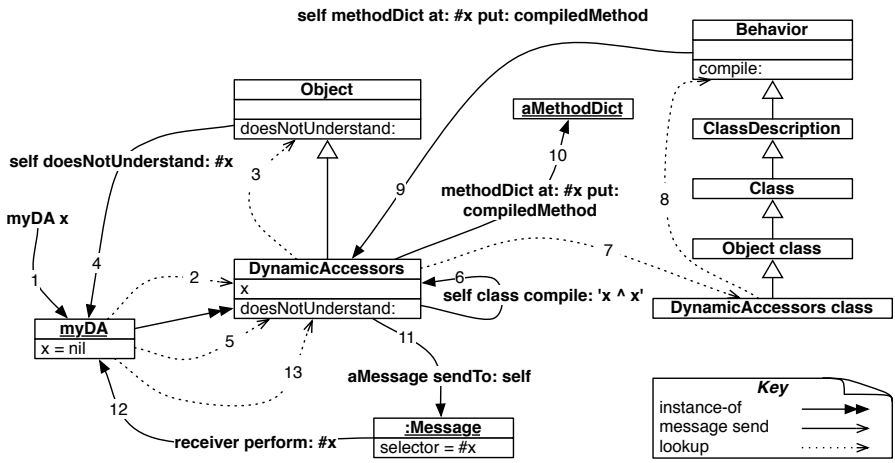


Figure 14.9: Creación dinámica de Accessors.

(1) Enviamos x a myDA, (2) el mensaje se busca desde la clase hacia arriba en la jerarquía, y (3) no se lo encuentra allí. (4) Esto genera un self doesNotUnderstand: #x que se envía de vuelta al objeto, (5) lo que dispara una nueva búsqueda. Esta vez, se encuentra a doesNotUnderstand: inmediatamente en DynamicAccessors, (6) el cual pide a su clase que compile la cadena 'x ↑ x'. El método compile se busca hacia arriba (7), y (8) finalmante se encuantra en Behavior, el cual (9–10) agrega el nuevo método compilado al diccionario de métodos de DynamicAccessors. Por último, (11–13) el mensaje se reenvía, y esta vez se lo encuentra.

La misma técnica se puede usar para generar setters para las variables de instancia, u otros tipos de código basado en plantillas, como los métodos de visita de un Visitor.

Se puede advertir el uso de Object»perform: en el paso (13), que se puede utilizar para enviar mensajes compuestos al momento de ejecución:

```
5 perform: #factorial                                  ⟶  120
6 perform: ('fac', 'torial') asSymbol                  ⟶  720
4 perform: #max: withArguments: (Array with: 6)        ⟶  6
```

14.7 Los objetos como métodos wrappers

Ya hemos visto que los métodos compilados son objetos ordinarios en Smalltalk, y que admiten ciertos métodos para permitir que el programador consulte el sistema de tiempo de ejecución. Lo que tal vez cause un poco de sorpresa es que *cualquier objeto* puede jugar el rol de un método compilado.

Todo lo que debe hacer es responder al mensaje run:with:in: y otros pocos más mensajes importantes.

 *Define una clase vacía* Demo. *Evalúa* Demo new answer42 *y nota cómo se dispara el error usual "Message Not Understood".*

Ahora vamos a instalar un objeto llano de Smalltalk en el diccionario de métodos de nuestra clase Demo.

 *Evalúa* Demo methodDict at: #answer42 put: ObjectsAsMethodsExample new. *Ahora intenta obtener de nuevo el resultado de* Demo new answer42. *Esta vez obtenemos la respuesta* 42.

Si echamos una mirada a la clase ObjectsAsMethodsExample vamos a encontrar los siguientes métodos:

```
answer42
    ↑42

run: oldSelector with: arguments in: aReceiver
    ↑self perform: oldSelector withArguments: arguments
```

Cuando nuestra instancia de Demo recibe el mensaje answer42, la búsqueda del método procede como es usual, sin embargo, la máquina virtual detecta que en lugar de un método compilado hay un objeto ordinario tratando de ocupar su lugar.

La VM enviará a este objeto un nuevo mensaje run:with:in: con el selector del método original, los argumentos y el receptor como sus argumentos. Como ObjectsAsMethodsExample implementa este método, intercepta el mensaje y lo delega a sí mismo.

Ahora podemos eliminar el método extraño con:

```
Demo methodDict removeKey: #answer42 ifAbsent: []
```

Si miramos con atención en ObjectsAsMethodsExample, veremos que su superclase también implementa los métodos flushcache, methodClass: y selector:, pero que están todos vacíos. Estos mensajes se pueden enviar a los métodos compilados, y por lo tanto deben ser implementados por cualquier objeto que pretenda hacerse pasar por un método compilado. (flushcache es el método más importante que se debe implementar; los otros pueden llegar a ser necesarios pero eso depende de si el método se instala mediante Behavior»addSelector:withMethod: o directamente mediante la utilización de MethodDictionary»at:put:.)

Usando métodos wrappers para realizar pruebas de cobertura

Los métodos wrappers son una técnica bastante conocida para interceptar mensajes[3]. En la implementación original[4], un método wrapper es una instancia de la subclase CompiledMethod. Al instalarlo, el método wrapper puede realizar acciones especiales antes o después de invocar al método original. Cuando se desinstala, el método original es devuelto a su correcta posición en el diccionario de métodos.

En Pharo, los métodos wrappers pueden ser implementados más facilmente al implementar run:with:in: como instancia de la la subclase CompiledMethod. De hecho, existe una implementación ligera de objetos como métodos wrappers[5], pero esto no forma parte de Pharo estandar al momento de escribir esto.

Sin embargo, el Pharo Test Runner usa precisamente esta técnica para evaluar la cobertura de los test. Hechemos un vistazo al funcionamiento de esto:

El punto de entrada para la prueba de cobertura es el método TestRunner»runCoverage:

```
TestRunner»runCoverage
    | packages methods |
    ... "identify methods to check for coverage"
    self collectCoverageFor: methods
```

El método TestRunner»collectCoverageFor: claramente ilustra el algoritmo que chequea la cobertura:

```
TestRunner»collectCoverageFor: methods
    | wrappers suite |
    wrappers := methods collect: [ :each | TestCoverage on: each ].
    suite := self
        reset;
        suiteAll.
    [ wrappers do: [ :each | each install ].
      [ self runSuite: suite ] ensure: [ wrappers do: [ :each | each uninstall ] ] ]
     valueUnpreemptively.
    wrappers := wrappers reject: [ :each | each hasRun ].
    wrappers isEmpty
        ifTrue:
            [ UIManager default inform: 'Congratulations. Your tests cover all code
     under analysis.' ]
        ifFalse: ...
```

[3] John Brant et al., Wrappers to the Rescue. In Proceedings European Conference on Object Oriented Programming (ECOOP'98). Volume 1445, Springer-Verlag 1998.

[4] http://www.squeaksource.com/MethodWrappers.html

[5] http://www.squeaksource.com/ObjectsAsMethodsWrap.html

Un wrapper es creado por cada método a ser chequeado y cada wrapper se instala. Al correr la prueba, todos los wrappers son desintalados. Finalmente el usuario obtiene la información acerca de cuales métodos no han estado cubiertos.

Cómo hace el wrapper su trabajo? El wrapper TestCoverage tiene tres variables de instacia hasRun, reference and method. Se inicializan así:

```
TestCoverage class»on: aMethodReference
    ↑ self new initializeOn: aMethodReference

TestCoverage»initializeOn: aMethodReference
    hasRun := false.
    reference := aMethodReference.
    method := reference compiledMethod
```

La instalación y desinstalación de los métodos simplemente actualiza el diccionario de métodos en la forma obvia:

```
TestCoverage»install
    reference actualClass methodDictionary
        at: reference methodSymbol
        put: self

TestCoverage»uninstall
    reference actualClass methodDictionary
        at: reference methodSymbol
        put: method
```

y el método run:with:in: simplemente actualiza la variable hasRun ,desinstala el wrapper (una vez que la cobertura haya sido verificada), y reenvía el mensaje al método original

```
run: aSelector with: anArray in: aReceiver
    self mark; uninstall.
    ↑ aReceiver withArgs: anArray executeMethod: method

mark
    hasRun := true
```

(Dele un vistazo a ProtoObject»withArgs:executeMethod: para ver cómo un método desplazado de su diccionario de métodos puede ser invocado.)

Esto es todo lo que hay que hacer!

Los métodos wrappers puede ser usados para realizar cualquier tipo de comportamiento adecuado antes o después de la normal operación de un método. Aplicaciones típicas son en instrumentación (recolectando estadísticas sobre invocaciones de los patrones de métodos), chequeo de pre- y post-

condiciones, y en memoization (opcionalmente haciendo cache de valores computados por los métodos).

14.8 Pragmas

Un *pragma* es una anotación que específica información específica sobre un programa, pero que no está envuelto en su ejecución. Los Pragmas no afectan directamente la operación del método que anotan. Los Pragmas tienen variados usos, como ser:

- Información del compilador: Los pragmas pueden ser usados para hacer que un método llame a una función primitiva. Esta función tiene que estar definida por la máquina virtual o ser un pluggin externo.
- Procesamiento en tiempo de ejecución: Algunos pragmas están disponibles para ser examinados en tiempo de ejecución.

Los pragmas pueden ser aplicados a la declaración de métodos de un programa solamente. Un método puede declarar uno o más pragmas y todos los pragmas deben estar declarados anteriormente a cualquier sentencia Smalltalk. Cada pragma es efectivamente un mensaje estático enviado con argumentos literales.

Brevemente mencionamos pragmas cuando introdujimos primitivas al comienzo de este capítulo. Una primitiva no es más qu una declaración de pragma. Considere <primitive: 73> que está contenida en instVarAt:. El selector de pragma es primitive: y su argumento es un inmediato valor literal 73.

El compilador es probablemente el mayor usuario de pragmas. SUnit es otra herramienta que hace uso de las anotaciones. SUnit es capaz de estimar la cobertura de una aplicación desde una unidad de prueba. Uno puede querer excluir algunos métodos de la cobertura. Este es el caso del método documentation en la clase SplitJointTest:

```
SplitJointTest class»documentation
    <ignoreForCoverage>
    "self showDocumentation"

    ↑ 'This package provides function.... '
```

Simplemente anotando un método con el pragma <ignoreForCoverage> uno puede controlar el alcance de la cobertura

Como instancias de la clase Pragma, los pragmas son objetos de primera clase. Un método compilado responde al mensaje pragmas. Este método devuelve un arreglo de pragmas.

```
(SplitJoinTest class >> #showDocumentation) pragmas
    ⟶    an Array(<ignoreForCoverage>)
(Float>>#+) pragmas    ⟶    an Array(<primitive: 41>)
```

Los métodos que definene una consulta particular pueden ser devueltos desde la clase. La class-side SplitJoinTest contiene algunos métodos anotados con <ignoreForCoverage>:

```
Pragma allNamed: #ignoreForCoverage in: SplitJoinTest class    ⟶    an Array(<
    ignoreForCoverage> <ignoreForCoverage> <ignoreForCoverage>)
```

Una variante de allNamed:in: puede ser encontrada en la class-side de Pragma.

Un pragma conoce en cual método esta definido (usando method), el nombre del método (selector) la clase que contiene el método (methodClass), su número de argumentos (numArgs), acerca de los literales que el pragma tiene por argumentos (hasLiteral: y hasLiteralSuchThat:).

14.9 Resumen del capítulo

Reflexión se refiere a la habilidad de consultar, examinar e incluso modificar metaobjetos tanto del sistema en ejecución como objetos ordinarios.

- El Inspector usa instVarAt: y métodos realacionados para consultar y modificar variables de instancia "privadas" de los objetos.
- Envie Behavior»allInstances para consultar instancias de una clase.
- Los mensajes class, isKindOf:, respondsTo: etc. son útiles para obtener métricas o construir herramientas de desarrollo, pero deben ser evitados en aplicaciones regulares: Violan el encapsulamiento de los objetos y hacen su código difícil de entender y mantener
- SystemNavigation es una clase útil para explotar muchas útiles consultas sobre la navegación e inspección de la jerarquía de lass. Por ejemplo, use \ t SystemNavigation default browseMethodsWithSourceString: 'pharo'. para encontrar y ver todos los métodos con una determinada cadena de caracteres. (Es lento, pero completo!)
- Cada clase de Smalltalk apunta a una instancia del MethodDictionary el cual mapea selectores a instancias de CompiledMethod. Un método compilado conoce su propia clase, cerrando el círculo.
- MethodReference es un proxy ligero para un método compilado, proveyendo conveniententemen métodos adicionales y es usado por muchas herramientas de Smalltalk.

- BrowserEnvironment, es parte de la infraestructura del Refactoring Browser, ofrece una interfaz más refinada que SystemNavigation para consultas el sistema, desde el resultado de una consulta, puede ser usado como el alcance de una nueva consulta. Tanto la GUI y las intefaces programables están disponibles.

- thisContext es una pseudo-variable que reifica la pila de ejecución de la máquina virtual. Es mayormente usada por el depurador para construir dinámicamente una vista interactiva de la pila. Es también útil para determinar dinámicamente el remitente de un mensaje.

- Los puntos de ruptura inteligentes pueden ser ajustados usando haltIf:, tomando un selector de método como su argumento. haltIf: se detiene solo si el método señalado es un remitente en la pila de ejecución.

- Una manera común de interceptar mensajes enviados a un determinado destino es usando un "objeto mínimo" como proxy del destino. El proxy implementa tan pocos métodos como le es posible y atrapa todos los mensajes enviados por la implementación de doesNotunderstand:. También puede ser usado para realizar algunas acciones adiciones y recién entonces enviar el mensajes al destino original.

- Enviar become: para intercambiar las referencias de dos objetos , tal como un proxy y su destino.

- Cuidado, algunos mensajes, como class y yourself nunca son realmenten enviados, pero son interceptados por la máquina virtual. Otros, como +, ifTrue: y − pueden ser directamente interceptados o introducidos por la máquina virtual dependiendo del receptor.

- Otro uso típico para la sobreescritura de doesNotUnderstand: es la carga lenta o la compilación de métodos perdidos.

- doesNotUnderstand: no puede atrapar self-sends.

- Una forma más rigurosa de interceptar mensajes es usar un objeto como un método wrapper. Este tipo de objeto es instalado en el diccionario de métodos en lugar de un método compilado. Debería implementar run:with:in:, el cual es enviado por la máquina virtual cuando detecta un objeto ordinario en lugar de un método compilado en el diccionario de métodos. Esta técnica es usada por SUnit Test Runner para recoger información sobre la cobertura.

Part IV

Appendices

Appendix A

Frequently Asked Questions

A.1 Getting started

FAQ 1 *Where do I get the latest Pharo?*

Answer http://www.pharo-project.org/

FAQ 2 *Which Pharo image should I use with this book?*

Answer You should be able to use any Pharo image, but we recommend you to use the prepared image on the Pharo by Example web site: http://PharoByExample.org. You should also be able to use most other images, but you may find that the hands-on exercises behave differently in surprising ways.

A.2 Collections

FAQ 3 *How do I sort an* OrderedCollection*?*

Answer Send it the message asSortedCollection.

```
#(7 2 6 1) asSortedCollection  ⟶  a SortedCollection(1 2 6 7)
```

FAQ 4 *How do I convert a collection of characters to a* String*?*

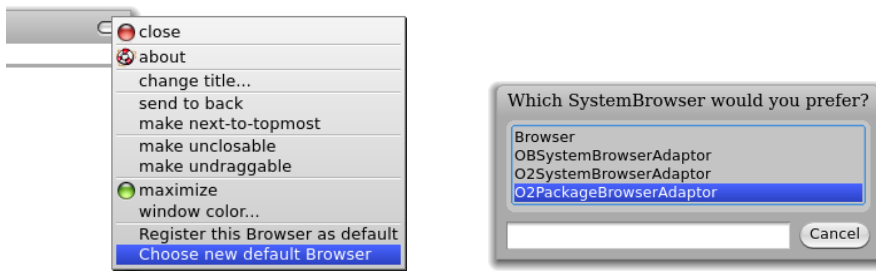
Answer

```
String streamContents: [:str | str nextPutAll: 'hello' asSet]  ⟶  'hleo'
```

A.3 Browsing the system

FAQ 5 *The browser does not look like the one described in the book. What gives?*

Answer You are probably using an image in which a different version of the OmniBrowser is installed as the default browser. In this book we assume that the OmniBrowser *Package Browser* is installed as the default. You can change the default by clicking on the menu bar of the browser. Just click on the gray lozenge in the top right corner of the window, select "Choose new default Browser", and then pick the O2PackageBrowserAdaptor. The next browser you open will be the Package Browser.



(a) Choose a new browser (b) Select the OB Package Browser

Figure A.1: Changing the default browser

FAQ 6 *How do I search for a class?*

Answer CMD−b (browse) on the class name, or CMD−f in the category pane of the browser.

FAQ 7 *How do I find/browse all sends to super?*

Answer The second solution is much faster:

```
SystemNavigation default browseMethodsWithSourceString: 'super'.
SystemNavigation default browseAllSelect: [:method | method sendsToSuper ].
```

FAQ 8 *How do I browse all super sends within a hierarchy?*

Answer

```
browseSuperSends := [:aClass | SystemNavigation default
   browseMessageList: (aClass withAllSubclasses gather: [ :each |
      (each methodDict associations
         select: [ :assoc | assoc value sendsToSuper ])
            collect: [ :assoc | MethodReference class: each selector: assoc key ] ])
   name: 'Supersends of ' , aClass name , ' and its subclasses'].
browseSuperSends value: OrderedCollection.
```

FAQ 9 *How do I find out which new methods are introduced by a class? (I.e., not including overridden methods.)*

Answer Here we ask which methods are introduced by True:

```
newMethods := [:aClass| aClass methodDict keys select:
   [:aMethod | (aClass superclass canUnderstand: aMethod) not ]].
newMethods value: True    ⟶    an IdentitySet(#asBit)
```

FAQ 10 *How do I tell which methods of a class are abstract?*

Answer

```
abstractMethods :=
   [:aClass | aClass methodDict keys select:
      [:aMethod | (aClass>>aMethod) isAbstract ]].
abstractMethods value: Collection    ⟶    an IdentitySet(#remove:ifAbsent: #add: #do:)
```

FAQ 11 *How do I generate a view of the AST of an expression?*

Answer Load AST from squeaksource.com. Then evaluate:

```
(RBParser parseExpression: '3+4') explore
```

(Alternatively *explore it*.)

FAQ 12 *How do I find all the Traits in the system?*

Answer

```
Smalltalk allTraits
```

FAQ 13 *How do I find which classes use traits?*

Answer

```
Smalltalk allClasses select: [:each | each hasTraitComposition and: [each
    traitComposition notEmpty ]]
```

A.4 Using Monticello and SqueakSource

FAQ 14 *How do I load a SqueakSource project?*

Answer

1. Find the project you want in http://squeaksource.com
2. Copy the registration code snippet
3. Select `open ▷ Monticello browser`
4. Select `+Repository ▷ HTTP`
5. Paste and accept the Registration code snippet; enter your password
6. Select the new repository and `Open` it
7. Select and load the latest version

FAQ 15 *How do I create a SqueakSource project?*

Answer

1. Go to http://squeaksource.com
2. Register yourself as a new member
3. Register a project (name = category)
4. Copy the Registration code snippet
5. `open ▷ Monticello browser`
6. `+Package` to add the category
7. Select the package
8. `+Repository ▷ HTTP`
9. Paste and accept the Registration code snippet; enter your password
10. `Save` to save the first version

FAQ 16 *How do I extend* Number *with* Number»chf *but have Monticello recognize it as being part of my* Money *project?*

Answer Put it in a method-category named *Money. Monticello gathers all methods that are in other categories named like *package and includes them in your package.

A.5 Tools

FAQ 17 *How do I programmatically open the SUnit TestRunner?*

Answer Evaluate TestRunner open.

FAQ 18 *Where can I find the Refactoring Browser?*

Answer Load AST then Refactoring Engine from squeaksource.com: http://www.squeaksource.com/AST http://www.squeaksource.com/RefactoringEngine

FAQ 19 *How do I register the browser that I want to be the default?*

Answer Click the menu icon in the top right of the Browser window.

A.6 Regular expressions and parsing

FAQ 20 *Where is the documentation for the RegEx package?*

Answer Look at the DOCUMENTATION protocol of RxParser class in the VB-Regex category.

FAQ 21 *Are there tools for writing parsers?*

Answer Use SmaCC — the Smalltalk Compiler Compiler. You should install at least SmaCC-lr.13. Load it from http://www.squeaksource.com/SmaccDevelopment.html. There is a nice tutorial online: http://www.refactory.com/Software/SmaCC/Tutorial.html

FAQ 22 *Which packages should I load from SqueakSource SmaccDevelopment to write parsers?*

Answer Load the latest version of SmaCCDev — the runtime is already there. (SmaCC-Development is for Squeak 3.8)

Bibliography

Sherman R. Alpert, Kyle Brown and Bobby Woolf: The Design Patterns Smalltalk Companion. Addison Wesley, 1998, ISBN 0–201–18462–1

Kent Beck: Smalltalk Best Practice Patterns. Prentice-Hall, 1997

Kent Beck: Test Driven Development: By Example. Addison-Wesley, 2003, ISBN 0–321–14653–0

John Brant et al.: Wrappers to the Rescue. In Proceedings European Conference on Object Oriented Programming (ECOOP'98). Volume 1445, Springer-Verlag 1998, 396–417

Erich Gamma et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison Wesley, 1995, ISBN 0–201–63361–2–(3)

Adele Goldberg and David Robson: Smalltalk 80: the Language and its Implementation. Reading, Mass.: Addison Wesley, May 1983, ISBN 0–201–13688–0

Dan Ingalls et al.: Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97). ACM Press, November 1997 ⟨URL: http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html⟩, 318–326

Wilf LaLonde and John Pugh: Inside Smalltalk: Volume 1. Prentice Hall, 1990, ISBN 0–13–468414–1

Alec Sharp: Smalltalk by Example. McGraw-Hill, 1997 ⟨URL: http://stephane.ducasse.free.fr/FreeBooks/ByExample/⟩

Bobby Woolf: Null Object. In **Robert Martin, Dirk Riehle and Frank Buschmann, editors:** Pattern Languages of Program Design 3. Addison Wesley, 1998, 5–18