El libro de Cuis-Smalltalk

Introducción a la programación con Smalltalk

Me gusta... es amigable y limpio y sencillo y bonito. ¡Muy amigable!
—*Alan Kay*

H. Fernandes with K. Dickey & J. Vuletich

This book is for Cuis-Smalltalk (6.0#5608 or later), a free and modern implementation of the Smalltalk language and environment.

Copyright © 2020, 2021, 2022, 2023, 2024, 2025 H. Fernandes with K. Dickey & J. Vuletich

Gracias a Matt Armstrong, Ezequiel Birman, Vanessa Freudenberg, Michael Khol, Szabolcs Komáromi, David Lewis, John McGarey, Mariano Montone, Ricardo Pacheco, Barry Perryman, Tommy Pettersson, Bernhard Pieber, Will Plusnick, Mauro Rizzi, Stephen Smith, Ignacio Sniechowski, Adam Teichert & Mark Volkmann por las revisiones del libro, sugerencias y borradores. Vuestra ayuda ha sido muy valiosa.

Traducción al español © 2025 Fernando Arroba (Notxor)

Compilation : 2 noviembre 2025

Documentation source: `https://github.com/DrCuis/TheCuisBook`

The contents of this book are protected under Creative Commons Attribution-ShareAlike 4.0 International.

You are free to:
Share – copy and redistribute the material in any medium or format
Adapt – remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:
Attribution. You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
Share Alike. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Complete license: `https://creativecommons.org/licenses/by-sa/4.0/legalcode`

Resumen del Contenido

Prólogo 1
1 Principios 4
2 El modo de vida del mensaje 14
3 Clase - Modelo de Comunicación entre Entidades 26
4 El estilo de vida de la colección 46
5 Mensajes de control de flujo 64
6 Visual con Morph 74
7 Los fundamentos de Morph 91
8 Eventos 112
9 Gestión del código 118
10 Depuración y manejo de excepciones 134
11 Compartir Cuis 142
A Copyright de documentos 148
B Resumen de sintaxis 149
C Los ejercicios 153
D Soluciones a los ejercicios 155
E Los Ejemplos 167
F Las Figuras 169
G Spacewar! Source Code 171
H Índice temático 179

Índice General

Prólogo 1

1 Principios 4
1.1 Contexto histórico 4
1.2 Instalar y configurar Cuis-Smalltalk 6
1.2.1 Editar tus preferencias 7
1.2.2 Diversión con la posición de las ventanas 8
1.3 Escribir tus primeros scripts 9
1.3.1 Diversión con números 10
1.4 Spacewar! 12

2 El modo de vida del mensaje 14
2.1 Comunicando entidades 14
2.2 Definiciones de envío de mensajes 15
2.3 Mensaje a entidades de cadena 16
2.4 Mensajes a entidades numéricas 18
2.5 Breve introducción al Browser del sistema 19
2.6 Modelos Spacewar! 22
2.6.1 Primeras clases 22
2.6.2 Paquete Spacewar! 23
2.6.3 El modelo newtoniano 24

3 Clase - Modelo de Comunicación entre Entidades 26
3.1 Comprendiendo la Programación Orientada a Objetos 26
3.2 Explorar POO desde el Browser 30
3.3 Sistema de clases de Cuis 33
3.4 Kernel-Numbers 34
3.5 Kernel-Text 38
3.6 Spacewar! Estados y comportamientos 40
3.6.1 Los estados del juego 40
3.6.2 Variables de instancia 42
3.6.3 Comportamientos 42
3.6.4 Inicializado 45

4 El estilo de vida de la colección 46
4.1 String – una colección particular 46
4.2 Diversión con variables 48
4.3 Diversión con colecciones 49
4.4 Detalles de las colecciones 56
4.5 Colecciones de SpaceWar! 60
4.5.1 Instanciar colecciones 60
4.5.2 Colecciones en acción 61

5 Mensajes de control de flujo 64
5.1 Elementos sintácticos 64
5.2 Pseudovariables 64

5.3 Sintaxis de método ... 65
5.4 Sintaxis de bloque ... 66
5.5 Control de flujo con bloques y mensajes ... 67
5.6 Métodos de Spacewar! ... 70
5.6.1 Inicializar el juego ... 70
5.6.2 Controles de la nave espacial ... 71
5.6.3 Colisiones ... 73

6 Visual con Morph ... 74
6.1 Instalar un paquete ... 74
6.2 Morph Ellipse ... 75
6.3 Submorph ... 76
6.4 Una breve introducción a los inspectores ... 77
6.5 Construyendo tu Morph especializado ... 82
6.6 Morphs de Spacewar! ... 83
6.6.1 Todos los Morphs ... 83
6.6.2 El arte de refactorizar ... 85

7 Los fundamentos de Morph ... 91
7.1 Pasando a Vector ... 91
7.1.1 Un primer ejemplo ... 91
7.1.2 Morph que puedes mover ... 92
7.1.3 Morph relleno ... 93
7.1.4 Morph animado ... 95
7.1.5 Morph dentro de morphs ... 96
7.2 Un Morph reloj ... 97
7.3 Volviendo a los Morphs de Spacewar! ... 102
7.3.1 Estrella central ... 103
7.3.2 Nave espacial ... 103
7.3.3 Torpedo ... 105
7.3.4 Revisar el dibujado ... 106
7.3.5 Dibujar simplificado ... 109
7.3.6 Revisar colisiones ... 110

8 Eventos ... 112
8.1 Eventos del sistema ... 112
8.2 Mecanismo general ... 113
8.3 Eventos de Spacewar! ... 114
8.3.1 Eventos de ratón ... 114
8.3.2 Eventos de teclado ... 116

9 Gestión del código ... 118
9.1 La imagen ... 118
9.2 El Change Log ... 118
9.3 El Change Set ... 120
9.4 El paquete ... 123
9.5 Flujo de trabajo diario ... 129
9.5.1 Automatiza tu imagen ... 130
9.6 Archivo de código fuente ... 133

10 Depuración y manejo de excepciones 134
10.1 Inspeccionar lo inesperado 134
10.2 El Debugger 135
10.3 ¡Alto! 139

11 Compartir Cuis 142
11.1 Reglas de oro del Gremio de Smalltalk 142
11.2 Refactorizar para mejorar la comprensión 142

Apéndice A Copyright de documentos 148

Apéndice B Resumen de sintaxis 149

Apéndice C Los ejercicios 153

Apéndice D Soluciones a los ejercicios 155
Preface 155
Filosofía Smallltalk 155
El estilo de vida del mensaje 155
Clase – modelo de comunicación de entidades 156
El estilo de vida de la colección 158
Mensajes de control de flujo 160
Visual con Morph 161
Los fundamentos de Morph 162
Eventos 165
Gestión del código 165

Apéndice E Los Ejemplos 167

Apéndice F Las Figuras 169

Apéndice G Spacewar! Source Code 171

Apéndice H Índice temático 179

Prólogo

> Un lenguaje que no afecta a la manera en que piensas sobre programación, no merece conocerse.
>
> —*Alan Perlis*

Cuis-Smalltalk —en breve, Cuis— es un entorno portable para hacer, construir y compartir. Como cualquier otra herramienta, Cuis no necesita una actitud en particular o seguir un proceso concreto. Sin embargo, Cuis está hecho sobre una visión específica de lo que es el software y lo que significa. Como consecuencia, Cuis es especialmente efectivo si esas ideas resuenan contigo y, al menos ocasionalmente, dejas que guíen tus pensamientos y acciones.

En este libro, te acompañaremos mientras exploras, descubres y aprendes Cuis y Smalltalk.

No suponemos ningún conocimiento sobre programación de ordenadores, pero si tienes algún tipo de experiencia con un lenguaje Orientado a Objetos o Funcional, reconocerás muchos conceptos. Si no tienes ningún tipo de experiencia en programación o has codificado en uno imperativo, más cercano al lenguaje máquina, como son C o Ensamblador, muchas ideas serán nuevas.

En cualquier caso, especialmente durante los primeros capítulos, lee este libro y sigue los ejemplos con una mente abierta. Un nuevo punto de vista de lo que significa el desarrollo de software puede ser una experiencia enriquecedora. Para nosotros, programar es un proceso reflexivo.

Entendemos el desarrollo de software como la actividad de aprender y documentar conocimiento para usarlo nosotros mismos y para otros, y que también puede ejecutarse en una computadora. El hecho de que un ordenador pueda ejecutar el software y producir soluciones útiles a cualquier problema es una consecuencia de un conocimiento sensato y relevante. ¡Solamente hacerlo funcionar no es la parte importante!

Estas ideas dan forma a Cuis y a la experiencia de usarlo de manera intensa, las desarrollaremos a lo largo de este libro. Entre sus más obvias consecuencias está una pasión por reducir la complejidad innecesaria, mientras se proporciona una completa y viva experiencia del desarrollo de software.

Este libro es una introducción y una invitación a explorar Cuis. Esperamos que te unas a nosotros en este viaje de usarlo como medio de expresar ideas y pensamientos, de hacer cosas frescas y de hacer Cuis incluso mejor.

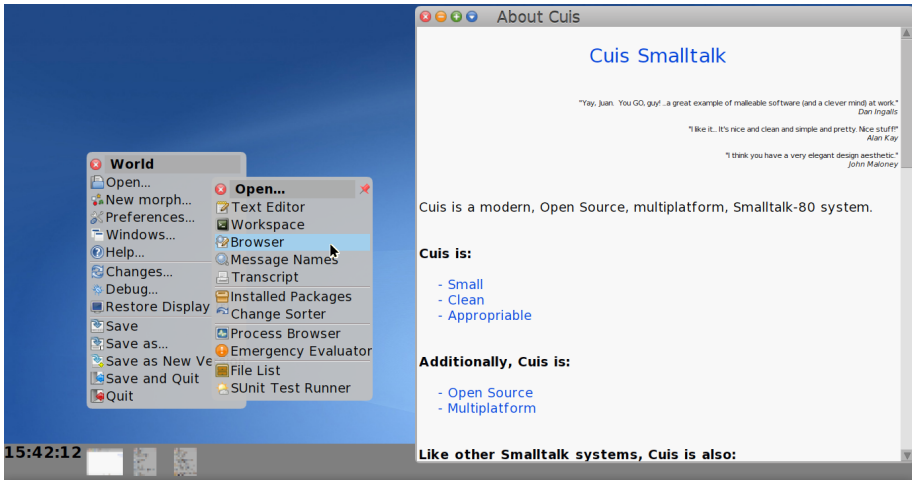

Figura 1: Cuis

Para hacer nuestro viaje con este libro más disfrutable, el proyecto de Spacewar![1] será un tema recurrente. Está destilado en el libro con ejemplos de código, ejercicios y capítulos dedicados. Al final del libro, habrás escrito una réplica de este histórico juego. El paquete final de Cuis-Smalltalk de este juego puedes descargarlo desde el repositorio del libro[2].

Cómo usar el libro

El lector puede estudiar Cuis-Smalltalk mediante dos versiones del libro:

- **Online.** Con un *web browser* en la dirección `https://DrCuis.github.io/TheCuisBook`. El libro está estructurado en capítulos y secciones, mostrándose de una en una. Es una manera bastante flexible de estudiar el libro: puedes abrir varios capítulos y apéndices en diferentes pestañas de tu navegador. Sin embargo, necesita una conexión a internet.
- **Offline.** En un fichero PDF único que puedes descargar y leer sin conexión. Se puede imprimir como un bonito libro en papel. La entrega más reciente se puede encontrar en `https://github.com/DrCuis/TheCuisBook`.

Los capítulos vienen con muchos ejemplos. Algunos se pueden copiar y pegar en Cuis-Smalltalk. Te animamos a hacer esto y en el proceso modificarlos para explorar por tu cuenta.

El código de los ejemplos en la versión online se puede copiar directamente y pegarlo en Cuis-Smalltalk. Esto es porque el carácter de asignación "←" como puedes ver en la ventana de desarrollo de Cuis-Smalltalk se imprime como ":=" en la versión online del libro. Lo mismo se puede aplicar al carácter de retorno "↑" impreso como "^" en la versión online.

En la versión PDF offline, el código de los ejemplos se impimen con los mismos caracteres de asignación y retorno que se pueden ver en las ventanas de Cuis-Smalltalk. Copiar y pegar también funciona como se espera.

---

[1] `https://es.wikipedia.org/wiki/Spacewar!`

[2] `https://github.com/DrCuis/TheCuisBook/blob/master/Spacewar!.pck.st`

Otros ejemplos son extractos de código que no son autosuficientes para ejecutarse tal cual. Su propósito es exponer facetas específicas del lenguaje Smalltalk.

Un ejemplo típico sin leyenda aparecerá como:

```
100 factorial
```

El mismo ejemplo con una leyenda aparecerá con un número y un título y se usará como referencia en cualquier punto del libro:

```
100 factorial
⇒  933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

Ejemplo 1: Soy un ejemplo con cabecera y resultado

También hay muchos ejercicios. La mayoría son muy sencillos; están para proporcionarte la oportunidad de aplicar lo que acabas de aprender.

Los ejercicios se identifican fácilmente por la mascona de Cuis-Smalltalk: *Cuis*, ¡un roedor de Sudamérica!

Busca en internet: ¿Cuántas versiones de Smalltalk existen?

Ejercicio 1: Soy un ejemplo de ejercicio

Las soluciones a los ejercicios se presentan en Apéndice D [Soluciones a los ejercicios], página 155.

Happy reading!

1 Principios

> Una computadora solo es un instrumento cuya música son ideas.
> —*Alan Kay*

Antes de meternos en detalles de cómo utilizar el lenguaje Cuis-Smalltalk y las herramientas para construir software necesitamos comprender el punto de vista, supuestos e intenciones que dan forma a cómo tiene sentido utilizar Cuis-Smalltalk. Podemos llamar a esto la filosofía de programación de Smalltalk,

1.1 Contexto histórico

Una idea muy extendida en el ámbito del software es que programar consiste simplemente en dar a un ordenador un conjunto de instrucciones para resolver algún problema. Desde este punto de vista, el único valor del software es lograr un resultado y, por lo tanto, un programa solo tiene interés en la medida en que resuelve ese problema. Además, como programar no es interesante en sí mismo, se deja en manos de un gremio profesional con conocimientos técnicos especializados sobre cómo escribir software, y el resto del mundo simplemente lo ignora.

Nos centramos en un aspecto concreto de la historia de las ideas fundamentales que nos proporcionan una perspectiva diferente desde la que explorar el desarrollo de software. La evolución histórica de estas ideas difiere de la visión de *simplemente resolver un problema*. Creemos que vale la pena volver a examinarla.

La primera visión clara de un procesador de información automatizado para aumentar nuestra memoria colectiva, para encontrar y compartir conocimientos —de hecho, para transformar la explosión de datos en una explosión de información y, a continuación, en una explosión de conocimientos y comprensión— se denominó Memex y se describió en el ensayo de Vannevar Bush «As We May Think» (Como podríamos pensar) en 1945. “As We May Think” in 1945 («Cómo podríamos pensar») en 1945.[1]. La descripción que hizo Bush de la posibilidad de desarrollar un sistema de procesamiento Memex para ayudar a las personas a acceder, desarrollar y capturar conocimientos en beneficio de todos inspiró a muchos pensadores e inventores posteriores en el desarrollo de ordenadores personales, redes, hipertexto, motores de búsqueda y repositorios de conocimientos como Wikipedia.[2].

A medida que la maquinaria computacional evolucionó desde grandes ordenadores centrales que ejecutaban un solo programa a la vez, pasando por ordenadores centrales de tiempo compartido, hasta llegar a los minicomputadores, se desarrolló otra área de conocimiento con modelos sobre cómo aprenden los seres humanos.[3]. La gente empezó a hablar de la *simbiosis humano-computadora*. Alan Kay pensó que el software y la programación informática podrían convertirse en un nuevo medio para expresar pensamientos y conocimientos. La capacidad de expresar ideas en este nuevo medio sería la nueva alfabetización.

---

[1] `https://es.wikipedia.org/wiki/Como_podr%C3%ADamos_pensar`

[2] Destacados hitos tempranos en esta linea de desarrollo son el *Sketchpad* de Ivan Sutherland (1963), la tablet RAND (1964) y el NLS (*oN-Line System*) de Doug Engelbart (1968). Desarrollos posteriores incluyen el *Xanadu* de Ted Nelson, el *Apple Macintosh*, la *World Wide Web*, los *smartphones* y las *tablets*.

[3] Destacados pioneros son J. Piaget, J. Brunner, O. K. Moore, y S. Papert.

Todo el mundo debería aprender a leer y escribir en este nuevo medio digital, y tendría a su disposición su propio libro dinámico, un *Dynabook*[4][5], para lograrlo.

Son muy interesantes los artículos sobre los desarrollos históricos de Alan Kay "The Early History of Smalltalk"[6] y "What is a Dynabook?"[7] en el que señala *«The best way to predict the future is to invent it»* (la mejor forma de predecir el futuro es inventarlo). Para hacer realidad la visión del Dynabook se necesitaban avances significativos y un desarrollo recíproco tanto del hardware como del software. El desarrollo original tuvo lugar en el «Xerox PARC research center» en los años 70. El primer hardware *interim Dynabook* fue el Xerox Alto, comúnmente considerado el primer ordenador personal. El sistema de software era Smalltalk. El diseño de ambos se guiaron por los objetivos de hacer reales el *Personal Dynamic Media* y el *Dynabook*. La última versión de Smalltalk hecha en Xerox fue Smalltalk-80.

Dadas las capacidades relativamente limitadas del hardware informático de aquella época, la implementación de esta visión planteaba retos reales y exigía mucha creatividad. Hoy en día, los teléfonos inteligentes, las tabletas y los ordenadores portátiles cuentan con las capacidades de hardware que requiere un Dynabook. Sin embargo, no se ha producido el mismo avance en el ámbito del software personal.

En 1981, Dan Ingalls, uno de los primeros inventores de Smalltalk, escribió en su artículo "Design Principles Behind Smalltalk" (Principios de diseño detrás de Smalltalk)[8] una serie de principios que aún hoy nos guían. Entre ellos:

> **Personal Mastery** (maestría personal). Para que un sistema sirva al espíritu creativo, debe ser totalmente comprensible para una sola persona.
>
> **Reactive Principle** (principio reactivo). Todos los componentes accesibles para el usuario deben poder presentarse de forma pertinente para su observación y manipulación.

Con la comercialización del software, la tendencia ha sido ofrecer a los usuarios aplicaciones «preempaquetadas», selladas y escritas por desarrolladores de software profesionales. Es posible personalizar una aplicación cambiando la «configuración del usuario», pero no hay forma de ver ni modificar sus capacidades más profundas.

En consonancia con el ideal de la alfabetización digital personal, creemos que todo el mundo debería tener pleno acceso al software que hace funcionar nuestros sistemas. Para comprender y explorar los sistemas informáticos es necesario escribir software de forma que pueda leerse y compartirse.

El cambio de mentalidad en la resolución de problemas puede pasar de *«¿Qué puedo hacer aquí con lo que se me presenta?»* a preguntarse *«¿Qué herramientas necesito para tener éxito aquí?»*, y luego desarrollarlas para tener cada vez más éxito.

Para nosotros es importante esta forma de pensar sobre los sistemas de software.

Por esta razón, Cuis es un sistema Smalltalk cuyo *kernel* (núcleo) se encuentra aún bastante cerca de Smalltalk-80. El objetivo de Cuis-Smalltalk es ser un entorno de desarrollo Smalltalk pequeño y coherente que, con estudio, resulte comprensible para una sola persona.

---

[4] "A personal computer for children of all ages"(1972) `http://www.vpri.org/pdf/hc_pers_comp_for_children.pdf`

[5] "Personal Dynamic Media"(1977) `http://www.vpri.org/pdf/m1977001_dynamedia.pdf`

[6] `http://worrydream.com/EarlyHistoryOfSmalltalk`

[7] `http://www.vpri.org/pdf/hc_what_Is_a_dynabook.pdf`

[8] `http://www.cs.virginia.edu/~evans/cs655/readings/smalltalk.html`

A medida que experimentamos y desarrollamos Cuis, este objetivo se lleva a cabo eliminando del sistema base todo lo que no sea esencial y adoptando una estrategia de composición que permita escribir o añadir cualquier otra característica según sea necesaria. A medida que se va adquiriendo conocimiento sobre el núcleo o corazón del software, solo hay que leer y aprender sobre cada característica adicional para comprender el conjunto.

Los entornos de software modernos y abiertos son muy complejos. Cuis es un intento de mantener la orientación y la capacidad de descubrir patrones sin perderse en una gran cantidad de posibilidades que no se comprenden del todo.

Se dice que uno entiende el mundo al construirlo. Desarrollar fluidez y profundidad en un nuevo medio requiere tiempo y práctica.

Te invitamos a que adquieras fluidez y esperamos que compartas con nosotros lo que vayas construyendo.

Sin embargo, primero debemos comenzar con algunos aspectos mecánicos.

Como dice el refrán: «Un viaje de mil millas comienza con un solo paso»[9].

Avancemos juntos.

1.2 Instalar y configurar Cuis-Smalltalk

Cuis-Smalltalk es un entorno y un lenguaje de programación que se ejecuta en un ordenador virtual idealizado. Se basa en dos componentes principales: la *máquina virtual* Smalltalk, que conceptualiza este ordenador virtual, y una *imagen* que representa el estado de este ordenador.

La máquina virtual es un programa ejecutable que se ejecuta en un host dedicado (GNU/Linux, Mac OS X, Windows, etc.). Se denomina *Open Smalltalk Virtual Machine*, o Squeak VM para abreviar. Existen diferentes tipos de VM, para diversas combinaciones de sistema operativo y arquitectura de CPU. Por lo tanto, una VM compilada para Windows en arquitectura Intel no funcionará en Linux en arquitectura ARM. Necesitas la VM específica compilada para la combinación de sistema operativo y arquitectura de CPU en la que se basa tu ordenador.

La *imagen* es un archivo normal que alimenta la VM con todos los objetos que definen el estado del ordenador virtual. Estos objetos son clases, métodos, instancias de esas clases como números, cadenas, ventanas, depuradores, es decir, todo lo que existía cuando se guardó el estado del ordenador virtual. Un *archivo imagen* guardado en un sistema operativo y una arquitectura de CPU determinados **se ejecutará de forma idéntica en otro sistema**, solo se necesita una VM compatible.

La VM permite reiniciar una imagen con ventanas en las mismas ubicaciones entre diferentes sistemas operativos y diferentes arquitecturas de CPU sin necesidad de recompilar. Esto es lo que significa para nosotros la *portabilidad*.

Lo que hace especial a Cuis-Smalltalk son las entidades vivas de la imagen: su población y disposición de clases, cómo las clases se heredan unas de otras. El número de clases suele ser menor de 700.

Para que empieces fácilmente, te recomendamos instalar Cuis-University[10]. 12 . Aquí encontrarás paquetes para GNU/Linux, Mac OS X y Windows, para la arquitectura Intel. Estos paquetes se diferencian de la distribución básica de Cuis-Smalltalk en que incluyen una VM personalizada, junto con una imagen de Cuis-Smalltalk compatible que contiene

[9] `https://en.wikipedia.org/wiki/A_journey_of_a_thousand_miles_begins_with_a_single_step`

[10] `https://sites.google.com/view/cuis-university/descargas`

paquetes adicionales preinstalados.[11] y algunos otros paquetes listos para instalar que te facilitarán la vida cuando sigas los ejemplos y ejercicios del libro, o cuando explores por tu cuenta. Para cuando leas este libro, es probable que Cuis-Smalltalk haya evolucionado a una versión más nueva, pero lo que aprendas aquí debería ser fácilmente transferible.

Para ejecutar Cuis-Smalltalk en tu ordenador, extrae el paquete y ejecuta el script de ejecución en Windows/Linux (`run.bat` o `run.sh`) o, en OS X, lo que lanzará la aplicación Squeak. Una vez que Cuis-Smalltalk esté en funcionamiento, lee la información que se muestra en las ventanas. Cuando hayas terminado, puedes cerrar estas ventanas y ajustar Cuis-Smalltalk según tus preferencias.

La distribución de Cuis University debería funcionar en la mayoría de las plataformas comunes, pero siempre hay más variantes de plataformas de las que podemos probar. **Si tienes algún problema**, aquí hay dos fuentes de información. Si *no* tienes ningún problema, puedes ignorarlas por ahora.

- Instrucciones de instalación actualizadas en el repositorio GitHub Cuis: `https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev#running-cuis`

- Pregúntanos en la lista de correo de Cuis: `https://lists.cuis.st/mailman/listinfo/cuis-dev`

1.2.1 Editar tus preferencias

Una vez que hayas leído la información de las ventanas predeterminadas, lo siguiente que debes hacer es ajustar las propiedades visuales según tus preferencias y necesidades. Para ello, accede al menú World...click en el fondo → `Preferences...` y, a continuación, selecciona el pin situado en la parte superior derecha del menú para que sea permanente. Aquí tienes las opciones más importantes: la elección del Font size y los temes, según prefieras colores claros u oscuros. Hay otras preferencias que puedes explorar por ti mismo. Una vez que hayas terminado, ve al menú World → `Save...` para que tus preferencias sean permanentes. En este libro, mantenemos el tema predeterminado de Cuis-Smalltalk, te sugerimos que hagas lo mismo para que tu entorno refleje las capturas de pantalla del libro.

[11] Measures and units (Aconcagua), Dates and calendars (Chaltén), Refactorizaciones automatizadas mejoradas y adicionales mediante LiveTyping (anotaciones automáticas de tipo), un widget de búsqueda, TDD Guru, DenotativeObject (objetos sin clase)

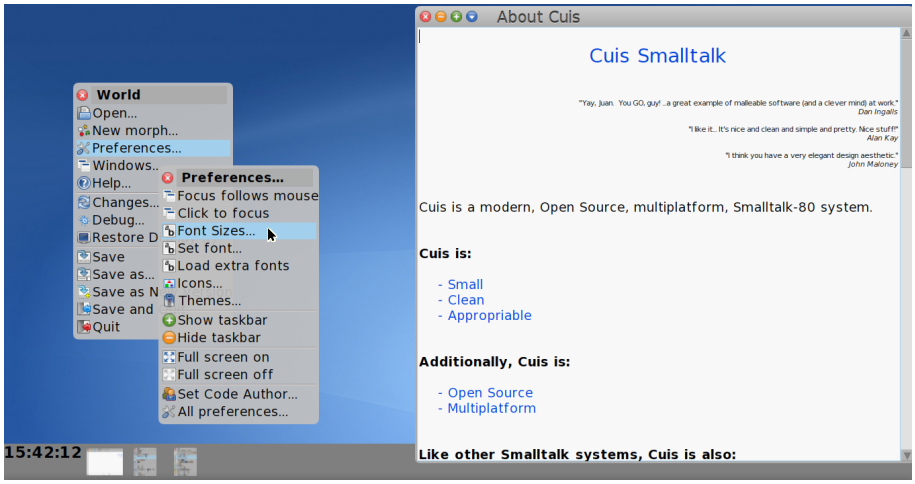

Figura 1.1: Establecer preferencias

1.2.2 Diversión con la posición de las ventanas

La primera herramienta que hay que descubrir es el *Workspace*. Que es una especie de editor de texto para introducir código Smalltalk que se puede ejecutar inmediatamente. Ve al ...menú World → `Open...` → `Workspace`...

Ahora le pediremos a Cuis-Smalltalk que cambie la ubicación de la ventana: haz clic en el icono azul (arriba a la izquierda) para acceder al menú de opciones de la ventana y experimenta con el área blanca para colocar la ventana del espacio de trabajo en la mitad izquierda del entorno Cuis-Smalltalk.

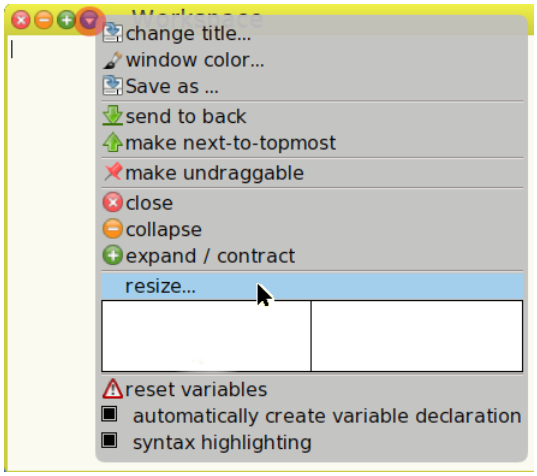

Figura 1.2: Opciones de la ventana

La opción `resize...` incluye incluso más libertad para posicionar la ventana. Prueba el siguiente ejercicio:

Utiliza la opción `resize...` *para situar el Workspace centrado en el entorno Cuis-Smalltalk.*

Ejercicio 1.1: Situar en el centro

1.3 Escribir tus primeros scripts

En esta sección aprenderemos a escribir sencillos scripts en el Workspace para probar y sentir el código Smalltalk. Los ejemplos se asocian con pequeños ejercicios para experimentar y los acompañan las soluciones en el apéndice. Intencionalmente dejamos los detalles de la sintaxis fuera de esta sección.

Se puede escribir el tradicional programa *Hello World!* en un Workspace de Smalltalk así:

```
Transcript show: 'Hello World!'
```

Ejemplo 1.1: El tradicional programa 'Hello World!'

Para ejecutar ese código, selecciónalo con el ratón y sobre él haz ...click derecho → `Do it (d)`... ¡Puede que no veas qué ocurre! De hecho para ver el resultado necesitas una ventana *Transcript* visible. El `Transcript` es un lugar donde un programador puede enviar información al usuario, como hemos visto aquí. Haz ...menú World → `Open...` → `Transcript`... y ejecuta el código de nuevo.

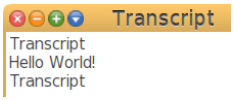

Figura 1.3: Ventana Transcript con salida

El código del Workspace tiene tres partes:

- la cadena literal `'Hello World!'`
- el mensaje `#show:` con su argumento `'Hello World!'`
- la clase `Transcript` que recibe el mensaje `#show:` con su argumento

La acción de imprimir toma lugar en la clase `Transcript`. También se puede invocar la ejecución de código con atajos de teclado `Ctrl-a` (*select* **A***ll*) luego `Ctrl-d` (**D***o it*).

```
Transcript show: 'Hello World!'.
Transcript newLine.
Transcript show: 'I am Cuising'
```

Ejemplo 1.2: Varias líneas

En este script de tres líneas, observa cómo las líneas están separadas por un punto «.». Este punto (o punto y coma) es un separador de líneas, por lo que no es necesario en la tercera línea ni en un script de una sola línea. El mensaje `#newLine` no tiene ningún argumento.

Una *String* es la forma en que se representa el texto en un lenguaje de programación, es una colección de caracteres. Ya conocimos las cadenas en nuestro primer script, están entre comillas simples: `'hello world!'`. Las ponemos en mayúsculas con el mensaje `#capitalized` message:

```
Transcript show: 'hello world!' capitalized
⇒ 'Hello world!'
```

Para converitr todos los caracteres en mayúsculas se utiliza el mensaje `#asUppercase`:

```
Transcript show: 'hello world!' asUppercase
⇒ 'HELLO WORLD!'
```

Dos cadenas se concatenan con el mensaje `#,`:

```
Transcript show: 'Hello ', 'my beloved ', 'friend'
⇒ 'Hello my beloved friend'
```

Ejemplo 1.3: Concatenar cadenas

Añade un mensaje para modificar el Ejemplo 1.3 para que produzca 'Hello MY BELOVED friends'.

Ejercicio 1.2: Concatenación y mayúsculas

1.3.1 Diversión con números

En tu Workspace, calcula un factorial ejecutando el ejemplo de abajo con *`Ctrl-a`* luego *`Ctrl-p`* (**P***rint it*):

```
100 factorial
⇒  933262154439441526816992388562667004907159682643816214685929
638952175999932299156089414639761565182862536979208272237582
511852109168640000000000000000000000000
```

Cuis-Smalltalk maneja números enteros muy grandes sin requerir ningún tipo de declaración especial o método. Para convencerte a tí mismo prueba el ejemplo de aquí abajo:

```
10000 factorial / 9999 factorial
⇒ 10000
```

Si ejecutas e imprimes con *`Ctrl-p`* el código: `10000 factorial`, verás que se tarda mucho más tiempo en imprimir un resultado factorial que en calcular dos factoriales y dividirlos. El resultado es un número entero, como era de esperar, y no un número decimal escalado, como devolverían muchos lenguajes de programación.

Como estamos hablando de división, es posible que no obtengas el resultado que esperas:

```
15 / 4
⇒ 15/4
```

Parece que Cuis-Smalltalk es perezoso porque no responde el número decimal `3.75` como esperábamos. De hecho, Cuis-Smalltalk quiere ser lo más preciso posible, y ¡su respuesta es un número racional! Después de todo, una fracción es solo una división que somos demasiado perezosos para calcular, –porque es complicado–, ¡pero Cuis-Smalltalk hace esto!

Prueba esto para comprender lo que, en el fondo, está sucediendo:

```
(15 / 4) + (1 / 4)
⇒ 4
```

¿No es maravilloso? Cuis-Smalltalk realiza cálculos con números racionales. Comenzamos con operaciones de división y suma con números enteros, y obtuvimos un resultado preciso gracias al cálculo intermedio con números racionales.

En el ejemplo, observa cómo se utilizan los paréntesis, aunque en el cálculo aritmético la división se realiza primero. Con Cuis-Smalltalk es necesario especificar el orden de las operaciones con paréntesis. Más adelante explicaremos por qué.


Escribe el código para calcular la suma de los inversos de los cuatro primeros números enteros. El inverso de 4 es 1/4.

Ejercicio 1.3: Suma de inversos

Los enteros se pueden mostrar de diversas maneras con el mensaje apropiado:

```
2020 printStringRoman ⇒ 'MMXX'
2020 printStringWords  ⇒ 'two thousand, twenty'
"Number as the Maya did"
2020 printStringBase: 20 ⇒ '510'
```


Imprime 2020 como palabras, en mayúscula, capitalized.

Ejercicio 1.4: Número en palabras en mayúsculas

Los números enteros y decimales con los que hemos trabajado son *literales numéricos*. Los literales son los componentes básicos del lenguaje y se consideran expresiones constantes. Son literalmente lo que parecen.

Hay varias variantes sintácticas para denotar números:

Literal numérico	**Lo que representa**
`1`	entero (notación decimal)
`2r101`	entero (base binaria)
`16rFF`	entero (base hexadecimal)
`1.5`	número de coma flotante
`2.4e7`	coma flotante (notación exponencial)

Dependiendo del valor que necesitemos utilizar, podemos mezclar estas representaciones literales:

```
16rA + 1 + 5e-1 + 6e-2
⇒ 289/25
```

1.4 Spacewar!

El juego Spacewar! se desarrolló inicialmente en 1962 por Steve Russell en un miniordenador DEC PDP-1. Se dice que fue el primer videojuego conocido instalado en varios ordenadores y fue muy popular en la comunidad de programación en los años 60. Se ha portado y reescrito varias veces a diferentes arquitecturas de hardware y complementado con características adicionales. *Computer Space*, la primera máquina recreativa de videojuegos, era una réplica de Spacewar!

Figura 1.4: El juego Spacewar! en un microordenador DEC PDP-1

Wikipedia describe con gran precisión este juego de simulación de combate espacial:

> El juego consiste en dos naves espaciales monocromas, «la aguja» (the needle) y «la cuña» (the wedge), cada una controlada por un jugador, que intentan dispararse mutuamente mientras maniobran en un plano bidimensional en torno al pozo gravitatorio de una estrella, con un campo estelar como fondo. Las naves pueden disparar torpedos, que no se ven afectados por la atracción gravitatoria de la estrella; disponen un número limitado de torpedos y de combustible, que se consume cuando el jugador activa los propulsores de la nave. Mientras se intenta controlar la rotación de la nave para apuntar, los torpedos se disparan de uno en uno pulsando un interruptor en el ordenador

> o un botón en el mando de control y entre cada lanzamiento hay un periodo de enfriamiento. Las naves siguen en movimiento incluso cuando el jugador no acelera y la rotación de las naves no supone un cambio en la dirección de su movimiento.
>
> Cada jugador controla una de las naves y debe intentar derribar la otra mientras evita colisionar con la estrella o la nave contraria. Volar cerca de la estrella puede proporcionar asistencia gravitatoria al jugador y aumentar la velocidad de la nave, aunque arriesgándose a calcular mal la trayectoria y acabar cayendo en la estrella. Si una nave alcanza el borde de la pantalla reaparece en el lado opuesto, manteniendo la velocidad y la trayectoria.
>
> —*Wikipedia, Spacewar!*

Por lo tanto, los protagonistas del juego son:

1. una **estrella central** generando un campo de gravedad,
2. de fondo un **campo de estrellas**,
3. dos **naves espaciales** llamadas «la aguja» ()*the needle*) y «la cuña» (*the wedge*) controladas por dos jugadores.
4. **torpedos** disparados por las naves.

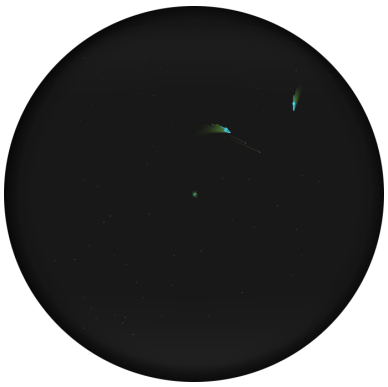

Figura 1.5: El juego Spacewar!

2 El modo de vida del mensaje

> La clave para crear sistemas excelentes y escalables reside mucho más en diseñar cómo se comunican sus módulos que en determinar cuáles deben ser sus propiedades y comportamientos internos.
>
> —*Alan Kay*

Un sistema Smalltalk es un conjunto de entidades que se comunican entre sí a través de mensajes. Eso es todo, no hay nada más.

2.1 Comunicando entidades

Cuando una entidad determinada recibe un mensaje de otra entidad, se activa un comportamiento específico. La entidad que recibe el mensaje se denomina *receptor* y la entidad que lo envía, *emisor*. En la terminología de Cuis-Smalltalk, una entidad se denomina *instancia de una clase*, *instancia de clase* o, simplemente, *instancia*. Una *clase* es una especie de modelo para una entidad.

El comportamiento se define internamente en el receptor y se puede activar desde cualquier instancia. **Los comportamientos sólo pueden ser invocados por mensajes enviados entre entidades**. Una entidad puede enviarse un mensaje a sí misma. Un comportamiento se define en una clase y se denomina *método*.

Esto da lugar a una enorme nube de entidades que se comunican entre sí mediante el envío de mensajes. Las nuevas entidades se instancian cuando es necesario y luego se reciclan automáticamente cuando ya no son necesarias. En un entorno Cuis-Smalltalk nuevo, el recuento de instancias de clase es superior a 150.000.

```
ProtoObject allSubclasses sum: [ :class | class allInstances size]
⇒ 152058
```

Ejemplo 2.1: Calculando el número de entidades

El número de clases, los modelos para las entidades –instancias de la clase `Class`– es menor de 700.

```
Smalltalk allClasses size
⇒ 671
```

Ejemplo 2.2: Calcular el número de clases

Puesto que no estás usando una imagen básica, sino una destinada a aprender, las clases pueden constituir un número más grande.

Para ser sinceros, en el capítulo anterior omitimos este importante detalle sobre el diseño de Smalltalk. Escribimos sobre el envío de mensajes sin dar muchas explicaciones, porque queríamos que descubrieras este diseño de manera informal. Los scripts que leíste y escribiste trataban sobre entidades que se comunicaban entre sí a través de mensajes.

2.2 Definiciones de envío de mensajes

Hay tres tipos de mensajes en Cuis-Smalltalk:

- **Mensajes unitarios** no toman argumentos.
  En `1 factorial` el mensaje `#factorial` se envía al objeto `1`.

- **Mensajes binarios** toman exactamente un argumento
  En `1 + 2` el mensaje `#+` se envía al objeto `1` con el argumento `2`.

- **Mensajes de palabra clave** toman un número cualquiera de argumentos.
  En `2 raisedTo: 6 modulo: 10` el mensaje consistente en el selector de mensaje `#raisedTo:modulo:` y los argumentos `6` y `10` se envía al objeto `2`.

Los selectores de mensaje unitarios se forman con caracteres alfanuméricos y comienzan con una letra minúscula.

Los selectores de mensaje binarios consisten en uno o más caracteres de la siguiente lista:

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

Los selectores de mensajes de palabra clave consisten en series de palabras clave alfanuméricas, donde cada clave comienza con una letra minúscula y termina con dos puntos.

Los mensajes unitarios tienen la mayor precedencia, seguidos de los mensajes binarios y por último los mensajes de palabra clave, así:

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

Primero se envía `factorial` a `3`, después de envía `+ 6` a `1`, y finalmente se envía `raisedTo: 7` a `2`.

Dejando a un lado la precedencia, la evaluación se realiza estrictamente de izquierda a derecha, por lo que

```
1 + 2 * 3
⇒ 9
```

no es 7. Se deben utilizar paréntesis para alterar el orden de evaluación:

```
1 + (2 * 3)
⇒ 7
```

Los paréntesis también se pueden utilizar para aclarar código que podría resultar confuso. Por ejemplo, el estricto orden de evaluación de izquierda a derecha puede resultar confuso cuando se aplica a expresiones matemáticas. En el fragmento de código de Spacewar! que se muestra a continuación, los paréntesis aclaran que la suma se realiza primero:

```
newVelocity ← (ai + ag) * t + velocity
```

Ejemplo 2.3: Velocidad de la nave

Para enviar varios mensajes al mismo destinatario, se puede utilizar una *cascada* para indicar el destinatario una vez, seguido de la cascada de mensajes separados por punto y coma. Aquí está nuestro código anterior del Ejemplo 1.2 expresado con una cascada:

```
Transcript
   show: 'Hello World!';
   newLine;
   show: 'I am Cuising'
```

Ejemplo 2.4: Cascada de mensajes

Otro ejemplo de una cascada desde el juego Spacewar!:

```
aShip
   velocity: 0 @ 0;
   morphPosition: randomCoordinate value @ randomCoordinate value
```

Ejemplo 2.5: Detener y teletransportar una nave espacial a una posición aleatoria

Fíjate que aquí el texto se formatea para facilitar la comprensión del código. Es posible escribir una cascada de mensajes en una sola línea, pero reduce la legibilidad del código:

```
Transcript show: 'Hello World!'; newLine; show: 'I am Cuising'
```

La clase `Transcript` con frecuencia ayuda a presentar información útil cuando desarrollamos una aplicación. Una alternativa al *`Ctrl-d`* (**D***o it*) es el atajo *`Ctrl-p`* (**P***rint it*), que ejecuta el script y muestra el resultado directamente en el Workspace.

En el Ejemplo 2.4, no hemos solicitado ningún resultado especial. Al seleccionar el texto y pulsar *`Ctrl-p`*, se obtiene el resultado predeterminado, que es devolver el objeto al que se envía un mensaje, en este caso, el `Transcript`.

2.3 Mensaje a entidades de cadena

Como se ha indicado anteriormente, una `String` es una secuencia de elementos `Character` y representa texto plano (sin ningún formato).

En Cuis, como en la mayoría de los lenguajes de programación, los literales `String` son una sintaxis conveniente para crear instancias `String`. Estos son algunos ejemplos de literales `String`:

```
'Hello World!'.
'Accented letters: cigüeña, camión, déjà, deçà, forêt, coïncidence'.
'Non-latin (Greek) letters: Λορεμ ιπσθμ δολορ σιτ αμετ'.
```

El primer ejemplo es una instancia de `String`. Esta clase se utiliza para literales de cadena si todos los caracteres están en el conjunto de caracteres ASCII limitado. Los siguientes ejemplos contienen caracteres no ASCII. Para estos, se utiliza en su lugar una instancia de `UnicodeString`. Normalmente no es necesario preocuparse por esto: las instancias de `String` y `UnicodeString` entienden los mismos mensajes, por lo que son intercambiables.

El acceso a un carácter en una cadena se realiza con la palabra clave `#at:` y su posición en el índice de la cadena como argumento. Ejecuta los siguientes ejemplos con el atajo *`Ctrl-p`*:

```
'Hello' at: 1 ⇒ $H
'Hello' at: 5 ⇒ $o
```

Fíjate cómo los caracteres están precedidos por el símbolo «$».

Precaución. El índice indica la posición, comenzando naturalmente por 1, y es válido hasta la longitud de la cadena.

```
'Hello' indexOf: $e
⇒ 2
```

Para cambiar un carácter, utiliza el mensaje de dos palabras clave complementario `#at:put:`. El argumento debe indicarse como un carácter:

```
'Hello' at: 2 put: $a; yourself
⇒ 'Hallo'
```

Fíjate que utiliza una cascada con el mensaje `#yourself`. Una cascada envía los siguientes mensajes al receptor original, por lo que `#yourself` devuelve la cadena actualizada. Sin la cascada, se devuelve `$a` como resultado del mensaje `#at:put:`.

Reemplaza cada carácter de la cadena 'Hello' para que se convierta en 'Belle'.

Ejercicio 2.1: de Hello a Belle

Se puede solicitar el código ASCII de los caracteres que forman parte del conjunto de caracteres ASCII. A la inversa, dado un código ASCII, podemos solicitar el `Character` correspondiente:

```
$A asciiValue
⇒ 65
Character codePoint: 65 + 25
⇒ $Z
```

Pero los caracteres que no forman parte del ASCII no tienen un ASCII. En general, es mejor utilizar codePoint Unicode en su lugar:

```
$A codePoint
⇒ 65
Character codePoint: 65 + 25
⇒ $Z
$ϕ codePoint
⇒ 966
Character codePoint: 966
⇒ $ϕ
```

Barajar (shuffling) una cadena puede ser divertido, pero no muy útil. Sin embargo, el barajado se puede aplicar a cualquier tipo de colección, no solo a una cadena, y resultará útil, como veremos más adelante:

```
'hello world' shuffled
⇒ 'wod llreohl'
```

Fíjate que cada vez que se baraja se obtiene un resultado diferente.

Los mensajes se componen de manera natural:

```
'hello world' shuffled asArray
⇒ #($h $d $l $ "20" $o $w $e $l $l $o $r)
```

Un literal `Array` comienza con un carácter almohadilla o *sharp*, `$#`, y los elementos del array están entre paréntesis. En este caso, los elementos son `Character`s, pero pueden ser instancias de cualquier clase.

Como `#shuffled`, todas las colecciones responden al mensaje `#sorted`, cuya respuesta es una colección ordenada.

```
'hello world' sorted
⇒ ' dehlllloorw'
```

2.4 Mensajes a entidades numéricas

Anteriormente, hemos hablado de cómo Cuis-Smalltalk reconoce los números racionales. Las cuatro operaciones aritméticas y las funciones matemáticas se implementan con mensajes unarios y binarios que comprenden los números racionales:

```
(15 / 14) * (21 / 5) ⇒ 9 / 2
(15 / 14) / ( 5 / 21) ⇒ 9 /2
(3 / 4) squared ⇒ 9 / 16
(25 / 4) sqrt ⇒ 5 / 2
```

Escribe el código para calcular la suma de los cuadrados de los inversos de los cuatro primeros números enteros.

Ejercicio 2.2: Suma de los cuadrados

Si la división entera de Cuis-Smalltalk devuelve un número racional, ¿cómo podemos obtener el resultado en decimal? Una opción es escribir el número como un literal de punto flotante, un *Float*. Este tipo de literal se escribe con la parte entera y la parte fraccionaria separadas por un punto «.»:

```
 15.0 / 4 ⇒ 3.75
15 / 4.0 ⇒ 3.75
```

Otra opción es convertir un entero en un *float* con el mensaje `#asFloat`. Es muy útil cuando el entero está en una variable:

```
15 asFloat / 4
⇒ 3.75
```

También puedes pedir una división truncada para obtener un entero utilizando el mensaje `#//`:

```
15 // 4
⇒ 3
```

El resto de la división euclideana se calcula con el mensaje `#\\`:

```
15 \\ 4
⇒ 3
```

Cuis-Smalltalk tiene operaciones aritméticas para comprobar si un número entero es impar, par, primo o divisible por otro. Solo tienes que enviar el mensaje unario o la palabra clave adecuada al número:

```
25 odd ⇒ true
25 even ⇒ false
25 isPrime ⇒ false
23 isPrime ⇒ true
91 isDivisibleBy: 7 ⇒ true
117 isDivisibleBy: 7 ⇒ false
117 isDivisibleBy: 9 ⇒ true
```

Ejemplo 2.6: Pruebas con enteros

Con *mensajes de palabras clave* específicas, puede calcular el mínimo común múltiplo y el máximo común divisor. Un mensaje de palabra clave se compone de uno o más signos de dos puntos «:» para insertar uno o más argumentos:

```
12 lcm: 15 ⇒ 60
12 gcd: 15 ⇒ 3
```

En el juego Spacewar!, la estrella central es la fuente de un campo gravitatorio. Según la ley de la gravitación universal de Newton, cualquier objeto con masa –una nave espacial o un torpedo en el juego– está sujeto a la fuerza gravitatoria. La calculamos como un vector para tener en cuenta tanto su dirección como su magnitud. El fragmento de código del juego que se muestra a continuación muestra un cálculo mixto (simplificado) de escalares y vectores realizado con el envío de mensajes (véase la Figura 2.4):

```
-10 * self mass * owner starMass / (position r raisedTo: 3) * position
```

Ejemplo 2.7: Calcular el vector de la fuerza de la gravedad

2.5 Breve introducción al Browser del sistema

Smalltalk organiza los comportamientos de las instancias utilizando clases. Una clase es un objeto que contiene un conjunto de métodos que se ejecutan cuando una de sus instancias recibe un mensaje que es el nombre de uno de estos métodos.

El *Browser del sistema*, o simplemente el *Browser*, es una herramienta que controla todas las clases de Cuis-Smalltalk. Es una herramienta para explorar las clases (tanto del sistema como de usuario) y para escribir nuevas clases y métodos.

Para acceder a la herramienta ...menú World → `Open...` → `Browser`...

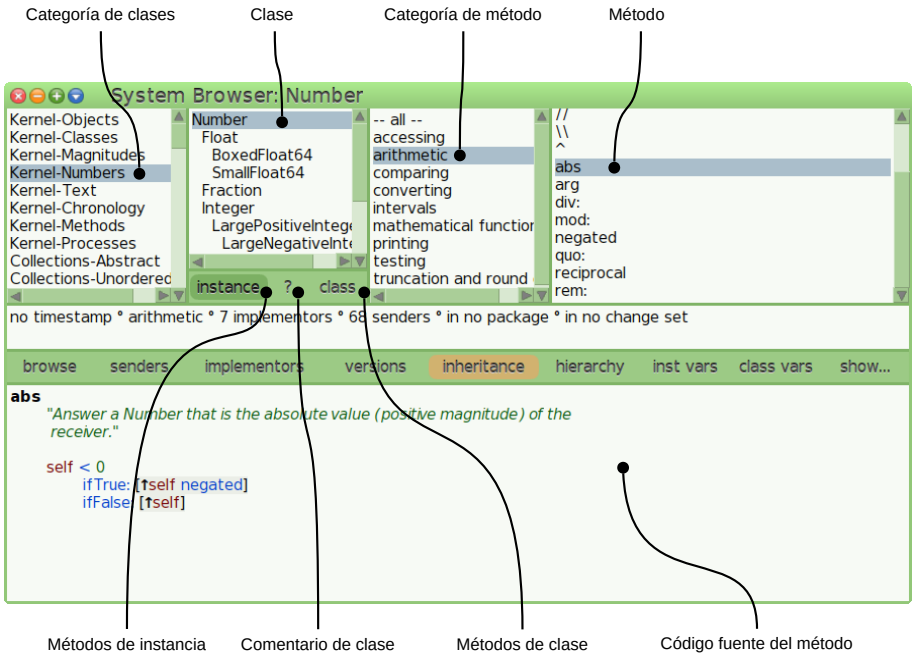

Figura 2.1: El Browser del sistema

En la parte superior izquierda se encuentran las *categorías de clases*, grupos de clases que comparten el mismo tema. Una categoría también se puede utilizar para crear un paquete (*Package*), que es un elemento organizativo para guardar código en un sistema de archivos. En la Figura 2.1, la categoría de clase seleccionada es **Kernel-Numbers**, un grupo de clases que ya hemos empezado a utilizar. El término **Kernel-** indica que forma parte de las categorías fundamentales, pero solo es una convención. Véanse otras categorías como **Kernel-Text** y **Kernel-Chronology**, relacionadas con entidades de texto y fecha.

A la derecha se encuentran las clases de la categoría seleccionada. Están bien presentadas en una jerarquía de clases padre-hijo. Cuando se selecciona una clase por primera vez en este panel, su declaración aparece en el panel grande de abajo. La declaración de la clase `Number` es:

```
Magnitude subclass: #Number
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Numbers'
```

Algunos puntos importantes de esta declaración:

- `Number` es una subclase de `Magnitude`. Es decir, `Number` es un tipo especializado de `Magnitude`.
- la propia declaración es código Smalltalk, efectivamente el mensaje `#subclass:instanceVariableNames:classVariableNames:...` se enviará a `Magnitude` para crear esta clase.
- el argumento de `subclass:`, `Number` tiene el prefijo "#". Es un símbolo, un tipo de

cadena única. Efectivamente cuando declaramos la clase `Number`, el sistema no tiene constancia de ella, por lo que la nombramos como un símbolo.

- El argumento de `instanceVariableNames:` es una cadena: las variables de instancia de la clase se declaran mediante nombres separados por un espacio. No hay ninguno en esta definición de clase.

Una subclase hereda comportamientos de su superclase padre, por lo que solo necesita describir lo que difiere de su superclase. Una instancia de `Number` añade métodos (que definen comportamientos) desconocidos para una instancia de `Magnitude`. Exploraremos esto en detalle a medida que avancemos.

Para conocer el propósito de una clase, es recomendable visitar **siempre** el comentario de la clase. A menudo, los comentarios también incluyen ejemplos de código para aprender a utilizar el objeto; estos fragmentos de código se pueden seleccionar y ejecutar in situ, tal y como se hace desde un espacio de trabajo. En la Figura 2.1, mira el botón `comment` para leer o editar el «comentario» de la clase seleccionada.

A la derecha del panel de clases se encuentra el panel de categorías de métodos. Una clase puede tener muchos métodos, por lo que agruparlos por categorías ayuda a otros usuarios a orientarse en la búsqueda de métodos relacionados. A modo de referencia, `Number` tiene más de 100 métodos de instancia implementados en sí mismo[1]. Al hacer clic en la categoría `arithmetic`, se accede directamente a los métodos relacionados en el siguiente y último panel de la derecha.

Un click derecho en el panel Categoría de Clases proporciona un menú contextual. Puedes seleccionar `find class ..` o, como indica el menú, utilizar *`Ctrl-f`* (**F***ind*), para obtener un panel de rellenar y escribir parte de un nombre de clase que buscar. Pruébalo con `String`.


¿Cuántos métodos hay en la categoría de métodos **arithmetic** *de la clase* `CharacterSequence`*?*

Ejercicio 2.3: Recuento de métodos

En el Browser, una vez seleccionado un método (como en Figura 2.1, método `abs`), la parte inferior muestra su código fuente, listo para ser explorado o editado. A menudo, encontrará un pequeño comentario justo después del nombre del método, entre comillas dobles.

Todo objeto conoce su propia clase y puede responder con ella al mensaje `#class`.

Truco. En el Workspace, hacer *`Ctrl-b`* (**B***rowse*) sobre el nombre de una clase abrirá un Browser con la clase nombrada:

- En el Workspace, teclea `2 class` e imprime con *`Ctrl-p`*,
- Se imprime `SmallInteger` y está destacado automáticamente como la selección actual,
- Invoca el Browser en la clase seleccionada `SmallInteger` con *`Ctrl-b`*,

[1] Si tenemos en cuenta a sus padres, el recuento combinado del método supera los 300.

- Se abre una nueva instancia de Browser en la clase `SmallInteger`, lista para la exploración.

2.6 Modelos Spacewar!

2.6.1 Primeras clases

En el último capítulo, enumeramos los protagonistas del juego. Ahora, proponemos una primera implementación del modelo del juego con un conjunto de clases que representan las entidades involucradas:

1. el juego ⇒ clase `SpaceWar`,
2. una estrella central ⇒ clase `CentralStar`,
3. dos naves espaciales ⇒ clase `SpaceShip`,
4. torpedos ⇒ clase `Torpedo`.

Antes de definir estas clases en Cuis-Smalltalk, queremos crear una categoría de clases dedicada para agruparlas ahí dentro.

En cualquier ventana de Cuis-Smalltalk, al hacer clic con el botón derecho del ratón en un panel, normalmente aparecerá un menú con las operaciones específicas que se pueden aplicar a ese panel.

Con el puntero del ratón sobre el panel de categorías de clases del navegador (el más a la izquierda), haz lo siguiente:

...click derecho → `add item`... luego teclea *Spacewar!*

Una vez creada nuestra nueva categoría, el navegador propone una plantilla de código en el panel del código fuente del método –el inferior– para crear una nueva clase en la categoría `Spacewar!`.

```
Object subclass: #NameOfSubclass
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Sustituimos el símbolo `#NameOfSubclass` por un símbolo que represente el nombre de la clase que queremos crear. Empecemos con `#SpaceWar`. Para guardar nuestra clase, mientras estamos sobre el código de declaración de la clase, hacemos do ...click derecho → `Accept`... Cuis-Smalltalk nos preguntará por nuestras iniciales y nombre si no lo ha hecho antes. También podemos simplemente hacer ***`Ctrl-s`*** (**S***ave*).

Ahora sólo tienes que repetir el proceso para cada uno de `#SpaceShip`, `#CentralStar` y `#Torpedo`. Si fuera necesario, para obtener otra plantilla de código de clases, clicka en la categoría de clase `Spacewar!`.

Cuando termines, tu categoría de clases debería estar rellena con cuatro clases, como se muestra en la Figura 2.2.

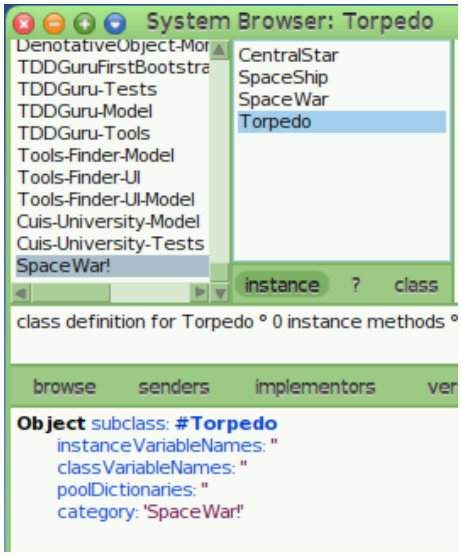

Figura 2.2: Categoría de clases Spacewar!

2.6.2 Paquete Spacewar!

Otro uso importante de una categoría de clase es definir un paquete para guardar código en un archivo. Un paquete guarda el código de las clases contenidas en una categoría de clase determinada y algo más, pero hablaremos de eso más adelante. Para crear nuestro paquete `Spacewar!` y guardar el código del juego, utilizamos la herramienta *Installed Packages* (Paquetes instalados).

1. Abrir la herramienta *Installed Packages* ...menú World → `Open...` → `Installed Packages`...

2. En la ventana de Installed Packages, haz ...click en botón `new` → Input *Spacewar!* → *`Return`*...

3. haz ...seleccionar paquete `Spacewar!` → botón `save`...

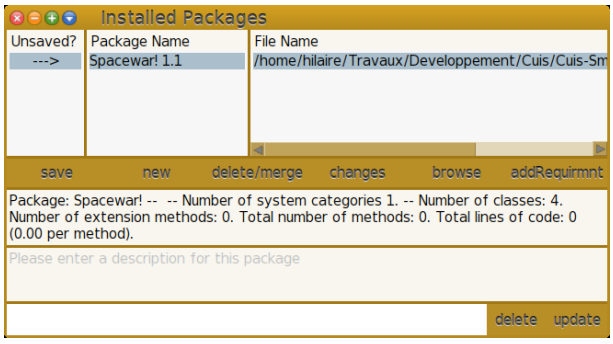

Figura 2.3: Ventana Installed Package

Se crea un fichero `Spacewar!.pck.st` junto al archivo de imagen Cuis-Smalltalk. Para instalar un paquete en un entorno nuevo de Cuis-Smalltalk, utiliza la herramienta `File List` (lista de archivos):

1. Haz ...menú World → `Open...` → `File List`...

2. Busca el archivo `Spacewar!.pck.st` y click el botón `install package`

También puedes arrastrar y soltar el archivo del paquete desde tu sistema operativo a la ventana de Squeak. Al soltar el archivo sobre la ventana, Cuis-Smalltalk te preguntará qué quieres hacer con este paquete. Para instalarlo en tu entorno, simplemente pulsa `Install package` (Instalar paquete).

O, puedes abrir un Workspace, teclear `Feature require: 'Spacewar!'` y *`Ctrl-d`* **D***o it*.

Ahora, hemos creado y guardado el paquete. Cada vez que inicies un nuevo entorno Cuis-Smalltalk, podrás cargar el paquete del juego.

Las clases que hemos definido son estructuras vacías sin estado ni comportamiento. Estas se completarán y refactorizarán en los siguientes capítulos.

2.6.3 El modelo newtoniano

Para disfrutar de una experiencia de juego agradable, las naves de los jugadores deben seguir las leyes del movimiento de Newton. La aceleración, la velocidad y la posición se calculan según estas leyes. Las naves están sometidas a dos fuerzas: la aceleración provocada por la fuerza gravitatoria de la estrella central y una aceleración interna procedente de los motores de la nave.

Más adelante, aprenderemos cómo estas ecuaciones se convierten fácilmente en cálculos informáticos.

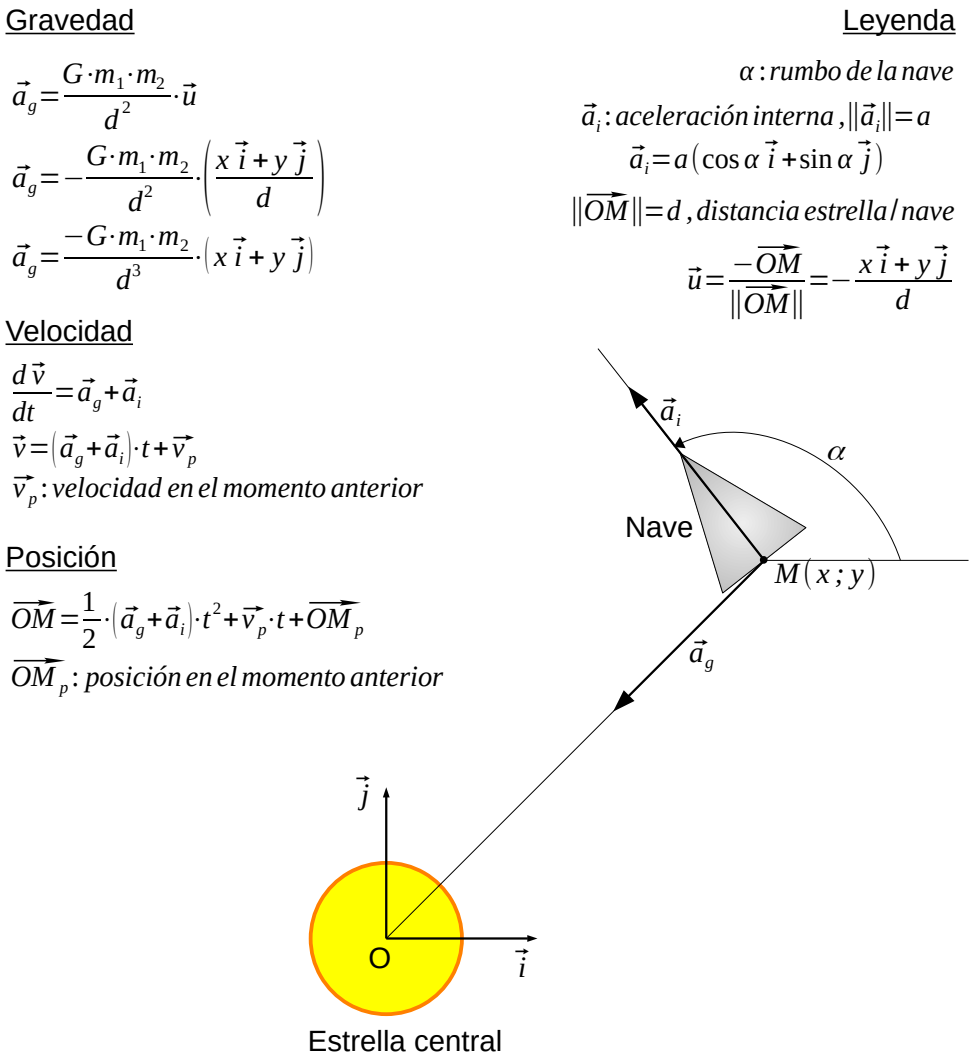

Figura 2.4: Ecuaciones de las aceleraciones, velocidad y posición

3 Clase - Modelo de Comunicación entre Entidades

> Si te doy algo con lo que puedes jugar y ampliar, aunque sea un trozo de papel con un párrafo y te digo que no está bien escrito, reescríbelo, eso es más fácil que no darte nada y decirte que crees algo; es decir, darte una hoja en blanco y empezar a escribir. de cero Así que lo bueno, que se ha demostrado tanto para los programadores profesionales como para los niños, es que cuando empiezas con algo, un objeto que hace algo, y luego juntas muchos objetos como ese y haces que interactúen, y luego los amplías y haces que se comporten de forma un poco diferente, puedes adoptar un enfoque muy gradual para aprender a controlar un sistema informático.
>
> —*Adele Goldberg*

Cuis-Smalltalk es un lenguaje de programación orientado a objetos (POO) puro. Todas las entidades del lenguaje: números enteros, números decimales, números racionales, cadenas, colecciones, bloques de código, etc., cada instancia utilizable como sustantivo en Smalltalk, es un objeto.

3.1 Comprendiendo la Programación Orientada a Objetos

Pero, ¿qué es un objeto?

Sencillamente, un objeto tiene dos componentes:

- **Estado interno.** Esto se materializa en variables que solo conoce el objeto. Una variable que solo es visible dentro del objeto se denomina variable *privada*. Como consecuencia, es imposible —si el objeto así lo decide— conocer el estado interno del objeto desde otro objeto.
- **Un repertorio de comportamientos.** Estos son los mensajes a los que responde una instancia objeto. Cuando el objeto recibe un mensaje que entiende, obtiene su comportamiento de un método con ese nombre conocido por su clase o superclase.

El nombre del método se denomina *selector* porque se utiliza para seleccionar qué comportamiento se invoca. Por ejemplo, en `'hello' at: 1 put: $B`, el método invocado tiene el selector `#at:put:` y los argumentos `1` y `$B`. Todos los selectores son símbolos.

Las instancias de objetos se crean (se *instancian*) siguiendo un modelo o plantilla. Este modelo se conoce como su *Class* (clase). Todas las instancias de una clase comparten los mismos métodos y, por lo tanto, reaccionan de la misma manera.

Por ejemplo, hay una clase `Fraction`, pero muchas fracciones (1/2, 1/3, 23/17, ...) que se comportan tal y como esperamos que lo hagan las fracciones. La clase `Fraction` y las clases de las que hereda definen este comportamiento común, tal y como describiremos a continuación.

Una clase determinada declara sus variables internas –estados– y el comportamiento mediante la implementación de los métodos. Una variable es básicamente un casillero con nombre que puede contener cualquier objeto. Cada variable de instancia de una clase obtiene su propio casillero con el nombre común.

Veamos cómo se declara la clase `Fraction`:

```
Number subclass: #Fraction
```

```
instanceVariableNames: 'numerator denominator'
classVariableNames: ''
poolDictionaries: ''
category: 'Kernel-Numbers'
```

Como era de esperar, hay dos variables, –denominadas *variables de instancia*–, para definir el `numerator` (numerador) y el `denominator` (denominador) de una fracción. Cada instancia de fracción tiene su propio numerador y su propio denominador.

A partir de esta declaración, observamos que existe una jerarquía en la definición de la clase: `Fraction` es un tipo de `Number`. Esto significa que una fracción hereda el estado interno (variables) y el comportamiento (métodos) definidos en la clase `Number`. `Fraction` se denomina *subclase* de `Number`, por lo que, naturalmente, denominamos a `Number` *superclase* de `Fraction`.

Una clase especifica el comportamiento de todas sus instancias. Es útil poder decir *este objeto es como aquel objeto, pero con estas diferencias*. En Smalltalk lo conseguimos haciendo que las clases hereden el estado y el comportamiento de las instancias de su clase padre. Esta clase hija, o subclase, sólo necesita especificar el estado y el comportamiento de sus instancias que difieren de su clase padre, conservando todos los comportamientos no modificados.

Este aspecto de la programación orientada a objetos se denomina *herencia*. En Cuis-Smalltalk, cada clase hereda de una clase padre.

En Smalltalk, decimos que cada objeto decide por sí mismo cómo responder a un mensaje. Esto se denomina *polimorfismo*. El mismo selector de mensajes puede enviarse a objetos de diferentes clases. La *forma* (morf) del cálculo será diferente dependiendo de la clase específica de las *muchas* (poli) clases posibles de objeto que entienden el mensaje.

Diferentes tipos de objetos responden al mismo mensaje `#printString` de maneras diferentes, pero adecuadas.

Ya hemos visto las fracciones. Esas fracciones son objetos llamados *instancia*s de la clase `Fraction`. Para crear una instancia escribimos `5 / 4`, el mecanismo se basa en el envío de mensajes y el polimorfismo. Veamos cómo funciona.

El número `5` es un entero que recibe el mensaje `#/`, por lo tanto, al observar el método `/` en la clase `Integer`, podemos ver cómo se instancia la fracción. Vea parte de este método:

```
/ aNumber
"Refer to the comment in Number / "
| quoRem |
aNumber isInteger ifTrue:
../..
   ifFalse: [↑ (Fraction numerator: self denominator: aNumber) reduced]].
../..
```

A partir de este código fuente, apreciamos que, en algunas situaciones, el método devuelve una fracción reducida. Podemos esperar que, en otras situaciones, se devuelva un número entero, por ejemplo, `6 / 2`.

En el ejemplo, observamos que el mensaje `#numerator:denominator:` se envía a la clase `Fraction`, dicho mensaje hace referencia a un *método de clase* que solo entiende la clase `Fraction`. Se espera que dicho método devuelva una instancia de `Fraction`.

Prueba esto en un workspace:

```
Fraction numerator: 24 denominator: 21
⇒ 24/21
```

Fíjate cómo la fracción resultante no se reduce. Sin embargo, sí se reduce cuando se instancia con el mensaje `#/`:

```
 24 / 21
⇒ 8/7
```

Con frecuencia se utiliza un método de clase para crear una nueva instancia a partir de una clase. En el Ejemplo 4.7, se envía el mensaje `#new` a la clase `OrderedCollection` para crear una nueva colección vacía; `new` es un método de clase.

En el Ejemplo 4.8, el mensaje `#newFrom:` se envía a la clase `OrderedCollection` para crear una nueva colección llena de elementos de la matriz dada en el argumento; `newFrom:` es otro método de clase.

Ahora observa la jerarquía de la clase `Number`:

```
Number
  Float
    BoxedFloat64
    SmallFloat64
  Fraction
  Integer
    LargePositiveInteger
      LargeNegativeInteger
    SmallInteger
```

`Float`, `Integer` y `Fraction` son descendientes directos de la clase `Number`. Ya hemos visto el mensaje `#squared` enviado a instancias de enteros y fracciones:

```
16 squared ⇒ 256
(2 / 3) squared ⇒ 4/9
```

Como el mensaje `#squared` se envía a instancias `Integer` y `Fraction`, el método asociado se denomina *método de instancia*. Este método se define tanto en la clase `Number` como en la clase `Fraction`.

Veamos este método en `Number`:

```
Number>>squared
"Answer the receiver multiplied by itself."
   ↑ self * self
```

En el código de un método de instancia, `self` se refiere al propio objeto, en este caso el valor del número. El símbolo ↑ (también `^`) indica que se debe *devolver* el siguiente valor `self * self`. Se podría pronunciar `^` como «return» (devolver).

Veamos ahora este mismo método en `Fraction`:

```
Fraction>>squared
   ↑ Fraction
      numerator: numerator squared
      denominator: denominator squared
```

Aquí se instancia una nueva fracción con el numerador y el denominador de la instancia original al cuadrado. Este método `squared` alternativo garantiza que se devuelva una instancia de fracción.

Cuando se envía el mensaje `#squared` a un número, se ejecutan diferentes métodos dependiendo de si el número es una fracción u otro tipo de número. El polimorfismo significa que la clase de cada instancia decide cómo responderá a un mensaje concreto. Aquí, la clase `Fraction` está *sobrescribiendo* (*overriding*) el método `squared`, definido anteriormente en la jerarquía de clases. Si un método no se sobrescribe, se invoca un método heredado para responder al mensaje.

Sin salir de la jerarquía `Number`, veamos otro ejemplo de polimorfismo con el mensaje `#abs`:

```
-10 abs ⇒ 10
5.3 abs ⇒ 5.3
(-5 / 3) abs ⇒ 5/3
```

La implementación en `Number` no necesita mucha explicación. Hay un `#ifTrue:ifFalse:` que aún no hemos comentado, pero el código se explica por sí mismo:

```
Number>>abs
"Answer a Number that is the absolute value (positive magnitude) of the
receiver."
   self < 0
      ifTrue: [↑ self negated]
      ifFalse: [↑ self]
```

Esta implementación funcionará perfectamente para las subclases de `Number`. Sin embargo, hay varias clases que la sobrescriben para casos especializados u optimizados.

Por ejemplo, en lo que respecta a los números enteros positivos grandes, `abs` está vacío. De hecho, **en ausencia de un valor devuelto explícitamente, el valor devuelto por defecto es la propia instancia**, en nuestro caso la instancia `LargePositiveInteger`:

```
LargePositiveInteger>>abs
```

El `LargeNegativeInteger` sabe que es negativo y que su valor absoluto es él mismo, pero con el signo invertido, es decir, `negated` (negado):

```
LargeNegativeInteger>>abs
   ↑ self negated
```

Estos dos métodos sobrescritos son más eficientes, ya que evitan comprobaciones innecesarias y ramificaciones ifTrue/ifFalse. El polimorfismo se utiliza a menudo para evitar comprobaciones innecesarias y ramificaciones de código.

Si selecciona el texto `abs` en un navegador o espacio de trabajo y hace clic con el botón derecho para abrir el menú contextual, encontrará una entrada llamada `Implementors of it`. Puede seleccionarla o utilizar `Ctrl-m` (*i***M***plementors*) para ver cómo los distintos métodos de '#abs' utilizan el polimorfismo para especializar su respuesta y producir el resultado esperado de forma natural.

Dado que una instancia de objeto se modela mediante su clase, es posible preguntar a cualquier objeto cuál es su clase con el mensaje `#class`. Fíjate atentamente en la clase devuelta en las líneas 2 y 3:

```
1 class ⇒ SmallInteger
(1/3) class ⇒ Fraction
(6/2) class ⇒ SmallInteger
(1/3) asFloat class  ⇒ SmallFloat64
(1.0/3) class ⇒ SmallFloat64
'Hello' class ⇒ String
('Hello' at: 1) class ⇒ Character
```

Ejemplo 3.1: Preguntando la clase de una instancia

3.2 Explorar POO desde el Browser

En la Figura 2.1 del Browser, debajo del panel de clases, hay tres botones:

- `instance`: para acceder a los **métodos de instancia** de la clase seleccionada. Estos métodos se aplican a todas y cada una de las instancias de la clase. Cada instancia reacciona a ellos.
- `?`: para acceder a la documentación –comentario– de la clase seleccionada.
- `class`: para acceder a los **métodos de clase** de la clase seleccionada. Estos métodos son accesibles sólo desde la propia clase. Sólo los objetos clase reaccionan a métodos de clase.

Observa el amplio panel de texto debajo de estos tres botones, que proporciona información contextual sobre el elemento seleccionado.

Una vez más, los métodos de clase se aplican a la propia clase. Los métodos de instancia se aplican a todas las instancias modeladas por la clase. Hemos visto anteriormente que la clase `Fraction` tiene un método `#numerator:denominator:` que se utiliza para obtener una nueva instancia de `Fraction`. Solo hay un objeto de clase `Fraction`. Los mensajes como `#squared` y `#abs` se envían a cualquier instancia de `Fraction`, de las que hay muchas.

Hasta ahora, hemos intentado ser muy cuidadosos con las definiciones, pero ya sabes que cuando decimos «una fracción» nos referimos a «una instancia de la clase `Fraction`». A partir de aquí, nuestro lenguaje será más informal.

Cuando se selecciona la clase `Float`*, ¿qué información proporciona el panel de texto?*

Ejercicio 3.1: Información de la clase Float

Hemos dedicado mucho tiempo a este tema porque es importante evitar la confusión entre los métodos de instancia y los métodos de clase. Consideremos la clase `Float` como ejemplo.

Métodos de clase. En la Figura 3.1 los métodos listados son del lado de la clase, con el botón `class` pulsado en el Browser, se puede ver esta lista.

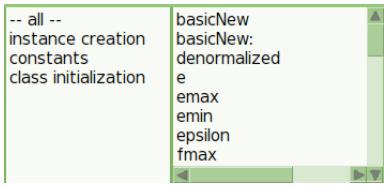

Figura 3.1: Métodos de clase en `Float`

Desde un Workspace, estos métodos se llaman con el nombre del mensaje directamente a la clase:

```
Float e
⇒ 2.718281828459045
Float epsilon
⇒ 2.220446049250313e-16
```

```
Float fmax
⇒ 1.7976931348623157e308
```

Habrás notado que el texto que escribes en el espacio de trabajo se colorea y resalta según lo que escribes. Hablaremos de esto más adelante, cuando tratemos el lenguaje Smalltalk, pero la idea es que resulte útil. Si empiezas a escribir una palabra que el espacio de trabajo de Cuis conoce, puedes pulsar la tecla ***tab*** y obtendrás un conjunto de opciones para completar la palabra. Prueba a escribir `Float epsi` y pulsa la tecla ***tab***. A continuación, pulsa la tecla ***enter*** y deberías ver `Float epsilon`. Haz clic en cualquier otro lugar del espacio de trabajo para que desaparezca este menú.

Sin embargo, no se puede enviar un mensaje de clase a una instancia de `Float`, ya que se produce un error y se abre la ventana roja del depurador. Por ahora, basta con cerrar la ventana del depurador para ignorar el resultado.

```
3.14 pi
⇒ MessageNotUnderstood: SmallFloat64>>pi
Float pi e
⇒ MessageNotUnderstood: SmallFloat64>>e
```

Con frecuencia, estos métodos de clase se utilizan para acceder a valores constantes, como se ve en el ejemplo anterior, o para crear una nueva instancia:

```
OrderedCollection new
⇒ Create a new empty ordered collection
Fraction numerator: 1 denominator: 3
⇒ 1/3 "a fraction instance"
Float new
⇒ 0.0
Float readFrom: '001.200'
⇒ 1.2
Integer primesUpTo: 20
⇒ #(2 3 5 7 11 13 17 19)
```

Métodos de instancia. En la Figura 3.2, los métodos listados son los de la parte instancia, con el botón `instance` pulsado en el Browser, aparece esta lista.

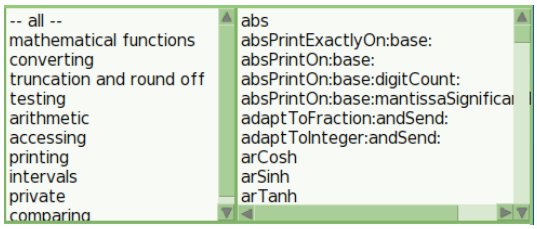

Figura 3.2: Métodos de instancia en `Float`

En un Workspace, estos métodos se llaman con el nombre del mensaje enviado directamente a una instancia:

```
-10.12 abs ⇒ 10.12
3.14 cos ⇒ -0.9999987317275395
-10.12 * 2 ⇒ -20.24
```

Los mensajes a métodos de instancia no se pueden enviar directamente a una clase, necesitas primero instanciar un objeto:

```
Float cos
⇒ MessageNotUnderstood: Float class>>cos
Fraction squared
⇒ MessageNotUnderstood: Fraction class>>squared
OrderedCollection add: 10
⇒ MessageNotUnderstood: OrderedCollection class>>add:
```

Por supuesto, puedes mezclar tanto métodos de clase como de instancia, mientras envíes el mensaje correcto a la clase o la instancia:

```
Float pi cos
⇒ -1.0
Float e ln
⇒ 1.0
(Fraction numerator: 4 denominator: 5) squared
⇒ 16/25
OrderedCollection new add: Float pi; add: Float e; yourself
⇒ an OrderedCollection(3.141592653589793 2.718281828459045)
```

Otro ejemplo desde Spacewar! mezclando métodos de clase e instancia. Este trozo de código actualiza la orientación de un torpedo de acuerdo con su vector de velocidad:

```
self rotation: (velocity y arcTan: velocity x) + Float halfPi
```

Ejemplo 3.2: Alineando un torpedo con su dirección de velocidad

Con esta breve introducción al browser del sistema estás preparado para explorar el sistema de clases.

3.3 Sistema de clases de Cuis

Como hemos señalado anteriormente, Cuis-Smalltalk es un entorno puramente orientado a objetos. Esto significa que cada entidad con la que se trabaja se representa como una instancia de una clase escrita en el propio Cuis-Smalltalk. Como consecuencia directa, Cuis-Smalltalk está escrito en su mayor parte en sí mismo. Esto significa que todo el sistema está abierto para que lo aprendas y juegues con él.

Lo que llamamos clases de sistema son modelos de objetos fundamentales. En otros lenguajes de programación, estos se implementarían en la biblioteca estándar de ese lenguaje.

En un sistema verdaderamente abierto, no existe una distinción real entre las clases de sistema y las clases de usuario, pero nos ayudará a delimitar los objetos más utilizados. Veamos una breve introducción a algunas clases fundamentales de Smalltalk y sus métodos más importantes.

En el panel superior izquierdo del navegador, las categorías de clases importantes para empezar son:

- **Kernel-Numbers**. Relacionado con las diferentes representaciones numéricas y cálculos, incluyendo funciones matemáticas, conversiones, intervalos e incluso iteraciones.
- **Kernel-Text**. Relacionado con los caracteres y la cadena como colección de caracteres.
- **Collections-Abstract**, **Collections-Unordered**, **Collections-Sequenceable**, **Collections-Arrayed**. Relativos a Array, Dictionary, Set, OrderedCollection y muchos más. Esta categoría incluye acceso normal, enumeración, funciones matemáticas y ordenación.

3.4 Kernel-Numbers

La clase jerárquica superior `Number` muestra la mayoría de los comportamientos heredados por las subclases como `Float`, `Integer` y `Fraction`. La forma de Smalltalk de aprender sobre un comportamiento es apuntar el navegador del sistema hacia una clase jerárquica superior y explorar las categorías de métodos.

Supongamos que queremos redondear un número flotante. En `Number`, exploramos la categoría **Truncation and round off method** (Métodos de truncamiento y redondeo) para descubrir varios comportamientos. Lo siguiente que debemos hacer es probar estos mensajes en un espacio de trabajo para descubrir el que estamos buscando:

```
1.264 roundTo: 0.1 ⇒ 1.3
1.264 roundTo: 0.01 ⇒ 1.26
1.264 roundUpTo: 0.01 ⇒ 1.27
1.264 roundTo: 0.001 ⇒ 1.264
```

Ejemplo 3.3: Redondeando números, pruebas en el Workspace

Number es un lugar muy extraño para buscar un bucle indexado en un intervalo determinado. Sin embargo, un intervalo se define por los números inicial y final. En la clase `Number`, la categoría de métodos **intervals** (intervalos) revela comportamientos relacionados. Estos métodos funcionan de forma polimórfica con la mayoría de los tipos de números:

```
1 to: 10 do: [:i | Transcript show: 1 / i; space]
⇒ 1 (1/2) (1/3) (1/4) (1/5) (1/6) (1/7) (1/8) (1/9) (1/10)

1 to: 10 by: 2 do: [:i | Transcript show: 1 / i; space]
⇒ 1 (1/3) (1/5) (1/7) (1/9)

1/10 to: 5/3 by: 1/2 do: [:i | Transcript show: i; space]
⇒ (1/10) (3/5) (11/10) (8/5) (1/10) (3/5) (11/10) (8/5)

Float pi to: 5 by: 1/3 do: [:i | Transcript show: i; space]
⇒ 3.141592653589793 3.4749259869231266 3.80825932025646
4.141592653589793 4.474925986923126 4.808259320256459
```

Ejemplo 3.4: Bucles de intervalo (for-loop)

Ahora, en la clase `Integer`, busca la categoría de métodos `enumerating`, para encontrar `timesRepeat:`. Cuando un trozo de código necesita ser ejecutado varias veces[1], sin necesidad de un índice, se envía el mensaje `#timesRepeat:` a un entero. Ya hemos visto

[1] Más estrictamente, repetir un número entero de veces.

esta variante en una sección anterior de este capítulo. Lanzar un dado de 6 caras 5 veces se puede simular con un número entero:

```
5 timesRepeat: [Transcript show: 6 atRandom; space]
⇒ 1 2 4 6 2
```

Ejemplo 3.5: Lanzar un dado 5 veces

Note: Espera un resultado distinto cada vez.

Se pueden definir intervalos de números por separado, para utilizarlos en el futuro:

```
1 to: 10
⇒ (1 to: 10)

1 to: 10 by: 2
⇒ (1 to: 9 by: 2)
```

Ejemplo 3.6: Intervalos

Los intervalos también funcionan con otro tipo de objetos como `Character`:

```
$d to: $h
⇒ #($d $e $f $g $h)
```

De hecho, un intervalo es un objeto por sí mismo. Es una especie de colección:

```
(1 to: 10) class
⇒ Interval

(1 to: 10 by: 2) squared
⇒ #(1 9 25 49 81)

(1 to: 10) atRandom
⇒ 4 "different result each time"
```

En Spacewar!, cuando una nave se destruye se la teletransporta a una dirección al azar en el cuadrado del área de juego. Los intervalos son útiles para seleccionar coordenadas aleatorias. En el ejemplo de abajo, la variable `randomCoordinate` contiene un bloque de código –llamado función anónima en otros lenguajes–. Selecciona un valor aleatorio en un intervalo consistente en los extremos izquierdo y derecho del área de juego:

```
randomCoordinate ← [(area left to: area right) atRandom].
aShip
   velocity: 0 @ 0;
   morphPosition: randomCoordinate value @ randomCoordinate value
```

Ejemplo 3.7: Teletransportar una nave

Calcular los valores de los cosenos en el intervalo [0 ; 2PI], cada 1/10. Mostrando el resultado en el Transcript.

Ejercicio 3.2: Tabla de cosenos

Los números enteros se representan en diferentes bases cuando se les antepone la base y letra «r». La `r` significa raíz, la base raíz con la que se interpreta el número siguente. Cuando ejecutamos e imprimimos con *`Ctrl-p`* en cualquier número de este tipo, se muestra inmediatamente en la base decimal:

```
2r1111 ⇒ 15
16rF ⇒ 15
8r17 ⇒ 15
20rF ⇒ 15
10r15 ⇒ 15
```

Ejemplo 3.8: Entero representado en diversas bases

Escribiendo números como Mayas o Babilonios[2]:

```
"The Babylonians"
60r10 ⇒ 60
60r30 ⇒ 180
60r60 ⇒ 360
60r30 + 60r60 ⇒ 540
(60r30 + 60r60) printStringRadix: 60 ⇒ '60r90'

"The Mayans"
20r10 ⇒ 20
20r40 ⇒ 80 "pronounced 4-twenties in some languages"
20r100 ⇒ 400
```

Ejemplo 3.9: Contando como los antiguos

Debido a la naturaleza de un número representado en base 2, desplazar sus bits hacia la izquierda y hacia la derecha equivale a multiplicar por 2 y dividir por 2:

```
(2r1111 << 1) printStringBase: 2 ⇒ '11110'
2r1111 << 1 ⇒ 30
(2r1111 >> 1) printStringBase: 2 ⇒ '111'
2r1111 >> 1 ⇒ 7
```

Ejemplo 3.10: Desplazando bits

[2] La representación de números en base 20 o 60 no es exclusiva de estas civilizaciones, aunque son los casos de uso más documentados.

¿Cómo podemos multilicar el entero 360 por 1024, sin utilizar la operación de multiplicación?

Ejercicio 3.3: Multiplicar por 1024

Interrupción con números decimales

Hemos visto que los números decimales se escriben con un punto «.» que separa la parte entera de la decimal: `1.5`, `1235.021` o `0.5`. El número `0.0000241` se escribe más fácilmente con la notación científica `2.41e-5`; que significa 2 precedido por 5 ceros o 2 como el quinto dígitos después del punto decimal.

Los ordenadores codifican y almacenan los números decimales de manera imprecisa. Debes prevenirte cuando hagas cálculos y comparaciones de igualdad. Muchos sistemas ocultas estos errores porque son muy pequeños. Cuis-Smalltalk no oculta esta imprecisión. Hay mejor información sobre esto en el comentario de la clase `Float`.

```
0.1 + 0.2 - 0.3
⇒ 5.551115123125783e-17
```

Ejemplo 3.11: ¡Discalculia del ordenador!

En el Ejemplo 3.11, el valor devuelto debería ser cero, pero no es el caso. El ordenador devuelve `5.55e-17`, o `0.0000000000000000555`, que es muy cercano a cero, pero que es un error.

Realiza 3 cálculos mostrando errores comparando con los valores esperados.

Ejercicio 3.4: Errores de cálculo con números decimales

Cuando la precisión esté obligada en Cuis-Smalltalk se utiliza la representación como números racionales.

Un número racionas se escribe con el símbolo de división entre dos enteros: haz *`Ctrl-p`* en `5/2` ⇒ `5/2`. Cuis-Smalltalk devuelve un número racional, no calcula un decimal.

¿Qué ocurre cuando ejecutamos este código `5/0`*?*

Ejercicio 3.5: Hacia el infinito

Regresemos a la discalculia de nuestro ordenador con los números decimales. Cuando utilizamos números racionales, el Ejemplo 3.11 se convierte en:

```
(1/10) + (2/10) - (3/10) ⇒ 0
```

Ejemplo 3.12: ¡El cálculo es correcto utilizando fracciones!

Esta vez obtenemos el resultado esperado. Debajo de la cubierta, el ordenador sólo realiza cálculos con componentes enteros por lo que no hay resultados redondeados. Esto es un buen ejemplo de que resolver algunos problemas necesita cambiar el paradigma.

Regresa a Ejercicio 3.4 y utilizar números racionales para representar los números decimales. Los errores deberían estar resueltos.

Ejercicio 3.6: Arreglando errores

Cuis-Smalltalk sabe cómo convertir un número decimal a una fracción, enviando el mensaje `#asFraction`. Ya hemos señalado la discalculia del ordenador con los números decimales, esto es por lo que podemos obtener resultados extraños cuando pedimos una fracción. La representación interna de `1.3` no es exactamente como parece:

```
(13/10) asFloat
⇒ 1.3

(13/10) asFloat asFraction
⇒ 5854679515581645/45035996273704
```

3.5 Kernel-Text

Cabe destacar, que esta categoría contiene las clases `Character`, `CharacterSequence`, `String` y `Symbol`. Las instancias `String` son colecciones de instancias `Character`. Todas ellas no se limitan al pequeño conjunto de caracteres ASCII, al contrario pueden manejar cualquier carácter conjunto de caracteres Unicode asociados con Códigos Unicode. Como se ha indicado anteriormente, las clases Unicode también pueden manejar ASCII, y son intercambiables. Por lo general, no es necesario preocuparse por el tipo real (ASCII o Unicode) que tiene una instancia.

Character/UnicodeCodePoint. Un carácter individual se escribe con un prefijo «$», por ejemplo: `$A` o `$`ϕ. Se puede definir con el método de clase `codePoint:`. Observa que puedes obtener el carácter de cualquier carácter Unicode utilizanzo Unicode Code Point:

```
Character codePoint: 65 ⇒ $A
Character codePoint: 966 ⇒ $ϕ
```

Hay métodos de clase para caracteres no imprimibles: `Character tab`, `Character lf`, etc.

Además, `Character` define `#namedCharactersMap`, que te permite entrar introducir varios caracteres Unicode fácilmente, como:

`\leftright` luego pulsa la barra espaciadora.

Como una cadena es una colección de caracteres, cuando iteramos una cadena podemos utilizar los métodos de instancia de `Character`:

```
'There are 12 apples.' select: [:c |c isDigit].
⇒ '12'
```

Ejemplo 3.13: Doce manzanas

Modifica el Ejemplo 3.13 para rechazar los caracteres numéricos.

Ejercicio 3.7: Seleccionar manzanas

String. `String` es una clase muy grande, viene con más de 200 métodos. Es muy útil examinar estas categorías de métodos para ver las formas habituales de agruparlos.

A veces es posible que no veas inmediatamente una categoría relacionada con lo que estás buscando.

Si esperas que un selector de método comience por una letra específica, click-select la categoría de métodos `-- all --`, luego mueve el cursos sobre el panel de lista de los nombres de método. Pulsa ese carácter, p. ej. `$f`. Esto desplazará el panel de métodos hasta el primer método cuyo nombre comience con la letra «f».

Considera el caso en el que necesitas buscar una subcadena, una cadena dentro de otra cadena: cuando explores la clase `String`, busca categorías de métodos con nombres como **finding...** o **accessing**. Allí encontrarás una familia de métodos `findXXX`. Lee los comentarios al principio de estos métodos:

```
findString: subString
   "Answer the index of subString within the receiver, starting at
   start. If the receiver does not contain subString, answer 0."
   ↑ self findString: subString startingAt: 1.
```

O:

```
findString: key startingAt: start caseSensitive: caseSensitive
   "Answer the index in this String at which the substring key first
   occurs, at or beyond start.  The match can be case-sensitive or
```

```
not.  If no match is found, zero will be returned."
../..
```

A continuación, prueba los mensajes que puedan resultar interesantes en un Workspace:

```
'I love apples' findString: 'love' ⇒ 3 "match starts at position 3"
'I love apples' findString: 'hate'
⇒  0 "not found"
'We humans, we all love apples' findString: 'we'
⇒ 12
'We humans, we all love apples'
   findString: 'we'
   startingAt: 1
   caseSensitive: false
⇒ 1
'we humans, we all love apples' findString: 'we'
⇒ 1
'we humans, we all love apples' findString: 'we' startingAt: 2
⇒ 12
```

Seguir estos caminos te llevará, en la mayoría de los casos, hacia la respuesta que estás buscando.

Queremos dar formato a una cadena del tipo 'Joe compró XX manzanas y YY naranjas' para que quede así: 'Joe compró 5 manzanas y 4 naranjas'. ¿Qué mensaje deberíamos usar?

Ejercicio 3.8: Formatea una cadena

3.6 Spacewar! Estados y comportamientos

3.6.1 Los estados del juego

Después de definir las clases involucradas en el diseño del juego, ahora definimos varios estados de estas clases:

- Una instancia `SpaceWar` representando las necesidades del juego para conocerla `centralStar`, las `ships`, los `torpedoes` disparados y su `color`.
- Una `CentralStar` tiene un estado `masa`. Que necesitamos para calcular la fuerza de la gravedad aplicada a cada nave.
- Una instancia `SpaceShip` que conozca su nombre, las coordenadas de su posición (`position`), su ángulo de orientación (`heading`), su vector de velocidad (`velocity`), su estimación de combustible (`fuel`) , su cantidad de torpedos disponibles (`torpedoes`), su masa (`mass`) y su empuje de aceleración del motor (`acceleration`).
- Un `Torpedo` tiene los estados de posición (`position`), velocidad (`velocity`) y tiempo de vida (`lifeSpan`).

Necesitamos explicar la naturaleza matemática de estos estados y, a continuación, analizar su representación objetiva en las variables de instancia de nuestras clases.


En las siguientes secciones, para facilitar la lectura, escribiremos «la variable `myVar` es una `String`» en lugar de la correcta pero más engorrosa «la variable de instancia `myVar` es una referencia a una instancia de `String`».

SpaceWar

Este objeto es la entrada al juego. Queremos un nombre de clase significativo. Sus variables de instancia son los protagonistas involucrados en el juego:

- `centralStar` es la única `CentralStar` del juego. Necesitamos conocer sobre ella y poder pedirle su masa.
- `ships` es una colectión de dos naves para los dos jugadores. Es una instancia de `Array`, su tamaño fijo es de dos elementos.
- `torpedoes` es una colección de los torpedos disparados en el juego. Como su cantidad es variables, tiene sentido una `OrderedCollection` dinámica.

CentralStar

Es una variable de instancia única, `mass` es un número, lo más probable un entero (`Integer`).

SpaceShip

La nave es el objeto más complejo, algunas aclaraciones al echar un vistazo a sus variables.

- `name` (*nombre*) es una cadena `String`.
- `position` es una coordenada de pantalla 2D, una localización. Smalltalk utiliza la clase `Point` para representar este tipo de objetos. Comprende muchas operaciones matemáticas y de vectores; muy utilizados en cálculos mecánicos.

  Un punto se instancia muy fácilmente con un mensaje binario `#@` enviando a un número otro número como argumento: `100 @ 200` devuelve una instancia de `Point` representando las coordenadas (x;y) = (100;200).

  La posición (`position`) de la nave se recalcula periódicamente de acuerdo a la ley de marco de referencia galileano. El cálculo depende de la velocidad de la nave, su empuje de motor actual y la gravedad que tira desde la estrella central.
- `heading` es un ángulo en radianes, la dirección en la que apunta la nariz de la nave. Por lo tanto es un número `Float`. Cuando comienza el juego, las naves se orientan hacia arriba. El valor de su `heading` es -pi/2 radianes; el eje oy está orientado de arriba hacia abajo de la pantalla.
- `velocity` es un vector representando la velocidad instantánea de la nave. Es una instancia de `Point`.
- `fuel`'fuel' es un indicador, siempre que no sea cero, de que el jugador puede encender los motores de cohete de la nave proporcionándole aceleración para moverse alrededor del tirón de gravedad de la estrella central Es un número entero.
- `torpedoes` es la cantidad de torpedos disponibles para ser disparados por el jugador. Es un número entero (`Integer`).
- `mass` es un `Integer` representando la masa de la nave.
- `acceleration` es la norma de aceleración intrínseca que se proporciona a la nave cuando se encienden los motores. Por lo tanto es un número `Integer`.

Unas pocas palabras sobre las coordenadas euclidianas: el origen de nuestro marco ortonormal es la estrella central, su primer vector está orientado hacia la derecha de la pantalla y el segundo hacia la parte superior de la pantalla. Esta elección facilita el cálculo de la aceleración, la velocidad y la posición de la nave. Más adelante daremos información al respecto.

`Torpedo`

Un torpedo se lanza o «dispara» desde una nave con la velocidad iniciar relacionada a la velocidad de la nave. Una vez que el tiempo de vida del torpedo llega a cero, se autodestruye.

- `position` es una coordenada 2D en la pantalla, una instancia de `Point`. A diferencia de la nave, no acelera por gravedad de la estrella central. En efecto, un torpedo carece de un estado de masa. Para nuestro propósito es esencialmente cero. Su posición en el tiempo depende sólamente de la velocidad y su aceleración inicial.
- `heading` es un ángulo en radianes, la dirección en la que la nariz del torpedo apunta. Su valor coincide con el de la nave cuando se dispara. Por tanto también es un número `Float`.
- `velocity` es un vector que representa la velocidad instantánea del torpedo. Es constante durante el tiempo de vida del torpedo. De nuevo la velocidad es una instancia de `Point`,
- `lifeSpan` es un contador entero, cuando alcanza cero el torpedo se autodestruye.

3.6.2 Variables de instancia

En el capítulo anterior, explicamos cómo definir las cuatro clases `SpaceWar`, `CentralStar`, `SpaceShip` y `Torpedo`. En esta sección añadiremos a esas definiciones las variables de instancia –estados– que describimos arriba.

Para añadir las variables de la clase `Torpedo`, desde el Browser, selecciona esa clase. A continuación, añade los nombres de variables a la clave `instanceVariableNames:`, separadas por espacio. Por último, guarda la definición de la clase actualizada con el atajo *`Ctrl-s`*:

```
Object subclass: #Torpedo
   instanceVariableNames: 'position heading velocity lifeSpan'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Ejemplo 3.14: La clase `Torpedo` con sus variables de instancia

Añade las variables de instancias descritas arriba a las clases `SpaceWar`*,* `CentralStar` *y* `SpaceShip`*.*

Ejercicio 3.9: Variables de instancia de los protagonistas de Spacewar!

3.6.3 Comportamientos

Necesitamos acceder a algunos de estos estados desde otras entidades:

- Cuando inicializamos una nave espacial queremos establecer su nombre con un mensaje categorizado como *setter*: `ship name: 'The needle'`.
- Para calcular la fuerza de la gravedad aplicada a un objeto que tenga masa, necesitamos saber su valor con un mensaje unitario categorizado como *getter*: `star mass * ship mass`.

Para escribir estos comportamientos en el Browser primero selecciona la clase, luego la categoría de método que quieres, o si no quieres ninguna `-- all --`.

En el panel de código aparece la plantilla de método:

```
messageSelectorAndArgumentNames
   "comment stating purpose of message"
   | temporary variable names |
   statements
```

Ejemplo 3.15: Plantilla de método

Se describe a sí misma:

1. **Línea 1**. Es un nombre de método obligatorio, el mismo del mensaje.
2. **Línea 2**. Un comentario opcional entre comillas dobles.
3. **Línea 3**. Una lista opcional de variables locales del método entre barras verticales.
4. **Línea 4**. La subsiguiente lista de mensajes enviados y asignaciones.

El *getter* `mass` de `SpaceShip` está escrito como:

```
SpaceShip>>mass
   ↑ mass
```

La parte `SpaceShip>>` no es código válido y no se debe escribir en el Browser. Es una convención de texto para indicarle al lector que es un método de la clase `SpaceShip`.

Escribe unos mensajes getter de `SpaceShip` *para sus atributos de posición, velocidad y masa.*

Ejercicio 3.10: Mensajes *getter* de `SpaceShip`

Algunas variables de instancia se deben inicializar desde otra entidad, para esto es necesario un mensaje *setter*. Para establecer el nombre de una nave espacial podemos añadir el siguiente método:

```
SpaceShip>>name: aString
   name ← aString
```

El caracter ← es una asignación, significa que la variable de instancia `name` contiene un objeto `aString`. Para teclear este símbolo, teclea `_` y luego espacio, Cuis-Smalltalk lo convertirá en un símbolo de flecha izquierda. Alternativamente puedes escribir `name := aString`. Se podría pronunciar ← como «gets».

Dado que `name` es una variable de instancia, cada método de instancia sabe que debe utilizar el espacio para el nombre. Lo que significa que estamos colocando el valor del argumento `aString` en el espacio de la instancia llamado `name`.

Dado que cada elemento de la variable de instancia puede contener un objeto de cualquier clase, nos gusta nombrar el argumento para indicar que pretendemos que la variable `name` contenga una cadena, una instancia de la clase `String`.

La posición (`position`) y velocidad (`velocity`) de la nave, tanto como el rumbo (`heading`) del torpedo se deben establecer al inicio del juego o cuando la nave salta en el hiperespacio. Escribe los apropiados setters.

Ejercicio 3.11: Mensajes *setter* de `SpaceShip`

Fíjate que no tenemos un mensaje *setter* para el atributo `mass` (masa) de la nave espacial. De hecho, no tiene sentido cambiar la masa de una nave desde otro objeto. De hecho, si consideramos que ambas naves de los jugadores tienen la misma masa, deberíamos eliminar la variable `mass` y editar el método `#mass` para que devuelva un número literal:

```
SpaceShip>>mass
   ↑ 1
```

Ejemplo 3.16: Un método devolviendo una constante

Por otro lado, también podríamos considerar que la masa depende del combustible y los torpedos consumidos. Después de todo, el 93 % de la masa del cohete Saturno V estaba compuesta por su combustible. Hablaremos más sobre esto más adelante, en el [artRefactoring], página 85.

Controles

Una nave espacial controlada por el jugador comprende mensajes para ajustar su dirección y aceleración[3]:

Dirección. El rumbo de la nave se controla con los mensajes `#left` y `#right`. El primero reduce el rumbo en 0.1 y el segundo lo incrementa en 0.1.

Escribe dos métodos llamados `left` y `right`para modificar el rumbo de la nave en 0.1 de acuerdo con las indicaciones anteriores.

Ejercicio 3.12: Métodos para controlar el rumbo de la nave

Aceleración. Cuando se envía el mensaje `#push` a la nave, se encienden los motores y una aceleración interna de 10 unidades de aceleración se aplican a la nave. Cuando se envía el mensaje `#unpush`, la aceleración se detiene.

[3] La velocidad es una consecuencia de las aceleraciones aplicadas a la nave espacial.

Escribe dos métodos llamados `push` *y* `unpush` *para ajustar la aceleración interna de acuerdo con las indicaciones anteriores.*

Ejercicio 3.13: Métodos para controlar la aceleración de la nave

3.6.4 Inicializado

Cuando se crea una instancia, por ejemplo, `SpaceShip new`, se inicializa automáticamente: se envía el mensaje `#initialize` al objeto recién creado y se llama al método de instancia `initialize` correspondiente.

El proceso de inicialización es útil para establecer los valores predeterminados de las variables de instancia. Cuando creamos un nuevo objeto nave espacial, queremos establecer su posición, velocidad y aceleración predeterminadas:

```
SpaceShip>>initialize
   super initialize.
   velocity ← 0 @ 0.
   position ← 100 @ 100.
   acceleration ← 0
```

Ejemplo 3.17: Inicializar la nave espacial

En el método Ejemplo 3.17, observa la primera línea `super initialize`. Cuando se envía un mensaje a `super`, se refiere a la superclase de la clase del método. Hasta ahora, la clase padre de `SpaceShip` es `Object`, por lo que, para la inicialización, primero se llama al método `Object>>initialize`.

Cuando se crea una nave espacial se coloca en la parte superior y derecha de la estrella central. No tiene velocidad ni aceleración interna, solo la fuerza gravitatoria de la estrella central. Su proa apunta hacia la parte superior de la pantalla del juego.

Probablemente hayas notado que no hay código para inicializar el atributo `heading`, algo como `heading ←Float halfPi negated` para orientar la nave en la dirección de la parte superior de la pantalla. La verdad es que no necesitamos el `heading`, ya que esta información la proporcionará la clase `Morph`, que se utilizará más adelante como clase padre de `SpaceShip` y `Torpedo`. En este momento, se eliminará la variable `heading` y definiremos el comportamiento de la dirección con los métodos `heading` y `heading:` adecuados.

Escribe el método para inicializar la estrella central con 8000 unidades de masa.

Ejercicio 3.14: Inicializar la estrella central

4 El estilo de vida de la colección

> La simplicidad no precede a la complejidad, sino que la sigue.
> —*Alan Perlis*

Desde la introducción del concepto en los años 70, las colecciones y sus iteradores asociados han sido elementos importantes de la programación en Smalltalk. Si se utilizan correctamente, mejoran tanto la densidad del código como su comprensión, dos paradigmas que pueden parecer antagónicos. Desde entonces, estas innovaciones se han extendido a muchos lenguajes de programación populares.

4.1 String – una colección particular

La clase `String` también hereda el comportamiento de sus clases ancestrales. De hecho, `String` es una subclase de `CharacterSequence`, que a su vez es una subclase de `SequenceCollection`. La consecuencia directa es que, al buscar un comportamiento específico, es posible que también tengas que explorar las clases padre. El comportamiento completo de una clase, definido en la propia clase y en sus padres, se denomina su *protocolo*.

De nuevo, el Borwser es muy útil para explorar el protocolo de clase. Tienes dos opciones:

1. **Explorar el protocolo.** En el panel de clases del Browser, hacer ...selecciona la clase `String` → botón derecho del ratón → `Browse protocol (p)...` Como alternativa, utiliza el atajo de teclado *`Ctrl-p`*.

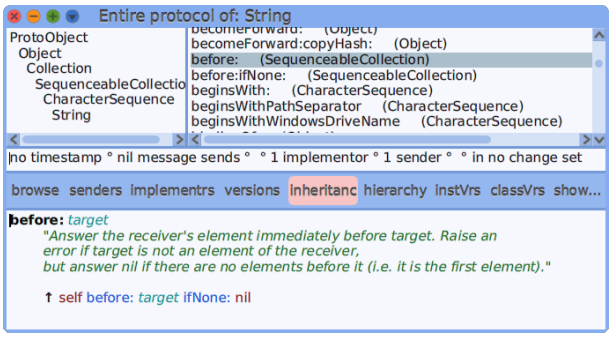

Figura 4.1: Ojear el protocolo de Browse `String`

La nueva ventana en un Browser de protocolos para la clase `String`. A la izquierda vemos la jerarquía con las clases ancestro de `String`. A la derecha están los selectores de método de las cadenas y, entre paréntesis, la clase en la que están definidos. Los métodos definidos en la propia clase `String` se muestran en negrita.

Al seleccionar una clase allí, solo se muestra el protocolo desde esa clase hasta la clase `String`. Si seleccionas `String` en el panel izquierdo, solo verá los métodos definidos en la propia clase `String`.

En la Figura 4.1,no está seleccionada ninguna clase específica, por lo tanto se muestra todo el protocolo de `String` a la derecha. Está seleccionado el método `before:` implementado en `SequenceableCollection` y su código fuente se muestra en el amplio panel inferior.

2. **Explorar la jerarquía.** En el panel de clases del Browser, hacer ...seleccionar la clase `String` → botón derecho del ratón → `Browse hierarchy (h)...` De manera alternativa, puedes usar la combinación de teclas *`Ctrl-h`* o el botón `hierarchy` en el browser del sistema.

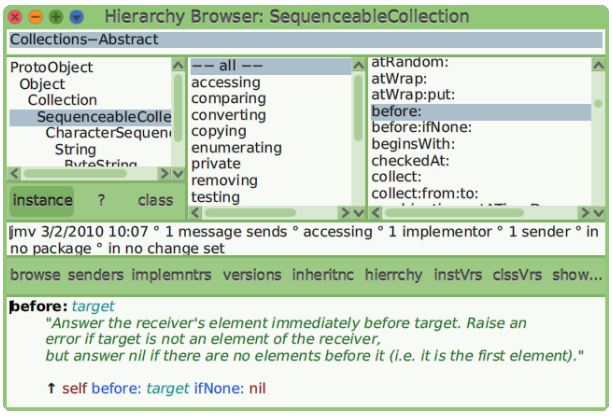

Figura 4.2: Ojear la jerarquía de `String`

El browser de jerarquía es muy similar al browser del sistema con dos diferencias:

- En el extremo izquierdo está ausente el panel de categorías de clases,
- En el panel de las clases, se muestra la jerarquía de `String`. Hace muy fácil visualizar las clases antecesoras o descendientes de `String`.

El browser de jerarquía es una herramienta general de exploracion. A diferencia del browser de protocolo, no muestra el protoco completo de una clase. No se muestran los métodos heredados, sólo los definidos directamente en la clase seleccionada. En la Figura 4.2, está seleccionada la clase `SequenceableCollection` en su método `before:`.

El método `before:` extrae de una colección el elemento anterior al elemento especificado. Cuando se hereda en la clase `String`, estos elementos son instancias de `Character`:

```
'1 + 3i' before: $i
⇒ $3
```

Practica las herramientas y resuelve el siguiente ejercicio.

Encuentra el método apropiado para transformar 'Hello My Friend' en 'My Friend'.

Ejercicio 4.1: Cortar una cadena

Ten en cuenta que algunos mensajes del protocolo `String` pueden no funcionar. Observa a continuación que se produce un error en una instancia `Character`:

```
'Hello My Friend' cos
⇒ MesageNotUnderstood: Character>>cos
```

Si observas los implementadores de `cos`, verás que `Collection` espera aplicar `cos` a cada miembro de una colección, por lo que se solicita el coseno de un carácter.

Símbolo. Un símbolo es muy similar a una cadena, pero es único y nunca se duplica. Dos referencias a `'hello'` pueden corresponder a dos objetos o a uno solo, dependiendo del historial computacional. Dos referencias a `#hello` siempre se refieren al mismo objeto.

Los símbolos reciben ese nombre porque se utilizan como *constantes simbólicas*. Ya has observado cómo en el libro escribimos los selectores de mensajes como símbolos. Utilizamos símbolos porque cada nombre de mensaje debe indexar de forma única el código de un método. Utilizarás un símbolo cuando necesites nombrar algo de forma única.

En el ejemplo siguiente, observa el mismo comportamiento con cadena:

```
'hello' == 'hello' copy
⇒ false
#hello  == #hello copy
⇒ true
```

Ahora ya sabes. Un símbolo no se puede duplicar ni cambiar. Los símbolos pueden contener espacios:

```
'hello my friend' asSymbol
⇒ #'hello my friend'
```

`Symbol` es una subclase de `String` y gran parte de su comportamiento es heredado. A medida que aprendemos sobre `String`s, también aprendemos bastante sobre los símbolos.

Ten en cuenta que muchos métodos @.class String@ están definidos para devolver cadenas.

```
'hello my friend' class.
⇒ String
#'hello my friend' class.
⇒ Symbol
#'hello my friend' asCamelCase
⇒ 'helloMyFriend'
#'hello my friend' asCamelCase asSymbol
⇒ #helloMyFriend
```

4.2 Diversión con variables

¿Cómo puede una variable ser divertida? Con Cuis-Smalltalk, una variable es el nombre de una caja que contiene un valor: un objeto, ¡eso es todo!

Una variable puede contener un valor de cualquier clase. El valor es fuertemente tipado (siempre podemos determinar su clase), pero la variable (caja) no está restringida a contener un valor de un único tipo.

Una consecuencia directa importante es que el *tipo* de una variable –es decir, la clase del objeto referenciado– puede cambiar en el tiempo. Observa este ejemplo:

```
| a |
a ← 1 / 3.
a class
⇒ Fraction
a ← a + (2 / 3)
⇒ 1
a class
```

```
⇒ SmallInteger
```

El valor iniciar de la variable `a` es una instancia de `Fraction`, después de algunos cálculos termina siendo una instancia de `SmallInteger`.

De hecho, no existe tal cosa como el tipo, solo hay objetos referenciados que pueden *mutar* con el tiempo en otro tipo de objeto: una estructura metálica a la que se le añaden dos ruedas puede convertirse en una bicicleta, o en un coche si se le añaden cuatro ruedas.

Por lo tanto, para declarar una variable de método, solo tenemos que nombrarla al principio del script y rodearla con caracteres de barra vertical «`|`».

Una variable siempre contiene un valor. Hasta que asignemos un valor diferente a una variable, esta contendrá el valor `nil`, una instancia de `UndefinedObject`. Cuando decimos que un valor está vinculado (*bound*) a una variable, queremos decir que el campo con ese nombre ahora contiene ese valor.

Hasta ahora hemos enviado mensajes directamente a objetos, pero también podemos enviar mensajes a una variable vinculada a un objeto.

Cualquier objeto responde al mensaje `#printString`.

```
| msg |
msg ← 'hello world!'.
Transcript show: msg capitalized printString, ' is a kind of '.
Transcript show: msg class printString; newLine.
msg ← 5.
Transcript show: msg printString, ' is a kind of '.
Transcript show: msg class printString; newLine.
```

Esta facilidad de uso tiene un inconveniente: al escribir código para enviar un mensaje a una variable vinculada a un objeto, el sistema no comprueba de antemano que el objeto comprenda el mensaje. No obstante, existe un procedimiento para detectar este tipo de situaciones cuando el mensaje se envía realmente.

4.3 Diversión con colecciones

Una colección es un conjunto de objetos. Los arrays y las listas son colecciones. Ya sabemos que una `String` es una colección; concretamente, una colección de caracteres. Muchos tipos de colecciones tienen comportamientos similares.

Un `Array` es una colección de tamaño fijo y, a diferencia de una cadena, puede contener cualquier tipo de literal entre `#( )`:

```
"array of numbers"
#(1 3 5 7 11 1.1)
"array of mixed literals"
#(1 'friend' $& 'al')
```

Un Array se construye directamente utilizando elementos *literales* bien formados. Llegaremos al significado de esta última afirmación cuando discutamos los detalles del lenguaje Smalltalk.

Por ahora, sólo observa cómo usando expresiones no literales al construir un array no funcionará como esperas:

```
#(1 2/3)
⇒ #(1 2 #/ 3)
```

De hecho, el símbolo `$/` se interpreta como un símbolo literal y obtenemos los componentes básicos de «2 / 3», pero este texto no se interpreta como una fracción. Para insertar una fracción en la matriz, se utiliza un *array en tiempo de ejecución* o un *array dinámico*, cuyos elementos son expresiones separadas por puntos y rodeadas por `{ }`:

```
{1 . 2/3 . 7.5}
⇒ #(1 2/3 7.5)
```

Con un array relleno de números puedes obtener información y operaciones aritméticas:

```
#(1 2 3 4) size ⇒ 4
#(1 2 3 4) + 10 ⇒ #(11 12 13 14)
#(1 2 3 4) / 10 ⇒ #(1/10 1/5 3/10 2/5)
```

Las operaciones matemáticas también funcionan:

```
#(1 2 3 4) squared ⇒ #(1 4 9 16)
#(0 30 45 60) degreeCos
⇒ #(1.0 0.8660254037844386
0.7071067811865475 0.49999999999999994)
```

También se pueden utilizar directamente métodos estadísticos básicos en arrays de números:

```
#(7.5 3.5 8.9) mean ⇒ 6.633333333333333
#(7.5 3.5 8.9) range ⇒ 5.4
#(7.5 3.5 8.9) min ⇒ 3.5
#(7.5 3.5 8.9) max ⇒ 8.9
```

Para obtener un array con los números naturales desde el 1 al 100, utilizaremos el mensaje `#to:`

```
(1 to: 100) asArray
⇒ #(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
92 93 94 95 96 97 98 99 100)
```

En esta línea de código, el mensaje `#to:` se envía a `1` con el argumento `100`. Devuelve un objeto intervalo. El mensaje `#asArray` enviado al intervalo devuelve un array.

Crea un array con los números enteros con un rango de -80 a 50.

Ejercicio 4.2: Números negativos enteros

El tamaño de un array es fijo, no puede crecer. Una `OrderedCollection` es una colección dinámica y ordenada. Crece cuando se añade un elemento con el mensaje `#add:`

```
| fibo |
fibo ← OrderedCollection newFrom: #(1 1 2 3).
fibo add: 5;
   add: 8;
   add: 13;
   add: 21.
fibo
⇒ an OrderedCollection(1 1 2 3 5 8 13 21)
```

Ejemplo 4.1: Colección de tamaño dinámico

El acceso al índice de los elementos de una colección se realiza mediante diversos mensajes. El índice abarca naturalmente desde 1 hasta el tamaño de la colección:

```
fibo at: 1 ⇒ 1
fibo at: 6 ⇒ 5
fibo last ⇒ 21
fibo indexOf: 2 ⇒ 3
fibo at: fibo size ⇒ 21
```

Jugando con enumeradores

Una colección incluye un conjunto de métodos útiles denominados enumeradores. Los enumeradores operan sobre cada elemento de una colección.

Las operaciones de conjunto entre dos colecciones se calculan con los mensajes `#union:`, `#intersection:` y `#difference:`.

```
#(1 2 3 4 5) intersection: #(3 4 5 6 7)
⇒ #(3 4 5)
#(1 2 3 4 5) union: #(3 4 5 6 7)
⇒ a Set(5 4 3 2 7 1 6)
#(1 2 3 4 5) difference: #(3 4 5 6 7)
⇒ #(1 2)
```

Ejemplo 4.2: Operaciones de conjunto

Construye la matriz de los números 1,...,24,76,...,100.

Ejercicio 4.3: Agujero en un conjunto

Las operaciones de conjunto funcionan con cualquier tipo de objeto. La comparación de objetos merece una sección aparte.

```
#(1 2 3 'e' 5) intersection: #(3.0 4 6 7 'e')
⇒ #(3 'e')
```

Para seleccionar los números primos del 1 al 100, utilizamos el enumerador `#select:`. Este mensaje se envía a una colección y, a continuación, selecciona cada elemento de la colección que devuelve verdadero a una condición de prueba:

```
(1 to: 100) select: [ :n | n isPrime ]
⇒  #(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
73 79 83 89 97)
```

Ejemplo 4.3: Seleccionar los números primos entre el 1 y 100

Este ejemplo introduce el mensaje `#select:` y un bloque de código, un elemento constitutivo primordial del modelo Cuis-Smalltalk. Un bloque de código, delimitado por corchetes, es una parte del código para su posterior ejecución. Expliquemos cómo se evalúa este script:

- `(1 to: 100)` se evalúa como un intervalo
- el bloque de código `[:n | n isPrime]` está instanciado (creado)
- the message `#select:` is sent to the interval with the block of code as the argument
- en el método `select:`, por cada entero del intervalo, se invoca el bloque de código con su parámetro `n` fijado al valor del entero. Un A *parámetro de bloque* comienza con «:» y es un identificar normal[1]. A continuación, cada vez que `n isPrime` se evalúa como verdadero, el valor `n` se añade a una nueva colección respondida cuando el método `select:` termina de comprobar cada elemento de la colección.

Un bloque de código se puede guardar en una variable, pasar como parámetro y utilizar varias veces.

```
| add2 |
add2 ← [:n| n + 2].
{ add2 value: 2. add2 value: 7 }.
⇒  #(4 9)
```

Los enumeradores implementan formas tremendamente potentes de procesar colecciones sin necesidad de un índice. Con esto queremos decir que son fáciles de usar correctamente. ¡Nos gusta lo sencillo!

Para hacerse una idea de lo útiles que son los enumeradores, eche un vistazo a la clase `Collection` en la categoría de métodos `enumerating`.

[1] Un *identificador* sólo es una palabra que comienza por una letra minúscula y está formada por letras minúsculas y mayúsculas y dígitos. Todos los nombres de variable son identificadores.

Selecciona los números impares entre -20 y 45.

Ejercicio 4.4: Enteros impares

Quieres saber cuántos números primos son menores de 100. Envía el mensaje `#size` a la colección respondida en el Ejemplo 4.3. Los paréntesis son obligatorios para asegurarnos de que `#size` se envía la última colección resultante:

```
 ( (1 to: 100) select: [:n | n isPrime] ) size ⇒
25
```

Ejemplo 4.4: Cuenta cuántos números primos hay entre 1 y 100

Para mayor claridad, utilizamos una variable llamada `primeNumbers` para almacenar la lista de números primos que hemos creado:

```
| primeNumbers |
primeNumbers ← (1 to: 100) select: [:n | n isPrime].
primeNumbers size
```

Modifica el Ejemplo 4.4 para calcular la cantidad de números primos entre 101 y 200.

Ejercicio 4.5: Números primos entre 101 y 200

Haz una lista de los múltiplos de 7 menores de 100..

Ejercicio 4.6: Múltiplos de 7

Construye una colección de los interos impares en [1 ; 100] que no sean primos.

Ejercicio 4.7: Impares y no primos

Una enumeración hermana de @.msg select:@ es @.msg collect:@. Devuelve una nueva colección del mismo tamaño, con cada elemento transformado por un bloque de código. Al buscar raíces cúbicas perfectas, es útil conocer algunos cubos:

```
(1 to: 10) collect: [:n | n cubed]
⇒ #(1 8 27 64 125 216 343 512 729 1000)
```

Ejemplo 4.5: Recoger cubos

Los elementos recopilados pueden ser de diferentes tipos. A continuación, se enumera una cadena y se recopilan números enteros:

```
'Bonjour' collect: [:c | c asciiValue ]
⇒  #(66 111 110 106 111 117 114)
```

Podemos cambiar el valor ASCII, convertirlo de nuevo en un carácter y luego recopilarlo en una nueva cadena. Es un cifrado sencillo:

```
'Bonjour' collect: [:c | Character codePoint: c asciiValue + 1 ]
⇒ 'Cpokpvs'
```

Ejemplo 4.6: Cifrado sencillo

Escribe un script para decodificar el cifrado 'Zpv!bsf!b!cptt', está codificado como el Ejemplo 4.6.

Ejercicio 4.8: Decodificar cifrado

El cifrado de César se basa en desplazar las letras hacia la derecha en el orden alfabético. El método lleva el nombre de Julio César, quien lo utilizó en su correspondencia privada con un desplazamiento de 3.

Escribe un script para generar el alfabeto en letras mayúsculas que representa el cifrado Caesar. La respuesta esperada es #($D $E $F $G $H $I $J $K $L $M $N $O $P $Q $R $S $T $U $V $W $X $Y $Z $A $B $C).

Ejercicio 4.9: Alfabeto del cifrado Caesar

Una vez que hayas descifrado correctamente el alfabeto, podrás codificar tu primer mensaje:

Codifica la frase 'SMALLTALKEXPRESSION'.

Ejercicio 4.10: Codifica con el cifrado Caesar

Y decodifica el mensaje:

Decodifica esta famosa cita atribuida a Julio Caesar 'DOHDMDFWDHVW'.

Ejercicio 4.11: Decodifica con el cifrado de Caesar

Diversión con bucles

La recopilación se puede iterar con bucles tradicionales: existe toda una familia de bucles *repeat*, *while* y *for*.

Un bucle *for* simple entre dos valores enteros se escribe con la palabra clave `#to:do:`, el último argumento es un bloque de código que se ejecuta para cada índice:

```
| sequence |
sequence ← OrderedCollection new.
1 to: 10 do: [:k | sequence add: 1 / k].
sequence
⇒ an OrderedCollection(1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10)
```

Ejemplo 4.7: Un bloque *for*

Sin embargo, un recolecta escribe más concisamente:

```
(1 to: 10) collect: [:k | 1 / k]
```

Un paso con un valor diferente a 1, se inserta un tercer argumento numérico:

```
1 to: 10 by: 0.5 do: [:k | sequence add: 1 / k]
```

Un bucle repetido sin índice ni colección alguna se escribe con el mensaje `#timesRepeat:` enviado a un entero:

```
| fibo |
fibo ← OrderedCollection newFrom: #(1 1).
10 timesRepeat: [
   fibo add: (fibo last + fibo atLast: 2)].
fibo
⇒ an OrderedCollection(1 1 2 3 5 8 13 21 34 55 89 144)
```

Ejemplo 4.8: Un bucle *repeat*

El cociente de los términos consecutivos de Fibonacci converge hacia el valor áureo.:

```
fibo pairsDo: [:i :j |
   Transcript show: (j / i ) asFloat ; cr]
⇒ 1.0
⇒ 1.5
⇒ 1.6
⇒ 1.6153846153846154
⇒ 1.6176470588235294
⇒ 1.6179775280898876
```

4.4 Detalles de las colecciones

Las categorías de clases `Collections-` son las más prolíficas, hay 7 de ellas conteniendo 46 clases.

La categoría `Collections-Abstract` agrupa clases que se consideran abstractas. Una clase *abstracta* no se puede instanciar, su comportamiento se declara pero no se implementa por completo. Es responsabilidad de sus subclases implementar la parte del comportamiento que falta.

Una clase abstracta es útil para establecer un conjunto de métodos polimórficos que se espera que cada una de sus subclases concretas especialice. Esto captura y comunica nuestra intención.

Observe cómo se declara el importante método `#do:`, pero no se implementa:

```
Collection>>do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."
self subclassResponsibility
```

A continuación, observa cómo lo implementan dos subclases diferentes de `Collection`:

```
OrderedCollection>>do: aBlock
firstIndex to: lastIndex do: [ :index |
   aBlock value: (array at: index) ]
```

and:

```
Dictionary>>do: aBlock
super do: [:assoc | aBlock value: assoc value]
```

Hay que distinguir dos grupos importantes de colecciones: las colecciones con un tamaño fijo y las colecciones con un tamaño variable.

Colección de tamaño fijo. Estas colecciones se agrupan en la categoría `Collections-Arrayed`. La más destacada es `Array`, cuyo tamaño (el número de elementos que puede contener) se establece al crear la instancia. Una vez instanciada, no se pueden añadir ni eliminar elementos de una matriz.

Hay diferentes formas de crear una instancia de `Array`:

```
array1 ← #(2 'Apple' $@ 4) "create at compile time"
array1b ← {2 . 'Apple' . 2@1 . 1/3 } "created a execution time"
array2 ← Array with: 2 with: 'Apple' with: 2@3 with: 1/3.
array3 ← Array ofSize: 4 "an empty array with a 4 element capacity"
```

Ejemplo 4.9: Colección con un tamaño fijo

El array `array1` y el `array1b` son un poco diferentes. El primero se crea y se rellena con su contenido durante la compilación del código, por lo que sólo se puede rellenar con elementos literales como enteros, flotantes o cadenas. El segundo se crea en el momento de la ejecución del código y se puede rellenar con elementos instanciados en ese momento, como instancias de `Fraction` o `Point`.

Puedes acceder a elementos con una gran variedad de mensajes:

```
array1 first ⇒ 2
array1 second ⇒ 'Apple'
array1 third ⇒ $@
array1 fourth ⇒ 4
array1 last ⇒ 4
array1 at: 2 ⇒ 'Apple'
array2 at: 3 ⇒ 2@3
array2 swap: 2 with: 4 ⇒ #(2 1/3 2@3 'Apple')
array1 at: 2 put: 'Orange'; yourself ⇒ #(2 'Orange' $@ 4)
array1 indexOf: 'Orange' ⇒ 2
```

Ejemplo 4.10: Acceso a elementos de colección

Utiliza el *System Browser* para descubrir formas alternativas de acceder a los elementos de una colección.

¿Cuál es el mensaje adecuado para acceder a los dos primeros elementos de la colección `array1`*?*

Ejercicio 4.12: Acceder a parte de una colección

Sin embargo, no se puede añadir ni eliminar ningún elemento:

```
array1 add: 'Orange'
⇒ Error: 'This message is not appropriate for this object'
array1 remove: 'Apple'
⇒  Error: 'This message is not appropriate for this object'
```

Sin embargo, es posible rellenar una matriz de una sola vez:

¿Rellenar todos los elementos de `array1` *con 'kiwi' de una sola vez?*

Ejercicio 4.13: Rellenar un array

Colección de tamaño variable. Dichas colecciones se agrupan en varias categorías de clases: `Collections-Unordered`, `Collections-Sequenceable`, etc. Representan las colecciones más comunes.

Es destacable `OrderedCollection`. Sus elementos están ordenados: los elementos se añaden uno tras otros en secuencia[2]. Su tamaño varía dependiendo de los elementos añadidos o eliminados.

```
coll1 ← {2 . 'Apple' . 2@1 . 1/3 } asOrderedCollection
coll2 ← OrderedCollection with: 2 with: 'Apple' with: 2@1 with: 1/3
coll3 ← OrderedCollection ofSize: 4
```

Ejemplo 4.11: Colección con un tamaño variable

El acceso a los elementos es idéntico al de una instancia de `Array`, pero las colecciones dinámicas te permiten añadir y eliminar elementos:

```
coll1 add: 'Orange'; yourself
⇒ an OrderedCollection(2 'Apple' 2@1 1/3 'Orange')
coll1 remove: 2@1; yourself
⇒ an OrderedCollection(2 'Apple' 1/3)
```

Ejemplo 4.12: Añadir, eliminar elementos de un array dinámico

¿Cómo añadir 'Orange' después de 'Apple' en `coll1`*?*

Ejercicio 4.14: Añadir un elemento después

Set. `Set` es una colección desordenada sin elemenntos duplicados. En cambio, el orden de los elementos no está garantizado. Observa cómo pi es el primer elemento del set:

```
set ← Set new.
set add: 1; add: Float pi; yourself
⇒ a Set(3.141592653589793 1)
```

Ejemplo 4.13: Colección Set

Se garantiza que no habrá duplicados, incluso con varios tipos diferentes. Fíjate cómo `1`, `3/3` y `1.0` se consideran iguales y no se duplican en el conjunto:

```
set ← Set new.
set add: 1; add: Float pi; add: 3/3; add: 1/3; add: 1.0; yourself
⇒ a Set(1/3 3.141592653589793 1)
```

Ejemplo 4.14: Set, sin duplicados

Una forma muy práctica de crear una instancia `Set`, o cualquier otra colección, es crear una matriz dinámica y convertirla con el mensaje `#asSet`:

[2] Por supuesto, pueden insertar un elemento entre otros dos. Sin embargo, una instancia `LinkList` es más efiiente en este caso.

```
{1 . Float pi . 3/3 . 1/3 . 1.0} asSet
⇒ a Set(3.141592653589793 1/3 1)
```

Ejemplo 4.15: Convertir un array dinámico

Fíjate en los mensajes de conversión alternativos:

```
{1 . Float pi . 3/3 . 1/3 . 1.0} asOrderedCollection
⇒ an OrderedCollection(1 3.141592653589793 1 1/3 1.0)

{1 . Float pi . 3/3 . 1/3 . 1.0} asSortedCollection
⇒ a SortedCollection(1/3 1 1 1.0 3.141592653589793)
```

Para recopilar de forma única la lista de divisores de 30 y 45 (no los divisores comunes):

```
Set  new
   addAll: #(1 2 3 5 6 10 15 30) ;
   addAll: #(1 3 5 9 15 45) ;
   yourself.
⇒ a Set(5 10 15 1 6 30 45 2 3 9)
```

¿Cómo recopilarás las letras en las frases 'buenos días' y 'bonjour'?

Ejercicio 4.15: Letras

Dictionary. Un diccionario es una lista de asociaciones de una clave y un objeto. Por supuesto, una clave es un objeto, pero debe responder a pruebas de igualdad. La mayoría de las veces, se utilizan símbolos como claves.

Para agrupar una lista de colores:

```
| colors |
colors ← Dictionary new.
colors
   add: #red -> Color red;
   add: #blue -> Color blue;
   add: #green -> Color green
```

Ejemplo 4.16: Diccionario de colores

Hay descripciones más breves:

```
colors ← Dictionary newFrom:
   {#red -> Color red . #blue -> Color blue . #green -> Color green}.
colors ← {#red -> Color red . #blue -> Color blue .
   #green -> Color green} asDictionary
```

Accedes al color mediante símbolos:

```
colors at: #blue
```

```
⇒ Color blue
colors at: #blue put: Color blue darker
colors at: #yellow ifAbsentPut: Color yellow
⇒ association `#yellow -> Colors yellow` added to the dictionary
```

Hay diferentes formas de acceder al contenido de un diccionario:

```
colors keys.
⇒ #(#red #green #blue)
colors keyAtValue: Color green
⇒ #green
```

Ten cuidado. Los enumeradores clásicos iteran los valores del diccionario:

```
colors do: [:value | Transcript show:  value; space ]
⇒ (Color r: 1.000 g: 1.000 b: 0.078) (Color r: 0.898 g: 0.000 b: 0.000)...
```

A veces, realmente es necesario iterar toda la asociación clave-valor:

```
colors associationsDo: [:assoc |
   Transcript show: assoc key; space; show: assoc value; cr ]
```

Hay otras variantes que puedes explorar por tu cuenta.

Con el apropiado enumerador, ¿cómo puedes modificar los contenidos del diccionario `colors` *para reemplazar sus valores por un bonita cadena capitalizada que represente el nombre del color?*

Ejercicio 4.16: Color por nombre

Hay muchas más colecciones por explorar. Ahora ya sabes lo suficiente como para explorar y buscar por ti mismo con el navegador del sistema, y para experimentar con el espacio de trabajo.

4.5 Colecciones de SpaceWar!

4.5.1 Instanciar colecciones

Siempre que tengas que manejar más de un elemento de la misma naturaleza (instancias de la misma clase), es recomendable utilizar una colección para almacenarlos. Además, cuando estos elementos son de cantidad fija, es más preciso utilizar una instancia `Array`. Un `Array` es una colección de tamaño fijo. No puede crecer ni reducirse.

Cuando esta cantidad es variable, se recomienda utilizar una instancia de `OrderedCollection`. Se trata de una colección de tamaño variable, que puede crecer o reducirse.

SpaceWar! es un juego para dos jugadores, siempre habrá dos jugadores y dos naves espaciales. Utilizamos una instancia `Array` para mantener una referencia a cada nave espacial.

Cada jugador puede disparar varios torpedos; por lo tanto, el juego admite cero o más torpedos, cientos si así lo decidimos. La cantidad de torpedos es variable, queremos usar una instancia `OrdredCollection` para llevar un registro de ellos.

En la clase `SpaceWar`, ya hemos definido dos variables de instancia: `ships` y `torpedoes`. Ahora queremos un método `initializeActors` para configurar el juego con los actores involucrados: estrella central, naves, etc. Parte de esta inicialización consiste en crear las colecciones necesarias.

A continuación se muestra una implementación incompleta de este método:

```
SpaceWar>>initializeActors
   centralStar ← CentralStar new.
   ../..
   ships first
      position: 200 @ -200;
      color: Color green.
   ships second
      position: -200 @ 200;
      color: Color red
```

Ejemplo 4.17: Incomplete game initialization

El ejemplo anterior no muestra la creación de las colecciones `ships` *y* `torpedoes`*. Reemplaza «../..» por líneas de código en las que se instancien estas colecciones y, si es necesario, se rellenen.*

Ejercicio 4.17: Colecciones para contener las naves y torpedos

4.5.2 Colecciones en acción

La nave espacial y los objetos torpedo son responsables de sus estados internos. Entienden el mensaje `#update:` para recalcular su posición de acuerdo con las leyes mecánicas.

Un torpedo disparado tiene una velocidad constante, no se le aplican fuerzas externas. Su posición se actualiza linealmente según el tiempo transcurrido. El parámetro `t` en el mensaje `#update:` es este intervalo de tiempo.

```
Torpedo>>update: t
"Update the torpedo position"
   position ← velocity * t + position.
   ../..
```

Ejemplo 4.18: Mecánicas del torpedo

Una nave espacial está sometida a la fuerza gravitatoria de la estrella y a la aceleración de sus motores. Por lo tanto, su velocidad y posición cambian según las leyes mecánicas de la física.

```
SpaceShip>>update: t
"Update the ship position and velocity"
   | ai ag newVelocity |
   "acceleration vectors"
   ai ← acceleration * self direction.
   ag ← self gravity.
   newVelocity ← (ai + ag) * t + velocity.
   position ← (0.5 * (ai + ag) * t squared) + (velocity * t) + position.
   velocity ← newVelocity.
   ../..
```

Ejemplo 4.19: Mecánicas de la nave espacial

Recuerda que Smalltalk no sigue la precedencia matemática de los operadores aritméticos. Estos se consideran **mensajes binarios** normales que se evalúan de izquierda a derecha cuando no hay paréntesis. Por ejemplo, en el fragmento de código `...(velocity * t)...`, los paréntesis son obligatorios para obtener el cálculo esperado.

Observa que en ese método anterior cómo la dirección y la gravedad están definidas en dos métodos específicos.

El mensaje `#direction` solicita el vector unitario que representa la dirección de la proa de la nave espacial:

```
SpaceShip>>direction
"I am a unit vector representing the nose direction of the mobile"
   ↑ Point rho: 1 theta: self heading
```

Ejemplo 4.20: Método dirección de la nave espacial

El mensaje `#gravity` solicita el vector de gravedad al que está sometida la nave espacial:

```
SpaceShip>>gravity
"Compute the gravity acceleration vector"
   | position |
   position ← self morphPosition.
   ↑ [-10 * self mass * self starMass / (position r raisedTo: 3) * position]
      on: Error do: [0 @ 0]
```

Ejemplo 4.21: Gravedad de la nave espacial

Observa el mensaje `#starMass` enviado a la propia nave espacial. Nosotros, como nave espacial, aún no hemos descubierto cómo preguntarle a la estrella central cuál es su masa estelar. Nuestro método `starMass` solo puede devolver, por ahora, el número 8000.

El juego es responsabilidad de una instancia de `SpaceWar`. A intervalos regulares, actualiza los estados de los actores del juego. Se llama al método `#stepAt:` a intervalos regulares determinados por el método `#stepTime`:

```
SpaceWar>>stepTime
"millisecond"
   ↑ 20

SpaceWar>>stepAt: millisecondSinceLast
   ../..
   ships do: [:each | each unpush].
   ../..
```

Ejemplo 4.22: Actualización periódica del juego

En el método `stepAt:`, hemos omitido intencionadamente los detalles para actualizar las posiciones de la nave y el torpedo. Nota: cada nave recibe periódicamente un mensaje `#unpush` para restablecer su aceleración `#push` anterior.

Reemplaza las dos líneas «../..» con código para actualizar las posiciones y velocidades de las naves y los torpedos.

Ejercicio 4.18: Actualizar todas las naves y torpedos

Entre otras cosas, el juego gestiona las colisiones entre los distintos protagonistas. Los enumeradores son muy útiles para esto.

Los barcos se guardan en una matriz de tamaño 2, simplemente la iteramos con un mensaje `#do:` y un bloque de código dedicado:

```
SpaceWar>>collisionsShipsStar
   ships do: [:aShip |
      (aShip morphPosition dist: centralStar morphPosition) < 20 ifTrue: [
         aShip flashWith: Color red.
         self teleport: aShip]
   ]
```

Ejemplo 4.23: Colisión entre las naves y el Sol

5 Mensajes de control de flujo

> Los necios ignoran la complejidad. Los pragmáticos la soportan. Algunos pueden evitarla. Los genios la eliminan.
>
> —*Alan Perlis*

La sintaxis de Cuis-Smalltalk es mínima. Básicamente, solo hay sintaxis para enviar mensajes (es decir, expresiones). Las expresiones se construyen a partir de un número muy reducido de elementos primitivos. Solo hay 6 palabras clave y **no hay sintaxis para estructuras de control** ni para declarar 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`. Como ya sabemos, las nuevas (sub)clases se crean enviando un mensaje a su superclase.

5.1 Elementos sintácticos

Las expresiones se componen de los siguientes bloques de construcción:

1. seis palabras reservada, o *pseudovariables*: `self`, `super`, `nil`, `true`, `false`, y `thisContext`,
2. expresiones constantes para *objetos literales*, incluyendo números, caracteres, cadenas, símbolos y matrices,
3. declaraciones de variables
4. asignaciones,
5. *block closures*,
6. mensajes.

5.2 Pseudovariables

En Smalltalk hay 6 palabras clave reservadas, o pseudovariables:

> `nil`, `true`, `false`, `self`, `super`, y `thisContext`.

Se denominan *pseudovariables* porque están predefinidas y no se les puede asignar ningún valor. `true`, `false` y `nil` son constantes, mientras que los valores de `self`, `super` y `thisContext` varían dinámicamente a medida que se ejecuta el código.

- `true` y `false` son las instancias únicas de las clases `True` y `False` de las clases `Boolean`.
- `self` se refiere siempre al receptor del método que se está ejecutando.
- `super` también se refiere al receptor del método actual, pero se le envía el mensaje a `super`, La búsqueda de métodos cambia de modo que comienza desde la superclase de la clase que contiene el método que utiliza `super`.
- `nil` es el objeto indefinido. Es la única instancia de la clase `UndefinedObject`. Las variables de instancia, las variables de clase y las variables locales se inicializan a `nil`.
- `thisContext` es una pseudovariable que representa el marco superior de la pila de tiempo de ejecución. En otras palabras, representa el `MethodContext` o `BlockContext` que se está ejecutando actualmente. Normalmente, `thisContext` no es de interés para la mayoría de los programadores, pero es esencial para implementar herramientas de desarrollo como el depurador y también se utiliza para implementar el manejo de excepciones y continuaciones.

5.3 Sintaxis de método

Mientras que las expresiones pueden evaluarse en cualquier lugar de Cuis-Smalltalk (por ejemplo, en un *workspace*, en un *debugger* o en un *browser*), los métodos se definen normalmente en la ventana System Browser o en el Debugger. Los métodos también pueden archivarse desde un medio externo, pero esta no es la forma habitual de programar en Cuis-Smalltalk.

Los programas se desarrollan método a método, en el contexto de una clase determinada. Una clase se define enviando un mensaje a una clase existente, pidiéndole que cree una subclase, por lo que no se requiere ninguna sintaxis especial para definir clases. Ya estamos familiarizados con esto gracias a los ejemplos anteriores.

Echemos otro vistazo a la sintaxis del método cuando interviene el flujo de control; nuestra primera explicación fue Sección 3.6 [Spacewar! Estados y comportamientos], página 40).

He aquí el método `keyStroke:` en la clase `SpaceWar`.

```
SpaceWar>>keyStroke: event
"Check for any keyboard stroke, and take action accordingly"
   | key |
   key ← event keyCharacter.
   event isArrowUp ifTrue: [↑ ships first push].
   event isArrowRight ifTrue: [↑ ships first right].
   event isArrowLeft ifTrue: [↑ ships first left].
   event isArrowDown ifTrue: [↑ ships first fireTorpedo].
   key = $w ifTrue: [↑ ships second push].
   ...
```

Ejemplo 5.1: SpaceWar! pulsación de tecla (*key stroke*)

Sintácticamente, un método consta de:

- el patrón del método, que contiene el nombre (es decir, `keyStroke:`) y cualquier argumento. Aquí, `event` es un `KeyboardEvent`,
- comentarios (pueden aparecer en cualquier lugar, pero lo habitual es colocar uno al principio que explique lo que hace el método),
- declaración de variables locales (en el ejemplo, `key`),
- y cualquier cantidad de expresiones separadas por puntos; ahí hay 5.

La evaluación de cualquier expresión precedida por un ↑ (escrito como `^`) hará que el método salga en ese punto, devolviendo el valor de esa expresión. Un método que termina sin devolver explícitamente el valor de alguna expresión siempre devolverá el valor actual de `self`.

Los argumentos y las variables locales siempre deben comenzar con letras minúsculas. Los nombres que comienzan con letras mayúsculas se consideran variables globales. Los nombres de clases, como `Character`, por ejemplo, son simplemente variables globales que hacen referencia al objeto que representa esa clase.

Como se puede deducir del Ejemplo 2.2, `Smalltalk allClasses size` simplemente envía el mensaje `#allClasses` a un diccionario llamado `Smalltalk`. Al igual que con cualquier otro objeto, se puede inspeccionar este diccionario. Aquí se puede observar un caso de autorreferencia: el valor de `Smalltalk at: #Smalltalk` es `Smalltalk`.

5.4 Sintaxis de bloque

Los bloques proporcionan un mecanismo para aplazar la evaluación de expresiones. Un bloque es esencialmente una función anónima. Un bloque se evalúa enviándole el mensaje `#value`. El bloque responde con el valor de la última expresión de su cuerpo, a menos que haya un retorno explícito (con ↑), en cuyo caso devuelve el valor de la expresión siguiente.

```
[ 1 + 2 ] value
⇒ 3
```

Los bloques pueden tomar parámetros, cada uno de los cuales se declara con dos puntos al principio. Una barra vertical separa la declaración de los parámetros del cuerpo del bloque. Para evaluar un bloque con un parámetro, debe enviarle el mensaje `#value:` con un argumento. A un bloque de dos parámetros se le debe enviar `#value:value:`, y así sucesivamente, hasta un máximo de 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 una matriz. (Un bloque con un gran número de parámetros suele ser señal de un problema de diseño).

Los bloques también pueden declarar variables locales, que están rodeadas por barras verticales, al igual que las declaraciones de variables locales en un método. Las variables locales se declaran después de cualquier argumento:

```
[ :x :y | | z | z ← x + y. z ] value: 1 value: 2
⇒ 3
```

Los bloques pueden hacer referencia a variables del entorno circundante. Se dice que los bloques «cierran» su entorno léxico, lo cual es una forma elegante de decir que recuerdan y hacen referencia a variables en su contexto léxico circundante, es decir, aquellas que aparecen en el texto que los rodea.

El siguiente bloque hace referencia a la variable x de su entorno circundante:

```
|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 se pueden asignar a variables y pasar como argumentos al igual que cualquier otro objeto.

Considera el siguiente ejemplo para calcular los divisores de un número entero:

```
| n m |
n ← 60.
m ← 45.
(1 to: n) select: [:d | n \\ d = 0 ].
"⇒ #(1 2 3 4 5 6 10 12 15 20 30 60)"
(1 to: m) select: [:d | m \\ d = 0]
"⇒ #(1 3 5 9 15 45)"
```

Ejemplo 5.2: Calcular divisores

El problema con este ejemplo es la duplicación del código en el cálculo del divisor. Podemos evitar la duplicación con un bloque dedicado que realice el cálculo y lo asigne a una variable:

Cómo reescribirías el Ejemplo 5.2 para evitar la duplicación de código?

Ejercicio 5.1: Calcular divisores con un bloque

El método `SpaceWar>>teleport:` contiene un buen ejemplo del uso de un bloque para evitar la duplicación de código al generar coordenadas aleatorias de abscisa y ordenada. Cada vez que se necesita una nueva coordenada, se envía el mensaje `#value` al bloque de código:

```
SpaceWar>>teleport: aShip
  "Teleport a ship at a random location"
  | area randomCoordinate |
  aShip resupply.
  area ← self morphLocalBounds insetBy: 20.
  randomCoordinate ← [(area left to: area right) atRandom].
  aShip
    velocity: 0 @ 0;
    morphPosition: randomCoordinate value @ randomCoordinate value
```

Ejemplo 5.3: Método `teleport:`

5.5 Control de flujo con bloques y mensajes

La decisión de enviar *este* mensaje en lugar de *aquél* se denomina *control de flujo*, es decir, controlar el flujo de un cálculo. Smalltalk no ofrece construcciones especiales para el control de flujo. La lógica de decisión se expresa enviando mensajes a booleanos, números y colecciones con bloques como argumentos.

Condicional

Las condiciones se expresan enviando uno de los mensajes `#ifTrue:`, `#ifFalse:` o `#ifTrue:ifFalse:` al resultado de una expresión booleana:

```
(17 * 13 > 220)
  ifTrue: [ 'bigger' ]
  ifFalse: [ 'smaller' ]
⇒ 'bigger'
```

La clase `Boolean` ofrece una visión fascinante de cuánto del lenguaje Smalltalk se ha incorporado a la biblioteca de clases. `Boolean` es la superclase abstracta de las clases *Singleton* `True` y `False`[1].

La mayor parte del comportamiento de las instancias `Boolean` puede entenderse considerando el método `ifTrue:ifFalse:`, que toma dos bloques como argumentos:

```
(4 factorial > 20) ifTrue: [ 'bigger' ] ifFalse: [ 'smaller' ]
⇒ 'bigger'
```

El método es abstracto en `Boolean`. Se implementa en sus subclases concretas `True` y `False`:

```
True>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ↑ trueAlternativeBlock value

False>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ↑ falseAlternativeBlock value
```

Ejemplo 5.4: Implementaciones de ifTrue:ifFalse:

De hecho, esta es la esencia de la POO: cuando se envía un mensaje a un objeto, el propio objeto determina qué método se utilizará 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 `Boolean` se implementan de esta manera para `True` y `False`. Veamos otro ejemplo:

```
True>>not
   "Negation----answer false since the receiver is true."
   ↑ false
```

Ejemplo 5.5: Implementación de la negación

Los booleanos ofrecen varios métodos útiles, como `ifTrue:`, `ifFalse:`, `ifFalse:ifTrue:`. También puedes elegir entre conjunciones y disyunciones rápidas y 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, se evalúan ambas subexpresiones `Boolean`, ya que `&` toma un argumento `Boolean`. En el segundo y tercer ejemplo, solo se evalúa la primera, ya que `and:` espera un `Block` como argumento. El `Block` solo se evalúa si el primer argumento es verdadero.

[1] Una clase *singleton* está destinada a tener sólo una instancia. Cada una de las clases `True` y `False` tiene sólo una instancia, los valores `true` y `false`.

Intenta imaginar cómo están implementados `and:` *y* `or:`.

Ejercicio 5.2: Implementar `and:` y `or:`

En el Ejemplo 5.1 al principio de este capítulo, hay 4 mensajes de flujo de control `#ifTrue:`. Cada argumento es un bloque de código y, cuando se evalúa, devuelve explícitamente una expresión, interrumpiendo así la ejecución del método.

En el fragmento de código del Ejemplo 5.6 a continuación, comprobamos si una nave se ha perdido en el espacio profundo. Depende de dos condiciones:

1. la nave está fuera del área de juego, probado con el mensaje `#isInOuterSpace`,
2. la nave toma la dirección del espacio profundo, probada con el mensaje `#isGoingOuterSpace`.

Por supuesto, la condición n.º 2 solo se comprueba cuando la condición n.º 1 es verdadera.

```
"Are we out of screen?
If so we move the mobile to the other corner
and slow it down by a factor of 2"
(self isInOuterSpace and: [self isGoingOuterSpace])
  ifTrue: [
     velocity ← velocity / 2.
     self morphPosition: self morphPosition negated]
```

Ejemplo 5.6: Nave perdida en el espacio

Bucle

Los bucles suelen expresarse enviando mensajes a bloques, enteros o colecciones. Dado que la condición de salida de un bucle puede evaluarse repetidamente, debe ser un bloque en lugar de un valor booleano. A continuación se muestra 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
```

Puedes consultar todas las alternativas en la categoría de métodos de `control` de la clase `BlockClosure`.

`#timesRepeat:` ofrece una manera sencilla de implementar una iteración:

```
n ← 1.
10 timesRepeat: [ n ← n * 2 ].
```

```
n ⇒ 1024
```

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

Puedes consultar todas las alternativas en la categoría del método `intervals` de la clase `Number`.

Si la condición de salida de un método como `whileTrue:` nunca se cumple, es posible que hayas implementado un *bucle infinito*. Solo tienes que pulsar *`Cmd-punto`* para acceder al depurador.

5.6 Métodos de Spacewar!

Ya estás familiarizado con la escritura de métodos simples para el juego Spacewar! Escribiremos algunos más y aprenderemos a clasificarlos.

5.6.1 Inicializar el juego

Queremos añadir el método `initialize` a nuestra clase `SpaceWar`. Por supuesto, necesitamos utilizar el *System Browser*: ...menú World → `Open...` → `Browser`...

Como recordatorio, haz lo siguiente (si es necesario mira la Figura 2.1):

1. En el panel de `Categoría de clases` a la izquierda, desplázate hacia abajo hasta la categoría `Spacewar!` y selecciónala.
2. En el panel de `Clases`, selecciona la clase `SpaceWar`.
3. Debajo, haz click en el botón de `instance` para exponer los métodos del lado de la instancia de la clase `SpaceWar`. Es el comportamiento por defecto del *browser* de todos modos, por lo que puedes saltarte este paso siempre que no hayas pulsado el botón `class`.
4. En el panel `Categoría de Métodos`, selecciona la categoría `-- all --`. Se mostrará en el panel inferior una plantilla de código para el método:

   ```
   messageSelectorAndArgumentNames
     "comment stating purpose of message"
     | temporary variable names |
     statements
   ```

   La plantilla contiene cuatro líneas: el nombre del métdo, un comentario, una declaración de variable local y *statements*. Puedes seleccionar todo y borrarlo o editar línea a línea de la plantilla según necesites.

   En nuestro caso, seleccionaremos todo y lo reemplazaremos con el código fuente de `SpaceWar>>initialize`:

```
SpaceWar>>initialize
  "We want to capture keyboard and mouse events,
  start the game loop(step) and initialize the actors."
  super initialize.
  color ← self defaultColor.
  self setProperty: #'handlesKeyboard' toValue: true.
  self setProperty: #'handlesMouseOver:' toValue: true.
  self startSteppingStepTime: self stepTime.
  self initializeActors
```

Ejemplo 5.7: Inicializar `SpaceWar`

5. Una vez cambiado, guárdalo con ***`Ctrl-s`*** o ...click derecho → `Accept (s)`...

El método recién creado se mostrará en el panel `Métodos`. También puedes clasificarlo automáticamente: con el ratón, ve al panel `Categorías de métodos`...haz clic con el botón derecho → `categorize all uncategorized (c)`... (clasificar todo lo no clasificado).

En la clase `SpaceWar`*, añade el método* `teleport:` *tal y como se define en Ejemplo 5.3 y, a continuación, clasifícalo en la categoría de métodos* `events`*.*

Ejercicio 5.3: Clasificar un método

5.6.2 Controles de la nave espacial

En un capítulo anterior, escribiste como ejercicio una implementación sencilla de los [métodos de control de la nave], página 44. Los métodos de control definitivos de la clase `SpaceShip` se reescriben como:

```
SpaceShip>>push
  "Init an acceleration boost"
  fuel isZero ifTrue: [↑ self].
  fuel ← fuel - 1.
  acceleration ← 50

SpaceShip>>unpush
  "Stop the acceleration boost"
  acceleration ← 0

SpaceShip>>right
  "Rotate the ship to its right"
  self heading: self heading + 0.1

SpaceShip>>left
  "Rotate the ship to its left"
  self heading: self heading - 0.1
```

Ejemplo 5.8: Controles de la nave

Observa los métodos `#right` y `#left`, son prácticamente idénticos a los que se piden en el Ejercicio 3.12. No modificamos directamente el atributo `heading`, sino que utilizamos los métodos `heading:` y `heading` para leer y escribir esta información.

Clasifica los métodos de control en una nueva categoría creada con el nombre de `control`.

Ejercicio 5.4: Clasificar los métodos de control

El control no estará completo sin el método para disparar un torpedo. Inicializar correctamente un torpedo es más complejo. Esto se debe a que una nave espacial suele estar en movimiento y, además, su rumbo y velocidad cambian con frecuencia. Por lo tanto, el torpedo debe configurarse de acuerdo con la posición, el rumbo y la velocidad actuales de la nave espacial antes de ser disparado.

```
SpaceShip>>fireTorpedo
  "Fire a torpedo in the direction of
  the ship heading with its velocity"
  | torpedo |
  torpedoes isZero ifTrue: [ ↑ self].
  torpedoes ← torpedoes - 1.
  torpedo ← Torpedo new.
  torpedo
    position: position + self nose;
    heading: self heading;
    velocity: velocity;
    color: self color muchLighter.
  owner addTorpedo: torpedo
```

Ejemplo 5.9: Disparar un torpedo desde una nave espacial en movimiento

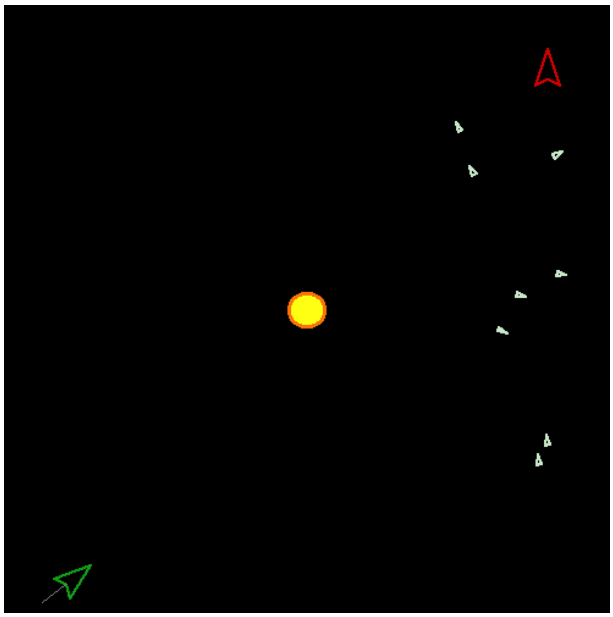

Figura 5.1: Spacewar! torpedos alrededor

5.6.3 Colisiones

En un capítulo anterior, ofrecimos una pequeña muestra del código de detección de colisiones entre las naves espaciales y la estrella central. Se basa en un iterador, un bloque de código y un flujo de control.

Sin embargo, existen otros escenarios, como colisiones entre naves, entre torpedos y el Sol, y entre torpedos y naves.

¿Cómo escribirás el método para detectar la colisión entre las dos naves y tomar las acciones correspondiente? (Adáptalo de Ejemplo 4.23).

Ejercicio 5.5: Colisión de naves

La detección entre las dos naves y la gran cantidad posible de torpedos necesitó dos enumeradores con bloques de código anidados:

```
SpaceWar>>collisionsShipsTorpedoes
ships do: [:aShip |
  torpedoes do: [:aTorpedo |
    (aShip morphPosition dist: aTorpedo morphPosition) < 15 ifTrue: [
      aShip flashWith: Color red.
      aTorpedo flashWith: Color orange.
      self destroyTorpedo: aTorpedo.
      self teleport: aShip]
  ]
]
```

Ejemplo 5.10: Colisión entre las naves y los torpedos

El último escenario de colisión entre torpedo y sol lo dejamos como un ejercicio para ti.

Escribe el método para detectar las colisiones entre los torpedos y la estrella central y realizar la acción correspondiente. (Adáptalo desde Ejemplo 4.23 y Ejemplo 5.10.)

Ejercicio 5.6: Colisiones entre los torpedos y el Sol

6 Visual con Morph

> Morphic es un framework de interfaz de usuario que facilita y hace más divertido crear interfaces interactivas y dinámicas.
>
> —*John Maloney*

¿Qué esperaríamos si solicitáramos un buen soporte para crear interfaces gráficas de usuario en un sistema de programación?

Todos los ordenadores modernos (y teléfonos, etc.) tienen pantallas a color de alta resolución. Cualquier software que se ejecute en ellos y que sea accesible para un usuario debe mostrar información en esa pantalla.

Los gestores de interfaz de usuario convencionales (es decir, los sistemas operativos y los navegadores web) comenzaron incluyendo solo los elementos más básicos de la interfaz gráfica de usuario: editores de texto básicos, botones, listas simples, desplazamiento para contenidos de gran tamaño y (por lo general) múltiples ventanas superpuestas y redimensionables. Cualquier otra cosa debe gestionarse mediante bibliotecas adicionales. Aunque existen bibliotecas para gestionar contenidos más ricos (D3.js y Matplotlib son algunos ejemplos), el resultado no es coherente, ni para los desarrolladores ni para los usuarios.

Cuis-Smalltalk adopta un enfoque diferente, iniciado por Smalltalk-80 y, especialmente, por Self. Entraremos en detalles en el siguiente capítulo, *The Fundamentals of Morph* (Los fundamentos de Morph). Por ahora, ocupémonos directamente de Morphs.

Damos por sentada la alta calidad de la pantalla, así como la del ratón, el dedo u otros dispositivos señaladores. Y nos basamos en el objetivo de proporcionar amplias posibilidades para las interfaces gráficas de usuario, tanto en estilos y diseños existentes como en otros novedosos aún por inventar. Además, al estilo habitual de Smalltalk, todo el código del *framework* está disponible para su estudio y modificación. No hay bibliotecas de terceros. Solo el código de nivel más bajo está precompilado, pero aún así se puede sobrescribir o modificar.

Por lo tanto, cada objeto que ves en Cuis-Smalltalk es un `Morph` o está compuesto por `Morph`s. Básicamente, un `Morph` es un objeto con estado y comportamiento que también puede representarse a sí mismo en la pantalla de un ordenador.

Puesto que los Morphs son útiles, cuando observes la clase `Morph` en un navegador jerárquico, verás una gran cantidad de métodos y muchas, muchas subclases. Pero las ideas básicas son bastante sencillas.

6.1 Instalar un paquete

Este capítulo requiere que instales el paquete `UI-Shapes.pck.st`, que forma parte del proyecto Cuis-Smalltalk-UI. Para descargar este paquete de Internet, tienes dos opciones:

1. Descarga el archivo `UI-Shapes.pck.st` y guárdalo en la misma carpeta que tu archivo de imagen Cuis-Smalltalk.
2. En la carpeta de imágenes Cuis-Smalltalk, clona su repositorio Git:

   ```
   git clone git@github.com:Cuis-Smalltalk/Cuis-Smalltalk-UI.git
   ```

   Además de los paquetes `UI-Shapes.pck.st`, hay otros que también puedes descubrir en este repositorio.

Una vez hecho esto, en un Workspace, ejecuta el código para instalar el paquete:

```
Feature require: 'UI-Shapes'
```

Tu entorno Cuis-Smalltalk ya está equipado para las siguientes secciones de este capítulo.

6.2 Morph Ellipse

Comencemos con uno de los morphs básicos, un `EllipseMorph`. Podrías escribir `EllipseMorph new openInWorld` y **Do-it**, pero por ahora nos estamos centrando en aspectos visuales, así que vamos a abrir el menú World, seleccionamos el submenú `New Morph... Basic` y lo arrastramos al escritorio.

Cada vez que se obtiene una transformación desde el submenú `New Morph...`, se obtiene una transformación diferente, pero realizada con un estilo estándar.

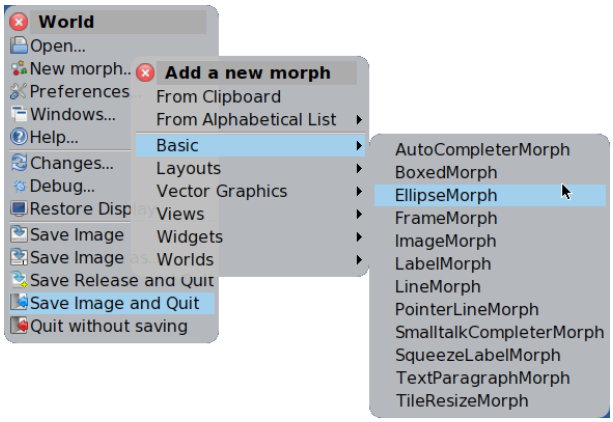

Figura 6.1: Selecciona un `EllipseMorph` desde el menú

El reto básico del *diseño de la interfaz de usuario* es comunicar visibilidad y control. ¿Dónde estoy? ¿Qué puedo hacer aquí?

Uno de los puntos clave del diseño es cómo eliminar el desorden. Una estrategia útil consiste en revelar las capacidades en contexto a medida que se necesitan.

En el caso de Cuis-Smalltalk, hay que conocer algunos conceptos básicos, ya que las herramientas útiles están ahí, pero no interfieren. En cualquier momento, puedes hacer *`clic con el botón derecho`* del ratón en el escritorio para acceder al menú World. También puedes hacer *`clic con el botón central`* del ratón en cualquier `Morph` para obtener un *halo* de *controles de construcción*, que aparecen como pequeños iconos circulares de colores. Si detienes el cursor sobre uno de ellos, aparecerá una *sugerencia*, un texto emergente temporal cuyo nombre te dará una pista sobre su uso.

Si haces clic en otro lugar, las manijas de construcción desaparecen, pero puedes recuperarlas en cualquier momento con un clic del ratón.

Figura 6.2: Arrastra el controlador de construcción para cambiar el tamaño

Ahora que ya lo sabes, mueve el controlador amarillo inferior derecho con la sugerencia `Change size` (Cambiar tamaño) haciendo clic y arrastrando. Solo tienes que mantener pulsado el botón izquierdo del ratón mientras el cursor está sobre el controlador, mover el cursor hacia la derecha y hacia abajo, y soltar el botón del ratón (*click-drag*).

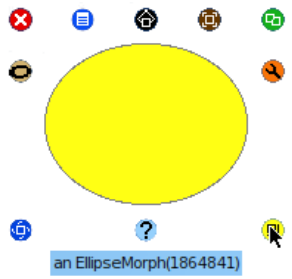

Figura 6.3: Una elipse más grande

6.3 Submorph

Los morphs pueden contener otros morphs. Estos morphs interiores se denominan *submorphs* del morph que los contiene. Una vez más, esto se puede hacer escribiendo el «código» del software, pero vamos a hacerlo directamente con un `BoxedMorph`.

Primero obtenemos un `BoxedMorph` desde los submenús `New morph...`. La instancia `BoxedMorph` se muestra como un rectángulo con un borde.

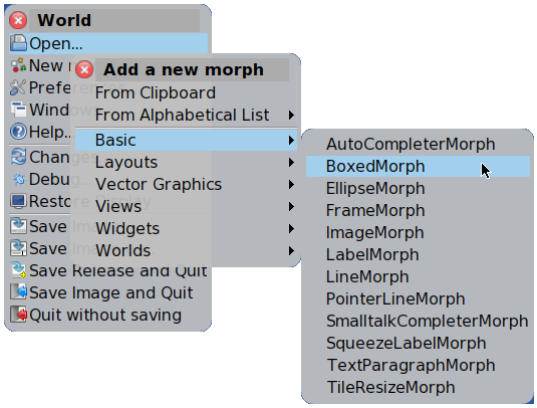

Figura 6.4: Obtener un `BoxedMorph`

Ahora arrastra el rectángulo sobre la elipse y haz *`clic-central`* del ratón sobre el rectángulo y haz clic en el controlador de construcción azul para acceder al *menú Morph* del rectángulo. Utiliza la opción del menú `embed into` (incrustar en) y selecciona la elipse como su nuevo elemento principal.

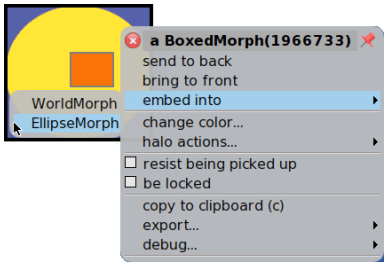

Figura 6.5: Hacer el rectángulo submorph de la elipse

Ahora, cuando hagas clic y arrastres la elipse, o utilices los controles de construcción `Pick up` o `Move`, el rectángulo será solo una decoración para la elipse.

De hecho, el rectángulo parece haberse fusionado con la elipse. Al utilizar el ratón donde se muestra el rectángulo, se está utilizando el ratón sobre la elipse. Este rectángulo no tiene muchos comportamientos interesantes.

Añadamos un comportamiento solo a este `BoxedMorph`.

6.4 Una breve introducción a los inspectores

Para obtener el halo de construcción de una morfología interior, simplemente haz *`clic-central`* varias veces para «profundizar» en la jerarquía de submorphs.

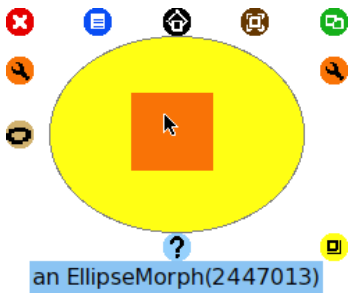

Figura 6.6: Clic-central para controles de construcción

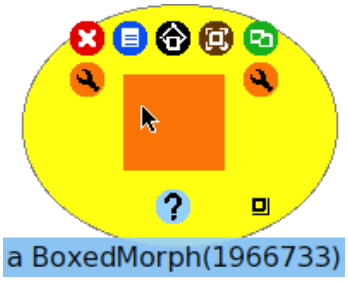

Figura 6.7: Click-central de nuevo para descender en los submorphs

Hay un control naranja a la derecha, justo debajo del control verde `Duplicate`. Haga *`clic-izquierdo`* del ratón para acceder al menú `Debug`. Utilice este menú para obtener un *inspector* para el rectángulo.

Observa la Figura 6.8, a la izquierda tenemos un panel para self, todas las variables de instancia y las variables de instancia individuales. Al hacer clic para seleccionar «all inst vars» y el panel de valores de la derecha, se muestra que el propietario del rectángulo es la elipse y que por ahora no tiene submorphs.

El panel inferior es un editor de código Smalltalk, básicamente un espacio de trabajo, donde `self` está vinculado al objeto que estamos inspeccionando.

Por cierto, los inspectores trabajan con todos los objetos, no solo con los morphs.

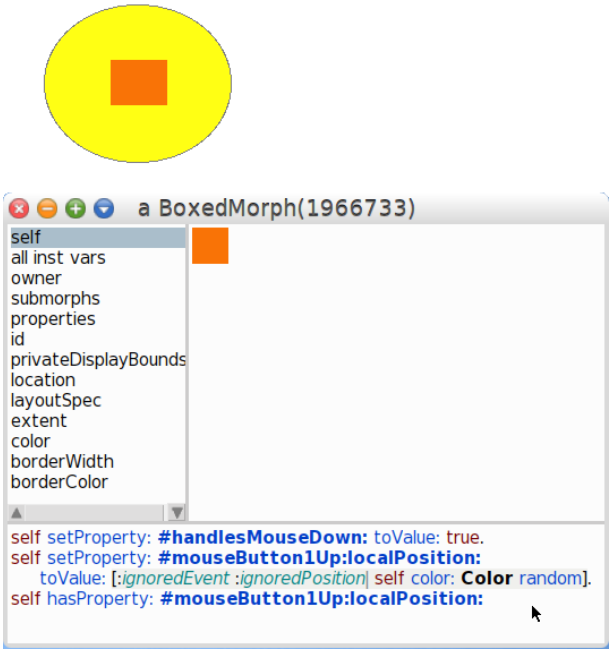

Figura 6.8: Añadir comportamiento específico de la instancia

Para añadir un comportamiento a todas las instancias de una clase, creamos un método de instancia. Aquí vamos a crear un comportamiento para «solo esta instancia `BoxedMorph`».

Además de las variables de instancia, un morph puede tener cualquier número de *propiedades* con nombre que pueden ser diferentes para cada morph.

Añadimos aquí dos propiedades:

```
self setProperty: #handlesMouseDown: toValue: true.
self setProperty: #mouseButton1Up:localPosition:
   toValue: [:ignoredEvent :ignoredPosition| self color: Color random]
```

Ejemplo 6.1: Modificar el comportamiento de este morph desde su Inspector

Estas propiedades son específicas de la interfaz de usuario. Puede encontrar métodos con estos nombres en la clase `Morph` para ver qué hacen.

Después de seleccionar el texto y **`Do-it`**, cada vez que hagas ***`clic-izquierdo`*** del ratón sobre el rectángulo, ¡cambiará de color!

Ten en cuenta que ya no puedes mover la elipse haciendo clic con el ratón sobre el rectángulo, porque ahora el rectángulo capta el evento del ratón. Tienes que hacer clic con el ratón sobre la elipse. Más información al respecto a continuación.

Una nota rápida sobre `Move` frente a `Pick up`. `Move` mueve un submorph «dentro» de su padre. `Pick up` saca un morph «fuera» de su padre.

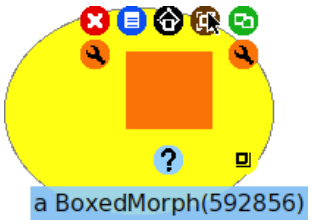

Figura 6.9: Mover un submorph dentro de su padre

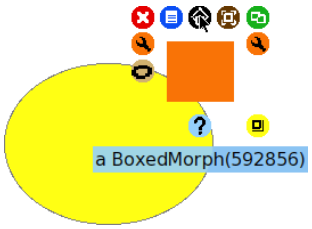

Figura 6.10: Sacar un submorph de su padre

Antes de continuar, utilicemos un inspector en la elipse para cambiar los valores de un par de sus variables de instancia.

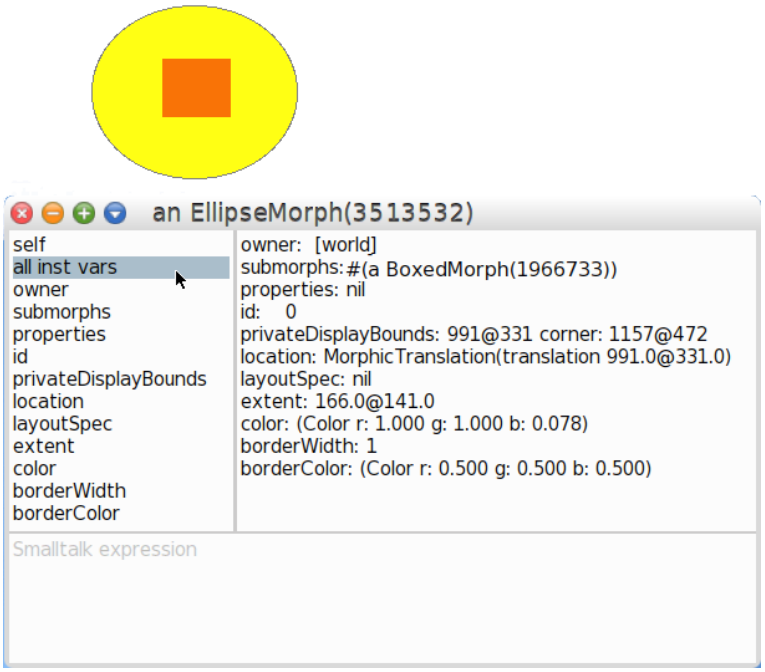

Figura 6.11: Inspeccionar variables de instancia de la elipse

Observa Figura 6.12 y Ejemplo 6.2. En el panel inferior del inspector, se puede ejecutar código en el contexto del objeto inspeccionado. `self` hace referencia a la instancia. Aquí, el panel contiene código para establecer el borderWidth y el borderColor.

```
self borderWidth: 10.
self borderColor: Color blue
```

Ejemplo 6.2: Modificar el estado de esta elipse desde su Inspector

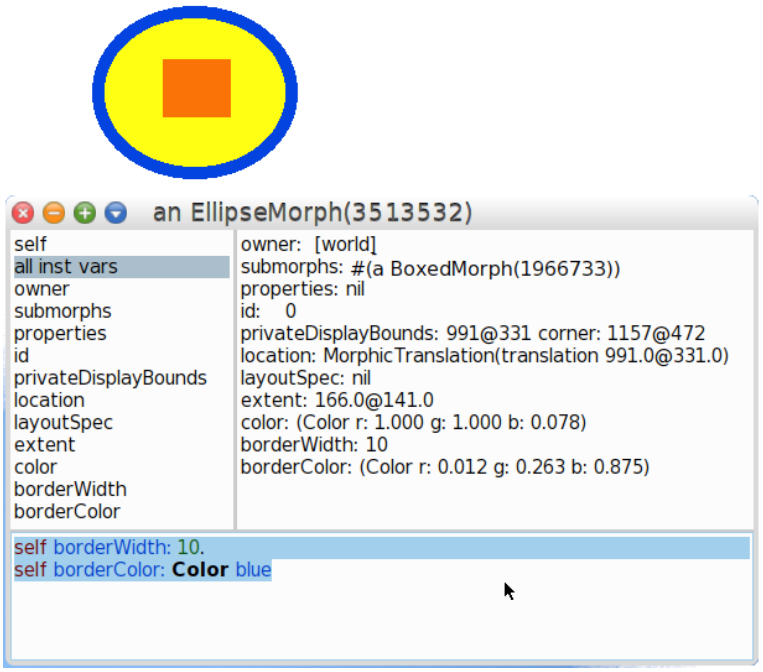

Figura 6.12: Utilizar el Inspector para establecer el color y el ancho del borde

En el caso típico, se desea refinar o cambiar los comportamientos de todas las instancias de una clase.

6.5 Construyendo tu Morph especializado

Creemos una subclase sencilla que cambie de color al hacer *clic-izquierdo*. Crea una nueva clase igual que hicimos con Spacewar!, pero como subclase de `EllipseMorph` con `#ColorClickEllipse`.

```
EllipseMorph subclass: #ColorClickEllipse
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Guarda la definición de la clase con *Ctrl-s*.

Haz *clic-derecho* del ratón en el panel `Message Category` (Categoría de mensajes) y selecciona `new category...` (nueva categoría). Aparecerán varias opciones y podremos crear otras nuevas. Selecciona «event handling testing» (prueba de gestión de eventos). A continuación, añada el método `ColorClickEllipse>>handlesMouseDown:`.

```
handlesMouseDown: aMouseButtonEvent
  "Answer that I do handle mouseDown events"
  ↑ true
```

Del mismo modo, añade una nueva categoría «event handing» y añade el otro método que necesitamos.

```
mouseButton1Up: aMouseButtonEvent localPosition: localEventPosition
   "I ignore the mouseEvent information and change my color."
   self color: Color random
```

Ahora, has creado una nueva clase Morph y puedes seleccionar un `ColorClickEllipse` en el menú World `New Morph...` y probarlo. Es divertido hacer *`clic-izquierdo`* sobre ellos. ¡Crea tantos como quieras!

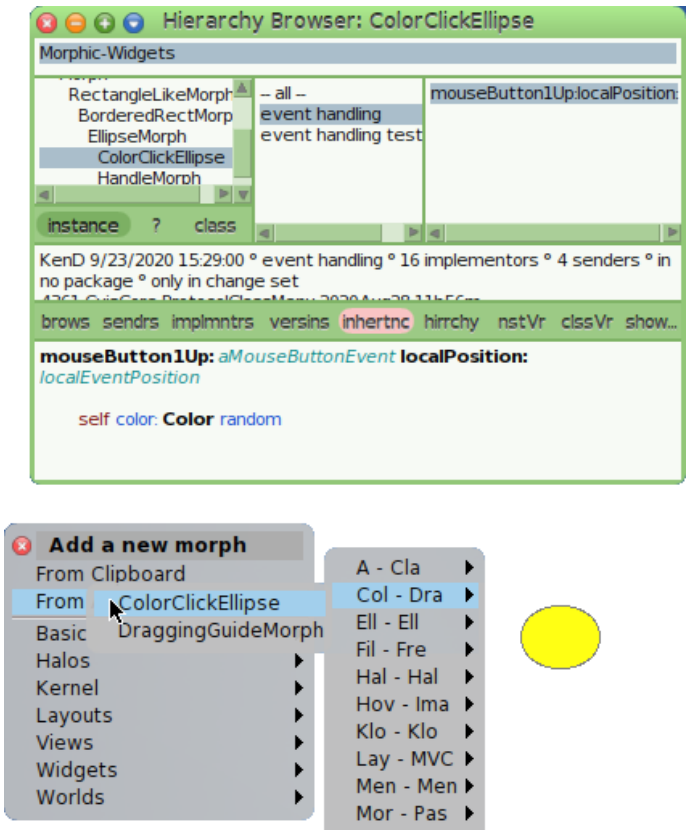

Figura 6.13: Obtener una `ColorClickEllipse`

Ahora ya sabes cómo especializar un morph individual o crear una clase completamente nueva.

6.6 Morphs de Spacewar!

6.6.1 Todos los Morphs

Anteriormente definimos a los actores del juego como subclases de la clase muy general `Object` (véase Ejemplo 3.14). Sin embargo, el juego, la estrella central, las naves y los torpedos son objetos visuales, cada uno con una forma gráfica específica:

- el juego consiste en un área rectangular simple rellena de color negro,
- la estrella central es un disco amarillo fluctuante con un halo naranja,

- las naves son figuras giratorias, cada una pintada con un color diferente,

- un torpedo es un triángulo giratorio que se pinta con un color diferente dependiendo de la nave que lo dispara.

Por lo tanto, tiene sentido convertir estos actores en tipos de `Morph`s, la entidad visual de Cuis-Smalltalk. Para ello, dirija un navegador del sistema a la definición de clase de cada actor, sustituya la clase padre `Object` por `PlacedMorph`[1], y luego guarde la definición de clase con *`Ctrl-s`*.

Por ejemplo, la clase torpedo tal y como se ve en Ejemplo 3.14 se modifica como:

```
PlacedMorph subclass: #Torpedo
   instanceVariableNames: 'position heading velocity lifeSpan'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Además, un *placed* Morph ya conoce su posición y orientación en la pantalla: se puede arrastrar y mover en la pantalla y girar con el cursor del ratón. Por lo tanto, las variables de instancia `position` y `heading` son redundantes y deben eliminarse. Por ahora las mantenemos, pero se eliminarán más adelante, cuando sepamos cómo sustituir cada uno de sus casos de uso por su equivalente Morph adecuado.

 *Modifica* `SpaceWar`*,* `CentralStar` *y* `SpaceShip` *para que sean subclases de la clase* `PlacedMorph`*.*

Ejercicio 6.1: Hacer todos los Morphs

Como se ha explicado en las secciones anteriores de este capítulo, un morph puede estar incrustado dentro de otro morph. En Spacewar!, una instancia morph `SpaceWar` que presenta el juego, es la *propietaria* de los morphs de la estrella central, la nave espacial y los torpedos. En otras palabras, la estrella central, las naves espaciales y los torpedos son *submorphs* de una instancia morph `SpaceWar`.

El código `SpaceWar>>initializeActors` en Ejemplo 4.17 no está completo sin añadir y posicionar la estrella central y las naves espaciales como submorfos del juego Spacewar!:

[1] Un `PlacedMorph` es un tipo de `Morph` con un atributo `location` suplementario, por lo que se le puede indicar que se mueva, se escale y gire en la pantalla.

```
SpaceWar>>initializeActors
   centralStar ← CentralStar new.
   self addMorph: centralStar.
   centralStar morphPosition: 0 @ 0.
   torpedoes ← OrderedCollection new.
   ships ← Array with: SpaceShip new with: SpaceShip new.
   self addAllMorphs: ships.
   ships first
      morphPosition: 200 @ -200;
      color: Color green.
   ships second
      morphPosition: -200 @ 200;
      color: Color red
```

Ejemplo 6.3: Completar el código para inicializar los actores de Spacewar!

Hay dos mensajes importantes: `#addMorph:` y `#morphPosition:`. El primero le pide al receptor morph que incruste su argumento morph como un submorph, el segundo le pide que establezca las coordenadas del receptor en el marco de referencia de su propietario. Al leer el código, se deduce que el origen del marco de referencia del propietario es su centro, de hecho, nuestra estrella central está en el centro del juego.

Hay un tercer mensaje que no aparece aquí, `#morphPosition`, para solicitar las coordenadas del receptor en el marco de referencia de su propietario.

Recuerda nuestra discusión sobre la variable de instancia `position`. Ahora entiendes claramente que es redundante y la eliminamos de las definiciones `SpaceShip` y `Torpedo`. Cada vez que necesitamos acceder a la posición, simplemente escribimos `self morphPosition` y cada vez que necesitamos modificar la posición escribimos `self morphPosition: newPosition`. Más adelante hablaremos más sobre esto.

6.6.2 El arte de refactorizar

En nuestro [modelo newtoniano], página 24, explicamos que las naves espaciales están sujetas a la aceleración del motor y a la fuerza gravitatoria de la estrella central. Las ecuaciones se describen en la Figura 2.4.

Basándonos en estas matemáticas, escribimos el método `SpaceShip>>update:` para actualizar la posición de la nave según el tiempo transcurrido; véase Ejemplo 4.19.

Hasta ahora, en nuestro modelo, un torpedo no está sujeto a la fuerza gravitatoria de la estrella central ni a la aceleración de su motor. Se supone que su masa es cero, lo cual es poco probable. Por supuesto, el método `Torpedo>>update:` es más sencillo que el de la nave espacial (véase Ejemplo 4.18. Sin embargo, es más preciso y aún más divertido que los torpedos estén sujetos a la fuerza gravitatoria[2] y a la aceleración de su motor; un piloto de nave espacial hábil podría utilizar la asistencia gravitatoria para acelerar un torpedo disparado con una trayectoria cercana a la estrella central.

¿Qué repercusiones tienen estas consideraciones en las entidades torpedo y nave espacial?

1. Compartirán *estados comunes* como la masa, la posición, el rumbo, la velocidad y la aceleración.
2. Compartirán *comportamientos comunes* como los cálculos necesarios para actualizar la posición, la dirección, la fuerza de gravedad y la velocidad.

[2] Por lo tanto, un torpedo debe tener masa.

3. Tendrán *diferentes estados*: un torpedo tiene un estado de vida útil, mientras que una nave espacial tiene un estado de capacidad del depósito de combustible y un estado de reserva de torpedos.
4. Tendrán *comportamientos diferentes*: un torpedo se autodestruye cuando expira su vida útil, una nave espacial dispara torpedos y acelera mientras su depósito de combustible y su recuento de torpedos no sean cero.

Los estados y comportamientos compartidos sugieren una clase común. Los estados y comportamientos no compartidos sugieren subclases especializadas que incorporan las diferencias. Así que «extraigamos» los elementos compartidos de las clases `SpaceShip` y `Torpedo` en una clase ancestral común, una más especializada que la clase `Morph` que comparten actualmente.

Realizar este análisis en el modelo informático del juego forma parte del esfuerzo de *refactorización* para evitar duplicaciones de comportamiento y estado, al tiempo que se hace más evidente la lógica común en las entidades. La idea general de la refactorización del código es reelaborar el código existente para hacerlo más elegante, comprensible y lógico.

Para ello, introduciremos una clase `Mobile`, un tipo de `PlacedMorph` con comportamientos específicos de un objeto móvil sometido a aceleraciones. Sus estados son la masa, la posición, el rumbo, la velocidad y la aceleración. Bueno, ya que estamos hablando de refactorización, el estado de masa no tiene mucho sentido en nuestro juego, ya que la masa de nuestro móvil es constante. Solo necesitamos un método que devuelva un número literal y entonces podremos eliminar la variable de instancia `mass`. Además, como se ha explicado anteriormente, una instancia `PlacedMorph` ya conoce su posición y rumbo, por lo que también eliminamos estos dos atributos, aunque hay comportamientos comunes a una nave espacial y un torpedo.

El resultado es esta definición de `Mobile`:

```
PlacedMorph subclass: #Mobile
   instanceVariableNames: 'velocity acceleration color'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Ejemplo 6.4: `Mobile` en el juego

¿Cuáles deberían ser las definiciones refactorizadas de las clases `SpaceShip` *y* `Torpedo`*?*

Ejercicio 6.2: Refactorizar `SpaceShip` y `Torpedo`

Los primeros comportamientos que añadimos a nuestro `Mobile` son su inicialización y su masa:

```
Mobile>>initialize
  super initialize.
  velocity ← 0 @ 0.
  acceleration ← 0
```

```
Mobile>>mass
  ↑ 1
```

Los siguientes métodos que se añadirán son los relativos a los cálculos físicos. En primer lugar, el código para calcular la aceleración gravitatoria:

```
Mobile>>gravity
"Compute the gravity acceleration vector"
   | position |
   position ← self morphPosition.
   ↑ -10 * self mass * owner starMass / (position r raisedTo: 3) * position
```

Ejemplo 6.5: Calcular la fuerza de la gravedad

Este método merece algunos comentarios:

- `self morphPosition` devuelve una instancia @.class Point@, la posición del móvil en el marco de referencia del propietario (*owner*).

- `owner` es la instancia `SpaceWar` que representa el juego. Es el propietario (morph padre) del móvil. Cuando se le pregunta `#starMass`, consulta la masa de su estrella central y devuelve su valor:

  ```
  SpaceWar>>starMass
    ↑ centralStar mass
  ```

  Como beneficio adicional, podemos eliminar el método `starMass` definido anteriormente en la clase `SpaceShip`.

- En `position r`, El mensaje `#r` solicita el atributo radio de un punto considerado en coordenadas polares. Es simplemente su longitud. Es la distancia entre el móvil y la estrella central.

- `* position` realmente significa multiplicar el valor escalar anterior por un `Point`, es decir, un vector. Por lo tanto, el valor devuelto es un `Point`, un vector en este contexto, el vector de gravedad.

El método para actualizar la posición y la velocidad del móvil es prácticamente el mismo que en Ejemplo 4.19. Por supuesto, las versiones `SpaceShip>>update:` y `Torpedo>>update:` deben eliminarse ambas. A continuación se muestra la versión completa con la forma en que morph accede a la posición del móvil:

```
Mobile>>update: t
"Update the mobile position and velocity"
  | ai ag newVelocity |
  "acceleration vectors"
  ai ← acceleration * self direction.
  ag ← self gravity.
  newVelocity ← (ai + ag) * t + velocity.
  self morphPosition:
     (0.5 * (ai + ag) * t squared)
     + (velocity * t)
     + self morphPosition.
  velocity ← newVelocity.
  "Are we out of screen? If so we move the mobile to the other corner
  and slow it down by a factor of 2"
  (self isInOuterSpace and: [self isGoingOuterSpace]) ifTrue: [
     velocity ← velocity / 2.
     self morphPosition: self morphPosition negated]
```

Ejemplo 6.6: Método `update:` de `Mobile`

Ahora debemos añadir los dos métodos para detectar cuándo un móvil se dirige al espacio profundo.

Pero primero definimos el método `localBounds` en cada uno de nuestros objetos Morph. Devuelve una instancia `Rectangle` definida en las coordenadas Morph por su origen y extensión:

```
SpaceWar>>localBounds
   ↑ -300 @ -300 extent: 600 @ 600

CentralStar>>localBounds
   ↑ Rectangle center: 0 @ 0 extent: self morphExtent

Mobile>>localBounds
   ↑ Rectangle encompassing: self class vertices
```

Ejemplo 6.7: Límites de nuestros objetos Morph

```
Mobile>>isInOuterSpace
"Is the mobile located in the outer space? (outside of the game
play area)"
   ↑ (owner localBounds containsPoint: self morphPosition) not

Mobile>>isGoingOuterSpace
"is the mobile going crazy in the direction of the outer space?"
   ↑ (self morphPosition dotProduct: velocity) > 0
```

Ejemplo 6.8: Comprobar que un mobile está «fuera del espacio»

Como puedes ver, estos métodos de prueba son sencillos y cortos. Al escribir código Cuis-Smalltalk, esto es algo que valoramos mucho y no dudamos en dividir un método largo en varios métodos pequeños. Mejora la legibilidad y la reutilización del código. El mensaje `#containsPoint:` pregunta al rectángulo receptor si el punto del argumento se encuentra dentro de su forma.

Cuando se actualiza un móvil, se actualizan su posición y velocidad. Sin embargo, las subclases `Mobile`, `SpaceShip` o `Torpedo` pueden necesitar actualizaciones específicas adi-

cionales. En la programación orientada a objetos existe un mecanismo especial denominado *sobrescritura* (*overriding*) para lograrlo.

Observa la definición de `Torpedo>>update:`:

See the `Torpedo>>update:` definition:

```
Torpedo>>update: t
   "Update the torpedo position"
   super update: t.
   "orientate the torpedo in its velocity direction, nicer effect
   while inaccurate"
   self heading: (velocity y arcTan: velocity x).
   lifeSpan ← lifeSpan - 1.
   lifeSpan isZero ifTrue: [owner destroyTorpedo: self].
   acceleration > 0 ifTrue: [acceleration ← acceleration - 500]
```

Aquí, el método `update:` está especializado para las necesidades específicas del torpedo. El cálculo mecánico realizado en `Mobile>>update:` sigue utilizándose para actualizar la posición y la velocidad del torpedo: esto se hace mediante `super update: t`. Ya hemos hablado de `super`. En el contexto de `Torpedo>>update:`, significa buscar un método @.msg update:@ en la clase padre de `Torpedo`, en la clase padre de esa clase, y así sucesivamente hasta encontrar el método; si no se encuentra, se señala un error de *Message Not Understood* (Mensaje no comprendido).

Entre los comportamientos añadidos específicos, la orientación del torpedo a lo largo de su vector de velocidad es inexacta, pero tiene un aspecto agradable. Para orientar el torpedo adecuadamente, ajustamos su rumbo con el rumbo de su vector de velocidad.

El control de la vida útil, la secuencia de autodestrucción y la aceleración del motor también se gestionan de forma específica. Cuando se dispara un torpedo, la aceleración de su motor es enorme, pero luego disminuye rápidamente.

Con el navegador del sistema apuntando al método `Torpedo>>update:`, observa el botón `inheritance`. Es de color verde claro, lo que indica que el mensaje también se envía a `super`. Esto es un recordatorio de que el método proporciona un comportamiento especializado. La información sobre herramientas del botón explica el significado de los colores resaltados en el texto del método. Al pulsar el botón `inheritance`, se pueden explorar todas las implementaciones del método `update:` dentro de esta cadena de herencia.

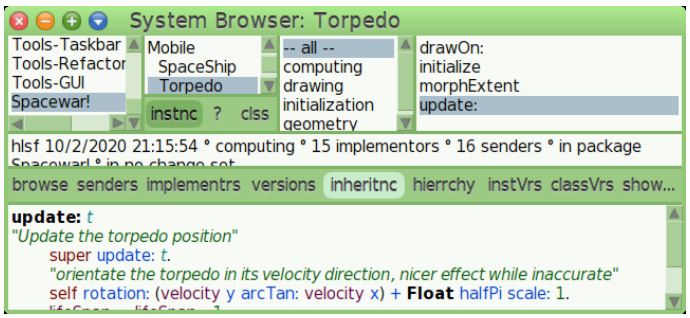

Figura 6.14: Botón `inheritance` de `update:`

Ya hemos visto un ejemplo de sobrescritura al inicializar una instancia de nave espacial: véase Ejemplo 3.17. En el contexto de la refactorización de nuestra clase, la sobrescritura de `initialize` abarca toda la jerarquía `Mobile`:

```
Mobile>>initialize
   super initialize.
   color ← Color gray.
   velocity ← 0 @ 0.
   acceleration ← 0

SpaceShip>>initialize
   super initialize.
   self resupply

Torpedo>>initialize
   super initialize.
   lifeSpan ← 500.
   acceleration ← 3000
```

Ejemplo 6.9: Sobreescribir Initialize en la jerarquía `Mobile` hierarchy

Observa cómo cada clase solo es responsable de la inicialización de su estado específico:

1. **SpaceShip.** Sus estados mecánicos se configuran con `super initialize` y, a continuación, se reabastece la nave con combustible y torpedos:

   ```
   SpaceShip>>resupply
      fuel ← 500.
      torpedoes ← 10
   ```

2. **Torpedo.** Estados mecánicos heredados inicializados; añadir inicialización de secuencia de autodestrucción y aceleración ajustada para imitar el impulso del torpedo al dispararse.

Los comportamientos específicos de cada móvil se establecen con métodos adicionales. El `SpaceShip` viene con sus métodos de control que ya describimos anteriormente en Ejemplo 5.8 y Ejemplo 5.9, por supuesto no hay ninguno para un `Torpedo`.

Otro comportamiento específico importante es cómo se dibuja cada tipo de `Mobile` en el juego, lo cual se tratará en el próximo capítulo sobre los fundamentos de Morph.

7 Los fundamentos de Morph

> Las cosas sencillas deben ser simples y las cosas complejas deben ser posibles.
> —*Alan Kay*

¿Qué podemos esperar si pedimos un buen soporte para crear interfaces gráficas de usuario en un sistema de programación?

En Capítulo 6 [Visual con Morph], página 74, comenzamos con esa misma pregunta y ofrecimos una visión general de los Morphs y su comportamiento interactivo. Este capítulo trata sobre cómo se construyen los Morphs, cómo crear nuevos Morphs y qué reglas siguen.

El framework de interfaz de usuario en Cuis-Smalltalk se llama Morphic. Morphic fue creado originalmente por Randy Smith y John Maloney como la UI para Self. Más tarde, John Maloney lo adaptó a Smalltalk para utilizarlo como la UI para Squeak.

7.1 Pasando a Vector

Para Cuis-Smalltalk, creamos Morphic 3, la tercera iteración de diseño de estas ideas, después de Morphic 1 de Self y Morphic 2 de Squeak. Si ya conoces Morphic en Self o Squeak, la mayoría de los conceptos son similares, aunque con algunas mejoras: las coordenadas de Morphic 3 no se limitan a ser números enteros, el tamaño aparente (nivel de zoom) de los elementos no está vinculado a la densidad de píxeles y todos los dibujos se realizan con antialiasing de alta calidad (subpíxel). Estas mejoras son posibles gracias al enorme avance en los recursos de //hardware// desde que se diseñaron Self y Squeak (a finales de los años 80 y 90, respectivamente). Además, el cuidadoso diseño del marco libera a los programadores de Morph de gran parte de la complejidad que se requería, especialmente en lo que respecta a la geometría.

7.1.1 Un primer ejemplo

Comencemos con algunos ejemplos. Lo que queremos es crear nuestros propios objetos gráficos, o Morphs. Una clase Morph forma parte de la jerarquía Morph y suele incluir un método `drawOn:` para dibujar su aspecto distintivo. Si nos olvidamos por un momento de los ordenadores y pensamos en dibujar con lápices de colores en una hoja de papel, una de las cosas más básicas que podemos hacer es dibujar líneas rectas.

Entonces, iniciemos una ventana del Browser del Sistema y construyamos un objeto de línea recta:

```
Morph subclass: #LineExampleMorph
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Morphic-Learning'
```

En la categoría de métodos `drawing` añade:

```
LineExampleMorph>>drawOn: aCanvas
   aCanvas strokeWidth: 20 color: Color green do: [
      aCanvas
         moveTo: 100 @ 100;
         lineTo: 400 @ 200 ].
```

Ahora, en un Workspace, ejecuta:

```
LineExampleMorph new openInWorld
```

Si aparece un mensaje preguntándole si desea instalar y activar la compatibilidad con gráficos vectoriales, responde que sí. Ya está. Ya has creado tu primera clase `Morph`.

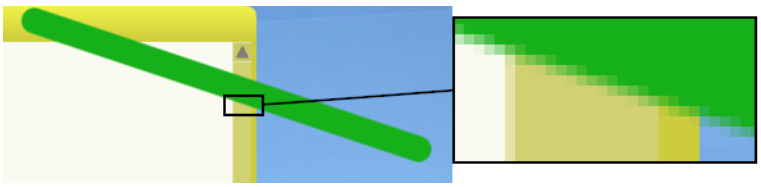

Figura 7.1: Detalle de nuestro morph de línea

El código es evidente, el método `drawOn:` toma una instancia `VectorCanvas` como argumento. `VectorCanvas` proporciona muchas operaciones de dibujo para que las utilicen los morphs. Puedes jugar con las diversas operaciones de dibujo y sus parámetros, y ver el resultado. Si cometes un error y el método `drawOn:` falla, aparecerá un cuadro de error rojo y amarillo. Después de corregir el método `drawOn:`, ve al ...menú World → `Debug...` → `Start drawing all again...` para que tu morph se vuelva a dibujar correctamente.

¿Cómo modificarás nuestra línea morph para que se dibuje como una cruz con una extensión de 200 píxeles?

Ejercicio 7.1: Morph cruz

7.1.2 Morph que puedes mover

Es posible que ya hayas intentado hacer clic y arrastrar tu línea, como se puede hacer con las ventanas normales y la mayoría de los demás Morphs. Si no lo has hecho, pruébalo ahora. ¡Pero no pasa nada! El motivo es que nuestro Morph está fijado en un lugar del morph propietario (el WorldMorph). Está fijado porque `drawOn:` dice que debe ser una línea entre `100 100` y `400 200`. Moverlo significaría modificar esos puntos. Una forma posible de hacerlo podría ser almacenar esos puntos en variables de instancia.

Pero ahora sólo deseamos codificar nuestro morph de la manera más sencilla posible y seguir pudiendo moverlo. La solución es convertirlo en una subclase de `PlacedMorph`, en lugar de `Morph`.

Para ello, primero evalúa el código siguiente para eliminar todas las instancias de `LineExampleMorph`:

```
 LineExampleMorph allInstancesDo: [ :m | m
delete]
```

Ejemplo 7.1: Borrar todas las instancias de un morph dado

A continuación, en la declaración de clase del navegador del sistema para LineExampleMorph, escribe `PlacedMorph` en lugar de `Morph` y guarda. Ahora vuelve a ejecutar:

```
LineExampleMorph new openInWorld
```

Obtendrá una línea que podrá seleccionar con el ratón y moverla. `PlacedMorph` añade una nueva variable de instancia llamada `location`. Si un morph tiene una `location`, se puede mover modificándola. La `location` también define un nuevo sistema de coordenadas local. Todas las coordenadas utilizadas en el método `drawOn:` ahora son relativas a este nuevo sistema de coordenadas. Por eso no necesitamos modificar el método `drawOn:`. `drawOn:` ahora indica cómo se debe dibujar el morph, pero no dónde. La `location` también especifica un posible factor de rotación y escala. Esto significa que las subinstancias de `PlacedMorph` también se pueden rotar y ampliar.

7.1.3 Morph relleno

Creemos otro morph para tener más diversión.

```
PlacedMorph subclass: #TriangleExampleMorph
   instanceVariableNames: 'borderColor fillColor'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Morphic-Learning'
```

En la categoría de métodos `initialization`, añade:

```
TriangleExampleMorph>>initialize
   super initialize.
   borderColor ← Color random alpha: 0.8.
   fillColor ← Color random alpha: 0.6.
```

En la categoría de métodos `drawing`, añade:

```
TriangleExampleMorph>>drawOn: aCanvas
   aCanvas strokeWidth: 10 color: borderColor fillColor: fillColor do: [
      aCanvas
         moveTo: 0 @ 100;
         lineTo: 87 @ -50;
         lineTo: -87 @ -50;
         lineTo: 0 @ 100 ].
```

Tómate un momento para comprender ese código, para adivinar qué hará. Ahora ejecuta:

```
TriangleExampleMorph new openInWorld
```

Hazlo varias veces y mueve cada triángulo. Cada nuevo triángulo que crees tendrá colores diferentes. Y estos colores no son completamente opacos. Esto significa que cuando coloques tu triángulo sobre otra forma, podrás ver a través de él.

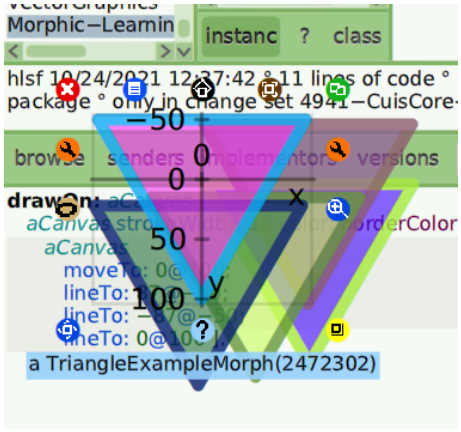

Figura 7.2: Varios morphs triangulares, uno decorado con su halo y sistema de coordenadas

¿Cómo escribirías un rectángulo móvil con unas dimensiones x,y de 200 por 100? El rectángulo se rellenará con un color translúcido aleatorio y estará rodeado por una fina línea azul.

Ejercicio 7.2: Morph rectángulo

Como hemos aprendido anteriormente, Morphic te ofrece formas adicionales de interactuar con tus morphs. Con un ratón de tres botones o un ratón con rueda, coloca el puntero del ratón (una instancia de `HandMorph`) sobre uno de tus triángulos y haz clic con el botón central o la rueda del ratón. Si no tienes un ratón de tres botones, sustituye por *`Command-clic`*. Obtendrás una constelación de pequeños círculos de colores alrededor de tu morph. Esto se denomina *halo* del morph, y cada círculo de color es un *controlador de halo*. Véase Figura 7.2.

En la parte superior izquierda tienes el controlador rojo `Remove` (Eliminar). Al hacer clic en él, simplemente se elimina el morph del mundo. Pasa el cursor por encima de cada controlador y aparecerá una ventana emergente con su nombre. Otros controladores te permiten `Duplicate` (Duplicar) un morph, abrir un `Menu` (Menú) con acciones, `Pick up` (Recoger) (lo mismo que arrastrarlo con el ratón como hiciste antes). La operación `Move` (Mover) es similar a `Pick up` (Recoger), pero no elimina el morph del propietario actual. Más adelante se ofrece más información al respecto. El controlador `Debug` (Depurar) abre un menú desde el que se puede abrir un Inspector o un Browser jerárquico para estudiar el morph.

También tienes los controles `Rotate` (Girar) y `Change scale` (Cambiar escala). ¡Pruébalos! Para utilizarlos, mueve la mano hacia el control, pulsa el botón del ratón y arrástralo. Como habrás adivinado, los controles de rotación hacen girar tu morph alrededor del centro del rectángulo que lo rodea. Los controles de escala controlan el zoom aparente aplicado a tu morph. Tanto la escala como la rotación (y también el desplazamiento, como cuando mueves tu morph) se implementan modificando el sistema

de coordenadas interno definido por tu morph. El desplazamiento, la rotación y la escala son números de punto flotante, por lo que no se limitan a números enteros.

Para cambiar el centro de rotación de un Morph, sobreescribe el método `rotationCenter` según corresponda:

```
RectangleExampleMorph>>rotationCenter
   ↑0 @ 0
```

Observa cómo nuestro rectángulo morph reacciona ahora al controlador de rotación. Aprenderemos a controlar todo esto con código y a animar nuestro morph.

7.1.4 Morph animado

Añadamos dos métodos a nuestro `TriangleExampleMorph` para *dar vida* a nuestro triángulo:

En la categoría de métodos `stepping`, define:

```
TriangleExampleMorph>>wantsSteps
   ↑ true
```

...y:

```
TriangleExampleMorph>>step
   fillColor ← Color random.
   self redrawNeeded
```

A continuación, crea algunos triángulos adicionales como hiciste antes.

Esto hará que nuestros triángulos cambien de color una vez por segundo. Pero lo más interesante es editar el método:

```
TriangleExampleMorph>>stepTime
   ↑ 100
```

...y:

```
TriangleExampleMorph>>step
   self morphPosition: self morphPosition + (0.4@0).
   self redrawNeeded
```

Ahora, nuestro morph se mueve diez veces por segundo y se desplaza hacia la derecha a una velocidad de cuatro píxeles por segundo. En cada paso se desplaza 0.4 píxeles, y no un número entero de píxeles. ¡El dibujo con antialiasing de alta calidad nos permite hacerlo! Puedes hacer que se mueva a una velocidad de cuatro veces por segundo y que se desplace 1 píxel cada vez, y verás lo diferente que se ve.

Ahora prueba esto:

```
TriangleExampleMorph>>step
   self morphPosition: self morphPosition + (0.2@0).
   self rotateBy: 4 degreesToRadians.
   self redrawNeeded
```

Y aún hay más. Primero, elimina todas las instancias:

```
TriangleExampleMorph allInstancesDo: [ :m | m delete]
```

Y modifica estos métodos:

```
TriangleExampleMorph>>initialize
   super initialize.
   borderColor ← Color random alpha: 0.8.
   fillColor ← Color random alpha: 0.6.
   scaleBy ← 1.1
```

Acepta `scaleBy` como una nueva variable de instancia de la clase `TriangleExampleMorph`.

```
TriangleExampleMorph>>step
   self morphPosition: self morphPosition + (0.2@0).
   self rotateBy: 4 degreesToRadians.
   self scaleBy: scaleBy.
   self scale > 1.2 ifTrue: [scaleBy ← 0.9].
   self scale < 0.2 ifTrue: [scaleBy ← 1.1].
   self redrawNeeded
```

Después, crea un nuevo triángulo:

```
TriangleExampleMorph new openInWorld
```

Fíjate en que, cuando el triángulo está haciendo su loca danza, aún puedes abrir un halo e interactuar con él.

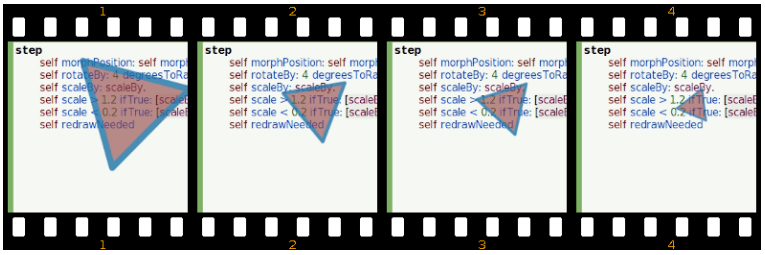

Figura 7.3: Morph animado

7.1.5 Morph dentro de morphs

Ahora, probemos algo diferente. Coge uno de tus `LineExampleMorph`. Con el halo, amplíalo hasta que tenga aproximadamente el tamaño de tu triángulo. Ahora coloca el triángulo sobre tu línea. Abre un halo en el triángulo, haz clic en el controlador `Menu` (Menú) y selecciona ...`embed into` (incrustar en) → `LineExampleMorph`. Esto convierte al triángulo en un submorph de la línea. Ahora, si mueves, escalas o giras la línea, el triángulo también se ajusta.

Puedes abrir un halo en el triángulo. Para ello, haz doble clic con el botón central del ratón sobre él. Con el halo en el triángulo, puedes girarlo o ampliarlo independientemente de la línea. Ten en cuenta también que cuando agarras el triángulo con la mano (sin usar el halo), agarras la línea + el triángulo compuesto. No puedes simplemente arrastrar el

triángulo. Para ello, necesitas el halo del triángulo. Usa su controlador `Move` (Mover)[1] para colocarlo *sin sacarlo* de la línea. Usa su controlador `Pick up` para cogerlo con la mano y soltarlo en el mundo. Ahora, el triángulo ya no es un submorph de la línea, y los morphs se pueden mover, rotar o escalar de forma independiente.

Pero probemos algo. Volvamos a crear el submorph triangular de la línea. Ahora añadamos el siguiente método a la categoría `geometry testing` de la clase `LineExampleMorph`:

```
LineExampleMorph>>clipsSubmorphs
   ↑ true
```

El dibujo del triángulo se corta exactamente en los límites de la línea. Esto resulta muy útil para implementar paneles de desplazamiento que solo muestran una parte de su contenido, pero también puede tener otros usos.

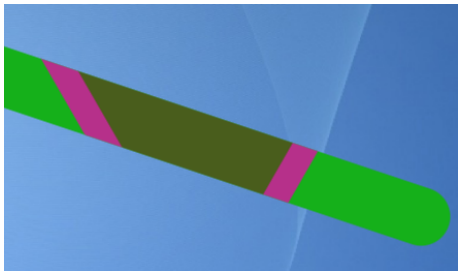

Figura 7.4: Un submorph tirangular animado y recortado

7.2 Un Morph reloj

Con todo lo que ya hemos aprendido, podemos crear un morph más sofisticado. Creemos un `ClockMorph` como se ve en la Figura 7.5.

Figura 7.5: Un morph reloj

Creemos `ClockMorph`, el reloj de esfera:

```
PlacedMorph subclass: #ClockMorph
   instanceVariableNames: ''
```

[1] A estas alturas, es probable que el triángulo se haya desplazado bastante

```
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

...y su método de dibujo en la categoría `drawing`:

```
ClockMorph>>drawOn: aCanvas
    aCanvas
        ellipseCenter: 0@0
        radius: 100
        borderWidth: 10
        borderColor: Color lightCyan
        fillColor: Color veryVeryLightGray.
    aCanvas drawString: 'XII' at: -13 @ -90 font: nil color: Color brown.
    aCanvas drawString: 'III' at: 66 @ -10 font: nil color: Color brown.
    aCanvas drawString: 'VI' at: -11 @ 70 font: nil color: Color brown.
    aCanvas drawString: 'IX' at: -90 @ -10 font: nil color: Color brown
```

Ejemplo 7.2: Dibujando el reloj de esfera

Creamos `ClockHourHandMorph`, la manecilla para las horas:

```
PlacedMorph subclass: #ClockHourHandMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

...y su método de dibujado en la categoría `drawing`:

```
ClockHourHandMorph>>drawOn: aCanvas
    aCanvas fillColor: (Color black alpha: 0.6) do: [
        aCanvas
            moveTo: 0 @ 10;
            lineTo: -5 @ 0;
            lineTo: 0 @ -50;
            lineTo: 5 @ 0;
            lineTo: 0 @ 10 ].
```

Ya puedes empezar a jugar con ellos. Podríamos usar varias instancias de un único `ClockHandMorph`, o crear varias clases. Aquí hemos optado por lo segundo. Ten en cuenta que todos los métodos `drawOn:` utilizan constantes codificadas para todas las coordenadas. Como hemos visto anteriormente, esto no es una limitación. ¡No necesitamos escribir un montón de fórmulas trigonométricas y de escalado especializadas para crear Morphs en Cuis-Smalltalk!

A estas alturas, quizá ya te imagines lo que estamos haciendo con todo esto, pero ten paciencia mientras terminamos de construir nuestro reloj.

Creamos `ClockMinuteHandMorph`, la manecilla de los minutos:

```
PlacedMorph subclass: #ClockMinuteHandMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Learning'
```

...y su método de dibujado en la categoría `drawing`:

```
ClockMinuteHandMorph>>drawOn: aCanvas
   aCanvas fillColor: ((Color black) alpha: 0.6) do: [
      aCanvas
         moveTo: 0 @ 8;
         lineTo: -4 @ 0;
         lineTo: 0 @ -82;
         lineTo: 4 @ 0;
         lineTo: 0 @ 8 ]
```

Y por último, la `ClockSecondHandMorph`, la manecilla de los segundos:

```
PlacedMorph subclass: #ClockSecondHandMorph
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Morphic-Learning'
```

...y su método de dibujado en la categoría `drawing`:

```
ClockSecondHandMorph>>drawOn: aCanvas
   aCanvas strokeWidth: 2.5 color: Color red do: [
      aCanvas
         moveTo: 0 @ 0;
         lineTo: 0 @ -85 ]
```

Ahora, solo queda juntar las piezas de nuestro reloj en `ClockMorph`. En su categoría de métodos `initialization`, añade su método `initialize` (acepta los nuevos nombres como variables de instancia):

```
ClockMorph>>initialize
   super initialize.
   self addMorph: (hourHand ← ClockHourHandMorph new).
   self addMorph: (minuteHand ← ClockMinuteHandMorph new).
   self addMorph: (secondHand ← ClockSecondHandMorph new)
```

Si aún no has añadido variables de instancia para las manecillas del reloj, el IDE Cuis lo detectará y te preguntará qué quieres hacer al respecto. Queremos declarar los tres nombres que faltan como variables de instancia.

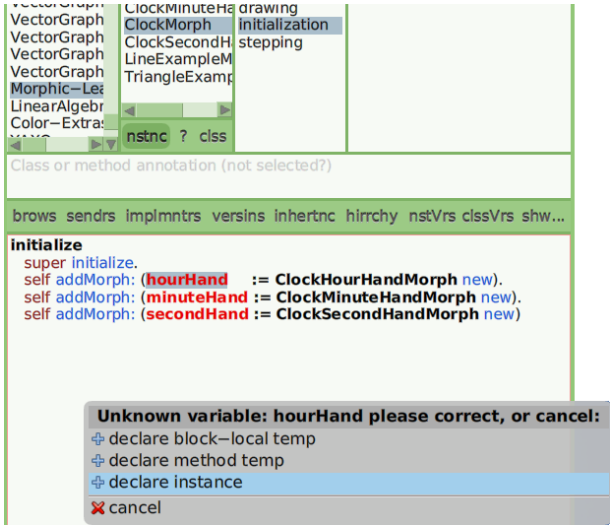

Figura 7.6: Declarar variables desconocidas como variables de instancia en la clase actual

¡La definición de tu clase `ClockMorph` ya debería estar completa!

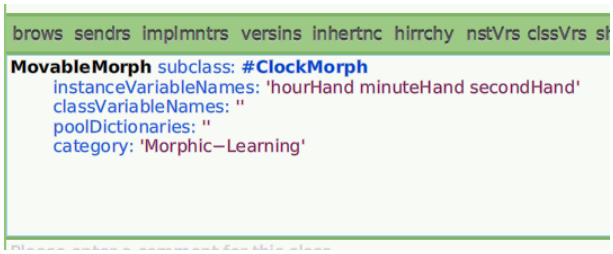

Figura 7.7: `ClockMorph` con variables de instancia añadidas

Por último, animamos nuestro reloj. En la categoría de métodos `stepping`, añade el método:

```
ClockMorph>>wantsSteps
    ↑ true
```

...y:

```
ClockMorph>>step
    | time |
    time ← Time now.
    hourHand rotationDegrees: time hour * 30.
    minuteHand rotationDegrees: time minute * 6.
    secondHand rotationDegrees: time second * 6
```

Echa un vistazo a cómo actualizamos las manecillas del reloj.

Como dijimos anteriormente, cualquier `PlacedMorph` define un sistema de coordenadas para su propio método `drawOn:` y también para sus submorfos. Este nuevo sistema de coordenadas podría incluir la rotación o reflexión del eje y el escalado de tamaños, pero por defecto no lo hace. Esto significa que solo trasladan el origen, especificando dónde se ubicará el punto propietario `0 @ 0`.

El sistema de coordenadas del World tiene `0 @ 0` en la esquina superior izquierda, con las coordenadas X aumentando hacia la derecha y las coordenadas Y aumentando hacia abajo. Las rotaciones positivas van en el sentido de las agujas del reloj. Esta es la convención habitual en los marcos gráficos. Tenga en cuenta que esto difiere de la convención matemática habitual, en la que Y aumenta hacia arriba y los ángulos positivos van en sentido contrario a las agujas del reloj.

Entonces, ¿cómo actualizamos las manecillas? Por ejemplo, para la manecilla de las horas, una hora significa 30 grados, ya que 12 horas significan 360 grados o una vuelta completa. Por lo tanto, multiplicamos las horas por 30 para obtener los grados. Las manecillas de los minutos y los segundos funcionan de manera similar, pero como hay 60 minutos en una hora y 60 segundos en un minuto, debemos multiplicarlos por 6 para obtener los grados. Como la rotación se realiza alrededor del origen, y el reloj ha establecido el origen en su centro (Ejemplo 7.2), no es necesario establecer la posición de las manecillas. Por lo tanto, su origen `0 @ 0` estará en el reloj `0 @ 0`, es decir, en el centro del reloj.

Figura 7.8: Un elegante morph de reloj

Mira el reloj de la Figura 7.8. ¿No te parece elegante la manecilla de los segundos decorada con un disco rojo y amarillo? ¿Cómo modificarías nuestro morph de reloj para obtener este resultado?

Ejercicio 7.3: Un reloj elegante

Crea algunas instancias de tu reloj: `ClockMorph new openInWorld`. Puedes girarlo y ampliarlo. Fíjate en la calidad visual de los números romanos de la esfera del reloj, especialmente cuando se gira y se amplía. ¡No obtendrás esta calidad gráfica en tu entorno de programación habitual! También puedes extraer las partes o escalarlas por separado. Otro experimento divertido consiste en extraer los números romanos a un `ClockFaceMorph` independiente y convertirlo en un submorph del reloj. A continuación, puedes girar solo la esfera, no el reloj, y este mostrará una hora falsa. ¡Pruébalo!

Sin embargo, es posible que hayas notado que faltan dos cosas: cómo calcular los rectángulos delimitadores para los Morphs y cómo detectar si un Morph está siendo tocado por el cursor, para poder moverlo u obtener un halo. El rectángulo de visualización que contiene completamente un morph es necesario para que el marco gestione la actualización requerida de las áreas de visualización como resultado de cualquier cambio. Pero no es necesario conocer este rectángulo para crear tus propios Morphs. En Cuis-Smalltalk, el marco lo calcula según sea necesario y lo almacena en la variable `privateDisplayBounds`. No tienes que preocuparte en absoluto por esa variable.

Con respecto a detectar si el cursor está tocando un Morph o, en términos más generales, si algún píxel pertenece a un Morph, lo cierto es que durante la operación de dibujo de un Morph, el marco conoce todos los píxeles a los que afecta. El método `drawOn:` especifica completamente la forma del Morph. Por lo tanto, no es necesario pedir al programador que vuelva a codificar la geometría del Morph en un método separado. Lo único que se necesita es un diseño cuidadoso del propio marco, para evitar que los programadores tengan que lidiar con esta complejidad adicional.

Las ideas que hemos esbozado en este capítulo son las fundamentales en Morphic, y el marco se implementa con el fin de respaldarlas. Los morphs (es decir, los objetos gráficos interactivos) son muy generales y flexibles. No se limitan a una biblioteca de widgets convencional, aunque se incluye una biblioteca de este tipo (basada en `BoxedMorph`) que se utiliza para crear todas las herramientas de Smalltalk.

Los ejemplos que hemos explorado utilizan el marco `VectorGraphics`. Incluye las clases `VectorCanvas` y `HybridCanvas`. Cuis-Smalltalk también incluye la clase heredada `BitBltCanvas`, heredada de Squeak. `BitBltCanvas` no admite las operaciones de dibujo de gráficos vectoriales y no realiza suavizado ni zoom. Sin embargo, es madura y se basa en la operación BitBlt incluida en la VM. Esto significa que ofrece un rendimiento excelente.

Para explorar más a fondo Morphic de Cuis-Smalltalk, evalúe `Feature require: 'SVG'` y, a continuación, `SVGMainMorph exampleLion openInWorld` y los demás ejemplos que hay allí. Además, asegúrese de probar los ejemplos de la categoría de clase `Morphic-Examples`, entre ellos, ejecute `Sample10PythagorasTree new openInWorld` y juegue con las direcciones arriba y abajo, izquierda y derecha de la rueda del ratón.

Antes de dejar esta sección, aquí hay un cambio de dos líneas para convertir nuestro reloj de cuarzo Cuis[2] en un reloj suizo automático[3][4]:

```
ClockMorph>>stepTime
    ^ 100 "milliseconds"
ClockMorph>>step
../..
   secondHand rotationDegrees: (time second + (time nanoSecond * 1.0e-9))* 6
```

Intenta comprender cómo afectan estos cambios al comportamiento del segundero y en qué fracción de segundo gira.

7.3 Volviendo a los Morphs de Spacewar!

[2] En un reloj de cuarzo, la manecilla de los segundos se mueve cada segundo

[3] En un reloj automático, la manecilla de los segundos se mueve cada fracción de segundo. Cuanto menor sea la fracción, mayor será la calidad del reloj

[4] Un reloj automático japonés también servirá

7.3.1 Estrella central

Nuestra estrella central tiene una extensión de `30 @ 30` que necesitamos utilizar en varios lugares del código. Por lo tanto, tiene sentido definir un método específico para responder a este valor:

```
CentralStar>>morphExtent
   ↑ `30 @ 30`
```

Ejemplo 7.3: Extensión de la estrella central

Una expresión rodeada por comillas invertidas `` '`' `` solo se evalúa una vez, cuando el método se guarda y compila por primera vez. Esto crea un valor literal compuesto y mejora el rendimiento del método, ya que la expresión no se evalúa cada vez que se llama al método: en su lugar, se utiliza el valor precompilado.

Como aprendiste anteriormente, un morph se dibuja a sí mismo desde su método `drawOn:`. Dibujamos la estrella como una elipse con radios x e y que fluctúan aleatoriamente:

```
CentralStar>>drawOn: canvas
   | radius |
   radius ← self morphExtent // 2.
   canvas ellipseCenter: 0 @ 0
      radius: (radius x + (2 atRandom - 1)) @ (radius y + (2 atRandom - 1))
      borderWidth: 3
      borderColor: Color orange
      fillColor: Color yellow
```

Ejemplo 7.4: Una estrella con un tamaño fluctuante

Los diámetros de la estrella en las direcciones x e y fluctúan independientemente entre 0 y 2 unidades. La estrella no parece perfectamente redonda.

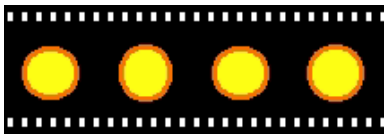

Figura 7.9: Una estrella con un tamaño fluctuante

7.3.2 Nave espacial

Al inicio del juego, la proa de la nave espacial apunta hacia la parte superior de la pantalla, como se ve en la Figura 7.10, por lo que el ángulo de su dirección es de -90°, mientras que el ángulo de su rotación es de 0°. Recuerda que las coordenadas Y están orientadas hacia la parte inferior de la pantalla.

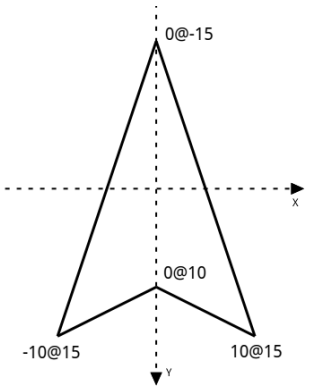

Figura 7.10: Diagrama de la nave espacial al inicio del juego

Entonces, su método `drawOn:` se escribe así:

```
SpaceShip>>drawOn: canvas
   | a b c d |
   a ← 0 @ -15.
   b ← -10 @ 15.
   c ← 0 @ 10.
   d ← 10 @ 15.
   canvas line: a to: b width: 2 color: color.
   canvas line: b to: c width: 2 color: color.
   canvas line: c to: d width: 2 color: color.
   canvas line: d to: a width: 2 color: color.
   "Draw gas exhaust"
   acceleration ifNotZero: [
      canvas line: c to: 0 @ 35 width: 1 color: Color gray]
```

Ejemplo 7.5: Dibujado de nave espacial

¿Cómo modificarás el método `drawOn:` *de la nave espacial para que el dibujo del escape de gas dependa de la tasa de aceleración?*

Ejercicio 7.4: Dibujo del chorro de gases

Cuando hay una aceleración del motor, dibujamos una pequeña línea gris para representar el chorro de gas.

Cuando el usuario gira la nave, la transformación se rota ligeramente ajustando su rumbo:

```
SpaceShip>>right
"Rotate the ship to its right"
   self heading: self heading + 0.1

SpaceShip>>left
"Rotate the ship to its left"
```

```
self heading: self heading - 0.1
```

Pero, ¿cómo afecta este rumbo a la rotación del morph?

Debajo, `MobileMorph` está equipado con una transformación afín para escalar, rotar y traducir las coordenadas pasadas como argumentos a los mensajes de dibujo recibidos por el lienzo. Por lo tanto, nuestros métodos `heading` se definen para que coincidan con esta representación interna y utilizamos el método `rotation:` de la clase `PlacedMorph@` para rotar adecuadamente. El atributo `location` representa una transformación afín, y obtenemos su ángulo de rotación con el mensaje `#radians`.

```
Mobile>>heading
   ↑ location radians - Float halfPi
Mobile>>heading: aHeading
   self rotation: aHeading + Float halfPi
```

Cuando un móvil está en posición vertical, apuntando hacia la parte superior de la pantalla, su rumbo es de -90° (-pi/2) en el sistema de coordenadas de la pantalla. En esa situación, el móvil no está girado, o bien está girado 0°, por lo que añadimos 90° (pi/2) al rumbo para obtener el ángulo de rotación correspondiente del móvil.

7.3.3 Torpedo

Al igual que una nave espacial, cuando se instancia un torpedo, su proa apunta hacia la parte superior de la pantalla y sus vértices vienen dados por la Figura 7.11.

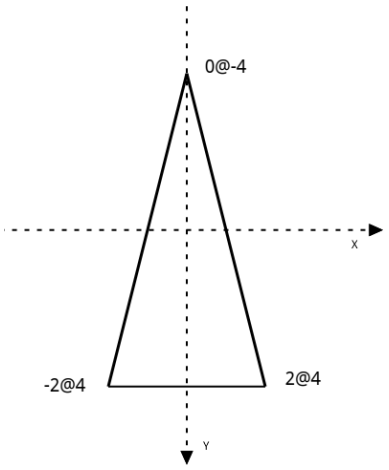

Figura 7.11: Diagrama de torpedo al inicio del juego

Dados los vértices proporcionados por la Figura 7.11, ¿cómo escribirías su método `morphExtent`*?*

Ejercicio 7.5: Extensión del torpedo

¿Cómo escribirás el método `drawOn:` *de* `Torpedo`*?*

Ejercicio 7.6: Dibujo de torpedo

7.3.4 Revisar el dibujado

Como habrás observado, los métodos `drawOn:` de `SpaceShip` y `Torpedo` comparten la misma lógica: dibujar un polígono a partir de sus vértices. Probablemente queramos trasladar esta lógica común a su antecesor común, la clase `Mobile`. Necesita conocer sus vértices, por lo que quizá queramos añadir una variable de instancia `vertices` inicializada en sus subclases con una matriz que contenga los puntos:

```
PlacedMorph subclass: #Mobile
   instanceVariableNames: 'acceleration color velocity vertices'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

SpaceShip>>initialize
   super initialize.
   vertices ← {0@-15 . -10@15 . 0@10 . 10@15}.
   self resupply

Torpedo>>initialize
   super initialize.
   vertices ← {0@-4 . -2@4 . 2@4}.
   lifeSpan ← 500.
   acceleration ← 3000
```

Sin embargo, esto no es una buena idea. Imagina el juego con 200 torpedos: ¡la matriz de vértices se duplicaría 200 veces con los mismos datos!

Variable de clase

En ese tipo de situación, lo que se necesita es una *variable de clase* definida en el lado de la clase, en contraste con el lado de la instancia, donde hemos estado programando hasta ahora.

Aprovechamos el hecho de que todos los objetos son instancias de alguna clase. ¡La `Mobile class` es una instancia de la clase `Class`!

1. Una *variable de clase* sólo puede ser accedida desde la propia clase en un *método de clase*.
2. Una entidad (por ejemplo, un torpedo disparado) puede acceder a las variables de clase a través de *métodos de clase*, enviando un mensaje a una clase (por ejemplo, `Torpedo`) en lugar de a sí misma o a otra entidad.
3. En la jerarquía de clases, cada subclase tiene su propia instancia de la variable de instancia de clase y puede asignarle un valor diferente, a diferencia de una *variable de clase*, que se comparte entre todas las subclases (se explica más adelante).
4. Para editar las variables de *clase* y los métodos de *clase*, en el Browser del sistema, pulsa el botón `class` situado debajo de la lista de clases.

En el navegador del sistema, hacemos clic en el botón `class` y luego declaramos nuestra variable en la definición de la `Mobile class` – Figura 7.12:

```
Mobile class
   instanceVariableNames: 'vertices'
```

Ejemplo 7.6: `vertices` como variable instanciada en la `Mobile class`

A continuación, escribimos un método de acceso en `Mobile class`, para que las instancias `SpaceShip` y `Torpedo` puedan acceder a él:

```
Mobile class>>vertices
   ↑ vertices
```

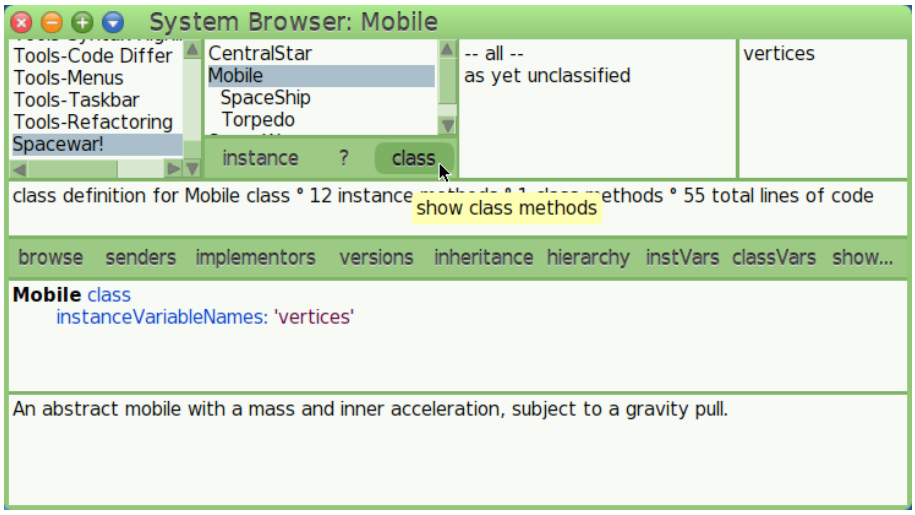

Figura 7.12: El lado de clase del Browser del sistema

A continuación, cada subclase es responsable de inicializar correctamente `vertices` con su método de clase `initialize`:

```
SpaceShip class>>initialize
"SpaceShip initialize"
   vertices ← {0@-15 . -10@15 . 0@10 . 10@15}

Torpedo class>>initialize
"Torpedo initialize"
   vertices ← {0@-4 . -2@4 . 2@4}
```

Ejemplo 7.7: Inicializar una clase

Cuando se instala una clase en Cuis-Smalltalk, se ejecuta su método de clase `initialize`. También puedes seleccionar el método y ejecutalo con *`Ctrl-d`*.

Experimenta en un espacio de trabajo para comprender cómo se comporta una variable de instancia de clase:

```
SpaceShip vertices.
⇒ nil
SpaceShip initialize.
SpaceShip vertices.
⇒ #(0@-15 -10@15 0@10 10@15)

Torpedo vertices.
⇒ nil
Torpedo initialize.
Torpedo vertices.
⇒ #(0@-4 -2@4 2@4)
```

Ejemplo 7.8: El valor de una variable de clase no es compartido por las subclases

Este es realmente el comportamiento que queremos: las instancias `SpaceShip` y `Torpedo` tienen un diagrama diferente. Sin embargo, todas las instancias de una `SpaceShip` tendrán el mismo diagrama, haciendo referencia a la misma matriz `vertices` (es decir, la misma ubicación en la memoria del ordenador).

Cada instancia pregunta a su lado de clase con el mensaje `#class`:

```
aTorpedo class
⇒ Torpedo
self class
⇒ SpaceShip
```

El método `drawOn:` de `Torpedo` se reescribe para acceder a los vértices en su lado de clase:

```
Torpedo>>drawOn: canvas
   | vertices |
   vertices ← self class vertices.
   canvas line: vertices first to: vertices second width: 2 color: color.
   canvas line: vertices third to: vertices second width: 2 color: color.
   canvas line: vertices first to: vertices third width: 2 color: color
```

¿Cómo reescribirías `drawOn:` *de* `SpaceShip` *para utilizar los vértices de su lado de clase?*

Ejercicio 7.7: Acceso de la nave espacial a su diagrama en el lado de la clase

Hasta ahora, seguimos teniendo esta redundancia en los métodos `drawOn:`. Lo que queremos es que `Mobile` se encargue de dibujar el polígono dada una matriz de vértices: `self drawOn: canvas polygon: vertices`.

El método `drawOn:` de `SpaceShip` y `Torpedo` se escribirá simplemente como:

```
Torpedo>>drawOn: canvas
   self drawOn: canvas polygon: self class vertices

SpaceShip>>drawOn: canvas
   | vertices |
```

```
vertices ← self class vertices.
self drawOn: canvas polygon: vertices.
"Draw gas exhaust"
acceleration ifNotZero: [
   canvas line: vertices third to: 0@35 width: 1 color: Color gray]
```

¿Cómo escribirías el método `drawOn:polygon:` *en* `Mobile`*? Sugerencia: utiliza el iterador* `withIndexDo:`*.*

Ejercicio 7.8: Dibujar en `Mobile`

Variable de clase

Una *variable de clase* se escribe en mayúsculas en el argumento de la palabra clave `classVariableNames:`:

```
PlacedMorph subclass: #Mobile
   instanceVariableNames: 'acceleration color velocity'
   classVariableNames: 'Vertices'
   poolDictionaries: ''
   category: 'Spacewar!'
```

Ejemplo 7.9: `Vertices` una variable de clase `Mobile`

Como variable de instancia de clase, se puede acceder a ella directamente desde el lado de la clase y las instancias solo tienen acceso mediante mensajes enviados al lado de la clase. A **diferencia** de una variable de instancia, su valor es común en toda la jerarquía de clases.

En Spacewar!, una variable de clase `Vertices` tendrá el mismo diagrama para una nave espacial y un torpedo. Esto no es lo que queremos.

```
SpaceShip>>vertices
   ↑ `{0@-15 . -10@15 . 0@10 . 10@15}`
```

7.3.5 Dibujar simplificado

Usar una variable de clase en el diseño actual del juego es un poco excesivo. Fue una excusa para presentar el concepto de variables de clase. Si el juego incluyera un editor en el que el usuario rediseñara los diagramas de la nave y los torpedos, tendría sentido guardar los vértices en una variable. Pero nuestros vértices de los diagramas de la nave espacial y los torpedos son constantes. No los modificamos. Al igual que hicimos con la masa de la nave espacial –Ejemplo 3.16–, podemos utilizar un método que devuelva una colección, rodeada de llaves para mejorar la eficiencia.

```
SpaceShip>>vertices
   ↑ `{0@-15 . -10@15 . 0@10 . 10@15}`

Torpedo>>vertices
   ↑ `{0@-4 . -2@4 . 2@4}`
```

Ejemplo 7.10: Vértices devueltos por un método de instancia

Luego, en los métodos de dibujo, reemplazamos `self class vertices` por `self vertices`.

7.3.6 Revisar colisiones

En Ejemplo 4.23, tenemos un enfoque muy ingenuo para la colisión entre la estrella central y las naves, basado en la distancia entre los morphs. Era muy impreciso.

En los juegos de bitmaps, una forma clásica de detectar colisiones es observar las intersecciones de los rectángulos que rodean los objetos gráficos. El mensaje `#displayBounds` enviado a un morph responde a sus límites en la pantalla, un rectángulo que abarca el morph dada su rotación y escala.

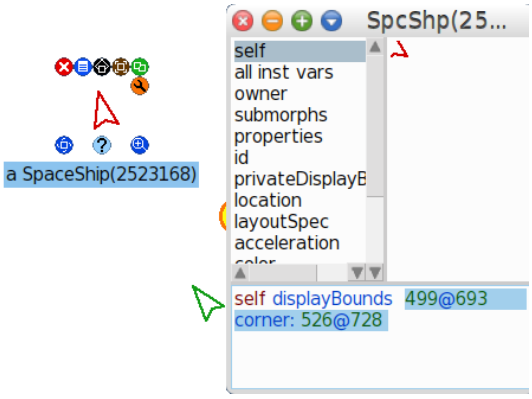

Figura 7.13: Los límites de visualización de una nave espacial

Al examinar la clase `Rectangle`, se aprende que el mensaje `#intersects:` nos indica si dos rectángulos se superponen. Esto es lo que necesitamos para una detección de colisiones más precisa entre la estrella central y las naves espaciales:

```
SpaceWar>>collisionsShipsStar
   ships do: [:aShip |
      (aShip displayBounds intersects: centralStar displayBounds) ifTrue: [
         aShip flashWith: Color red.
         self teleport: aShip]]
```

Ejemplo 7.11: Colisión (superposición de rectángulos) entre las naves y el Sol.

Sin embargo, utilizamos el framework VectorGraphics, que conoce la forma de cada morph que está renderizando. Utiliza este conocimiento para la detección de colisiones con precisión de píxeles. Reescribimos nuestro método de detección de forma más sencilla con el mensaje `#collides:`:

```
SpaceWar>>collisionsShipsStar
   ships do: [:aShip |
      (aShip collides: centralStar) ifTrue: [
         aShip flashWith: Color red.
         self teleport: aShip]]
```

Ejemplo 7.12: Colisión (precisión de píxel) entre las naves y el Sol

Reescribe los tres métodos de detección de colisiones entre naves espaciales, torpedos y la estrella central.

Ejercicio 7.9: Detección precisa de colisiones

8 Eventos

> Cuando leía cuentos de hadas, pensaba que ese tipo de cosas nunca sucedían,
> ¡y ahora aquí estoy, en medio de uno!
>
> —*Lewis Carroll, Alicia en el país de las maravillas*

¿Qué acaba de pasar?

Anteriormente hemos hablado sobre el control de flujo, es decir, cómo se toman decisiones sobre qué hacer al realizar cálculos. Hemos hablado de ello como si todos los recursos de procesamiento del ordenador estuvieran dedicados a esta tarea. Pero no es así.

Las computadoras pueden ser rápidas en algunos cálculos, pero sólo lo son hasta cierto punto, y cuando hay muchas cosas que hacer, se comparten haciendo turnos.

Así que, además de *hacer esto y luego aquello*, ocurren *eventos*.

Además, un ordenador puede ser tan rápido que, literalmente, no tenga *nada* que hacer. ¿Qué hace entonces? Cuando un procesador entra en *modo de suspensión*, ¿cómo llamamos su atención?

Estás leyendo el capítulo adecuado para saberlo.

8.1 Eventos del sistema

El hardware moderno integrado *System On a Chip* (*SOC*) tiene muchos circuitos que están activos al mismo tiempo. Por lo tanto, un tipo de evento es detectar algo que está sucediendo en el mundo. La clase `EventSensor` gestiona las pulsaciones del teclado y las *interrupciones de hardware* del ratón, traduciendo entre las señales del hardware y los objetos de eventos del software.

Básicamente, un morph *levanta la mano* y dice qué eventos, si los hay, le interesan recibir. A continuación, implementa métodos para obtener los objetos de evento que contienen la información de los eventos capturados. En Cuis-Smalltalk, la clase `MorphicEvent` y sus subclases representan la diversidad de eventos del sistema.

```
MorphicEvent
  DropEvent
  DropFilesEvent
  UserInputEvent
    KeyboardEvent
    MouseEvent
      MouseButtonEvent
      MouseMoveEvent
      MouseScrollEvent
    WindowEvent
```

A medida que se generan los eventos `MouseMoveEvent`s, el `HandMorph` ajusta su posición en la pantalla. Cuando llegan eventos del ratón y pulsaciones de teclas, el `HandMorph` coordina el «envío» de eventos al morph adecuado bajo el cursor, además de mostrar información sobre herramientas y transportar morphs en tránsito durante las operaciones de arrastre.

Como vimos en el capítulo anterior con `ColorClickEllipse`, cualquier morph puede anular los métodos predeterminados de `Morph` para afirmar que gestiona diversos eventos

de usuario y los métodos que toman los objetos de evento asociados cuando llegan los eventos.

Básicamente, se generan eventos de entrada del usuario, un `HandMorph` refleja cualquier movimiento del cursor, los morphs reaccionan a los eventos, cada tarea de larga duración obtiene un intervalo de tiempo y avanza, cualquier cambio en la visualización se actualiza en la pantalla y se pasa al siguiente paso. El tiempo avanza un paso.

Esto ocurre una y otra vez, manteniendo la ilusión del malabarista de que todas las pelotas en el aire se mueven al mismo tiempo. Por debajo, cada una de las pelotas se mueve ligeramente, en secuencia.

8.2 Mecanismo general

En Sección 6.4 [Una breve introducción a los inspectores], página 77, explicamos cómo configurar las propiedades de una instancia morph individual para gestionar un evento específico. En este caso, una propiedad informaba a Cuis-Smalltalk de que nos interesaba un evento determinado (`#handlesMouseDown`), mientras que una segunda propiedad definía el comportamiento con un bloque de código que se ejecutaba cada vez que se producía este evento.

Alternativamente, para gestionar eventos en una clase morph determinada, definimos el comportamiento con métodos de instancia. En la clase `Morph`, observa las categorías de métodos `event` y `event handling testing`.

La categoría de métodos `event handling testing` (prueba de gestión de eventos) enumera los métodos que devuelven un valor booleano para indicar si se debe notificar a la instancia sobre determinados eventos. Echemos un vistazo a uno de estos métodos, `handlesMouseDown:`, cuyo comentario merece la pena leer:

```
Morph>>handlesMouseDown: aMouseButtonEvent
"Do I want to receive mouseButton messages ?
- #mouseButton1Down:localPosition:
- #mouseButton1Up:localPosition:
- #mouseButton2Down:localPosition:
- #mouseButton2Up:localPosition:
- #mouseButton3Down:localPosition:
- #mouseButton3Up:localPosition:
- #mouseMove:localPosition:
- #mouseButton2Activity
NOTE: The default response is false. Subclasses that implement these
messages directly should override this one to return true.
Implementors could query the argument, and only answer true for (for
example) button 2 up only."
"Use a property test to allow individual instances to dynamically
specify this."

   ↑ self hasProperty: #'handlesMouseDown:'
```

Tal y como se define por defecto, este método y los demás controladores comprueban si una instancia ha definido una propiedad con el mismo nombre que el método estándar. De este modo, cada instancia individual puede añadir su propio comportamiento.

En una clase morph en la que queremos que *todas* las instancias gestionen los eventos de pulsación del ratón, simplemente sobrescribimos el método adecuado para que devuelva `true`:

```
MyMorph>>>>handlesMouseDown: aMouseButtonEvent
   ↑ true
```

Ahora, en la categoría del método `events` de la clase `Morph`, encontramos los controladores enumerados en el comentario anterior. Una `ScrollBar`, un tipo de Morph que representa el control de posición de una lista, desplaza el contenido de la lista cuando se pulsa el botón 1 del ratón:

```
ScrollBar>>mouseButton1Down: aMouseBtnEvent position: eventPosition
"Update visual feedback"
   self setNextDirectionFromEvent: aMouseBtnEvent.
   self scrollByPage
```

Para descubrir otros eventos disponibles para tu morph, explora con el Browser del sistema como se ha descrito anteriormente.

8.3 Eventos de Spacewar!

Obviamente, nuestro juego Spacewar! gestiona eventos. En primer lugar, queremos controlar las naves con el teclado. En segundo lugar, queremos que el juego se pause cuando el cursor del ratón salga del juego y se reanude cuando vuelva a entrar.

En nuestro diseño, un morph único, la instancia `SpaceWar`, modela el juego. Por lo tanto, queremos que esta instancia gestione los eventos descritos anteriormente.

8.3.1 Eventos de ratón

El cursor del ratón entra en la zona de juego

Queremos capturar los eventos cuando el cursor del ratón se mueve sobre nuestro morph `SpaceWar`.

¿Qué método debería devolver verdadero para que el juego reciba una notificación con mensajes específicos cuando el cursor del ratón entra o sale? ¿En qué clase deberíamos implementar este método?

Ejercicio 8.1: Recibir notificaciones del evento de desplazamiento del ratón

Una vez que dejamos claro que queremos que el juego reciba eventos de desplazamiento del ratón, debemos configurar el comportamiento en consecuencia con métodos específicos.

Cada vez que el cursor del ratón entra en el juego, queremos:

- **Obtener el foco del teclado.** Sigue al cursor del ratón: la entrada del teclado va al morph situado bajo el cursor del ratón. En Cuis-Smalltalk, el cursor del ratón se modela como una instancia `HandMorph`, un objeto de evento (véase la jerarquía de clases de eventos al principio de este capítulo). Se pregunta a un objeto de evento por su manejo con el mensaje `#hand`. En definitiva, queremos que el foco del teclado se dirija hacia nuestro juego cuando entra el ratón:

  ```
  event hand newKeyboardFocus: self
  ```

- **Reanudar el juego.** La actualización continua del juego se realiza mediante un mecanismo de pasos dedicado, que se explicará en el siguiente capítulo. El juego solo tiene que solicitar la reanudación de los pasos:

```
self startStepping
```

¿Qué mensaje se envía al juego para notificar que el cursor del ratón entra en el área de juego? ¿Cómo se debe escribir el método correspondiente?

Ejercicio 8.2: Gestionar el evento de entrada del ratón

El cursor del ratón deja la zona de juego

También queremos que se nos informe cuando el cursor del ratón abandone nuestro morph `SpaceWar`. Gracias al trabajo realizado en Ejercicio 8.1, ya hemos informado a Cuis-Smalltalk de que queremos que se nos notifique el movimiento del ratón sobre el juego. Sin embargo, necesitamos programar el comportamiento cuando el cursor del ratón abandona el juego:

- **Liberar el foco del teclado.** Le indicamos a Cuis-Smalltalk que el juego no quiere el foco del teclado:

```
event hand releaseKeyboardFocus: self
```

- **Pausar el juego.** Detendremos la actualización (*stepping*) continua del juego:

```
self stopStepping
```

¿Qué mensaje se envía al juego para notificar que el cursor del ratón ha salido del área de juego? ¿Cómo debemos sobrescribir el método?

Ejercicio 8.3: Manejar el evento de salida del ratón

En la interfaz gráfica de usuario, a menudo se utiliza un efecto visual para informar al usuario de que el foco del teclado ha cambiado. En Spacewar! cambiamos el fondo del juego en función del estado del foco del teclado.

En la Figura 8.1, a la izquierda el foco del teclado está en el juego; a la derecha el foco del teclado no está en el juego, está en pausa y podemos ver a través.

Figura 8.1: Efecto Spacewar! dependiendo del foco del teclado

En el marco Morph, el mensaje `#keyboardFocusChange:` se envía al morph que pierde o gana el foco del teclado, su parámetro es un booleano. Por lo tanto, implementamos el comportamiento Figura 8.1 en el método correspondiente `keyboardFocusChange` de `SpaceWar`:

```
SpaceWar>>keyboardFocusChange: gotFocus
   gotFocus
      ifTrue: [color ← self defaultColor]
      ifFalse: [color ← self defaultColor alpha: 0.5].
   self redrawNeeded
```

Ejemplo 8.1: Efecto del foco del teclado en Spacewar!

8.3.2 Eventos de teclado

Para controlar las naves espaciales, utilizamos el teclado. Por lo tanto, queremos que el juego sea notificado de los eventos del teclado.

Averigua qué método debería devolver verdadero para que el juego reciba la notificación del evento del teclado.

Ejercicio 8.4: Recibir notificaciones de eventos del teclado

Podemos decidir que se nos notifique el evento de pulsación o liberación de tecla, así como el evento de pulsación y liberación de tecla (*pulso de tecla*). Siempre que nuestro morph `SpaceWar` responda verdadero al mensaje `#handlesKeyboard`, recibirá los mensajes `#keyUp:`, `#keyDown:` y `#keyStroke:`. Por defecto, los métodos coincidentes de la clase `Morph` no hacen nada.

El argumento de estos mensajes es un objeto `KeyboardEvent` al que, entre otras cosas, se le puede preguntar el `#keyCharacter` de la tecla pulsada o comprobar algunas teclas

especiales, como las flechas del teclado. La nave del primer jugador, –la verde–, se controla con las flechas del teclado cuando se pulsan:

```
SpaceWar>>keyStroke: event
   event isArrowUp ifTrue: [↑ ships first push].
   event isArrowRight ifTrue: [↑ ships first right].
   event isArrowLeft ifTrue: [↑ ships first left].
   event isArrowDown ifTrue: [↑ ships first fireTorpedo].
   ...
```

Ejemplo 8.2: Teclas para controlar la nave del primer jugador

Para controlar la nave del segundo jugador, utilizamos otra disposición clásica en los juegos controlados con teclado QWERTY: WASD.[1].

Añade el código adicional al Ejemplo 8.2 para controlar la nave del segundo jugador con las teclas WASD. Como recordatorio, en Smalltalk el código de carácter para q se puede escribir como `$q`.

Ejercicio 8.5: Teclas para controlar la nave del segundo jugador

[1] `https://en.wikipedia.org/wiki/Arrow_keys#WASD_keys`

9 Gestión del código

> El cambio es fácil, excepto por la parte que cambia.
> —*Alan Kay*

En cuanto al código fuente, Cuis-Smalltalk incluye varias herramientas para manipularlo: la imagen, el *change log* (registro de cambios), el *change set* (conjunto de cambios) y el sistema de paquetes. Te damos un recorrido por estos mecanismos y luego te explicamos cómo debes gestionar el código de una aplicación escrita con Cuis-Smalltalk.

9.1 La imagen

Ya hemos escrito sobre la *imagen* de Cuis-Smalltalk (Véase Sección 1.2 [Instalar y configurar Cuis-Smalltalk], página 6). Al guardar el estado de la máquina virtual en el archivo de imagen, cada uno de los cambios realizados en el entorno quedará reflejado en la imagen guardada: esto incluye las ventanas del entorno, el contenido del espacio de trabajo, las clases y métodos recién escritos, las instancias existentes, incluidos los cambios visuales, una sesión de depuración con un navegador del sistema, un inspector, etc.

En cualquier momento, el usuario puede guardar la imagen con... Menú World → `Save...` (Guardar). Otra posibilidad es `Save as...` (Guardar como...), que guarda la imagen con un nombre alternativo proporcionado por el usuario.

Guardar la imagen es el método más fácil y sencillo para guardar tu propio código. Pero no podemos llamar a eso gestión de código, ya que tu código no se guarda en un archivo específico, sino que se mezcla con otro código en una imagen. Además, no será práctico compartir tu trabajo con otros, por ejemplo, a través de un sistema de control de versiones.

Por diversas razones, una imagen puede estar en un estado confuso: la máquina virtual puede bloquearse al ejecutarla, el sistema de archivos puede ser inestable o el entorno puede estar bloqueado. Esto supone un inconveniente cuando se utiliza la imagen como único repositorio de código fuente. El resultado final podría ser la pérdida de tu trabajo.

Si ha perdido código debido a un fallo de la máquina virtual, existe una solución para recuperar la edición perdida: el *Change Log*.

9.2 El Change Log

Cuis-Smalltalk registra cualquier acción que se produzca en el entorno: el código que editas en el navegador del sistema, el código que ejecutas en un espacio de trabajo. Por lo tanto, en caso de que Cuis-Smalltalk se bloquee, puedes restaurar los cambios no guardados cuando vuelvas a iniciar la misma imagen de Cuis-Smalltalk. Exploremos esta función con un ejemplo sencillo.

En una instalación nueva de Cuis-Smalltalk, primero configura la preferencia adecuada ejecutando en un Workspace: `Preferences at: #checkLostChangesOnStartUp put: true`. Ahora crea una nueva categoría de clase llamada `TheCuisBook` y dentro la clase `TheBook`:

- Sobre el panel de categorías de clases del Browser del sistema (en el extremo izquierdo), haz clic derecho → `add items... (a)` y escribe `TheCuisBook`.
- Selecciona esta nueva categoría de clase y crea la clase `TheBook` como un tipo de `Object:` selecciona la categoría `TheCuisBook` y, a continuación, en el código fuente

que aparece a continuación, edita la plantilla de clase para sustituir `#NameOfClass` por `#TheBook` y, a continuación, guarda la definición de clase con *`Ctrl-s`*.

Abre un Workspace y escribe el siguiente código:

```
| myBook |
myBook ← TheBook new
```

Cuis-Smalltalk no guarda el código que escribes en el espacio de trabajo, sino el código que ejecutas. Ejecutemos este código: *`Ctrl-a`* y luego *`Ctrl-p`*, el espacio de trabajo imprime el resultado: `a TheBook`, una instancia de una clase `TheBook`.

Ahora cierra Cuis-Smalltalk de forma abrupta. En GNU/Linux, puedes usar el comando `xkill` para cerrar Cuis-Smalltalk haciendo clic en su ventana.

Ahora vuelve a iniciar Cuis-Smalltalk y te informará inmediatamente de que hay cambios sin guardar:

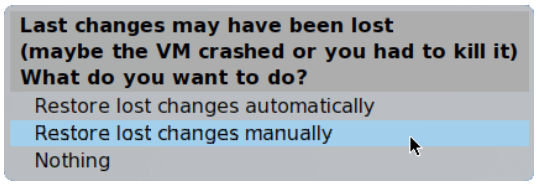

Figura 9.1: Cuis-Smalltalk informa sobre los cambios perdidos

Desde ahí tienes tres opciones:

- **Restaurar automáticamente los cambios perdidos.** Cuis-Smalltalk aplicará todos los cambios: nuevas definiciones de clases, nuevos métodos; definiciones de clases editadas y código fuente de métodos; código ejecutado (en espacios de trabajo o en cualquier lugar donde se pueda ejecutar código). A menudo, esto no es realmente lo que se desea hacer, especialmente en el caso del código ejecutado.

- **Restaurar manualmente los cambios perdidos.** En la ventana `Lost changes` (Cambios perdidos) que aparece a continuación, se muestran los cambios no guardados, uno por línea, en orden cronológico, con los más antiguos en la parte superior de la lista. Seleccione cada cambio (línea) que desee restaurar y, a continuación, aplique su selección con el botón `file in selections` (archivo en selecciones).

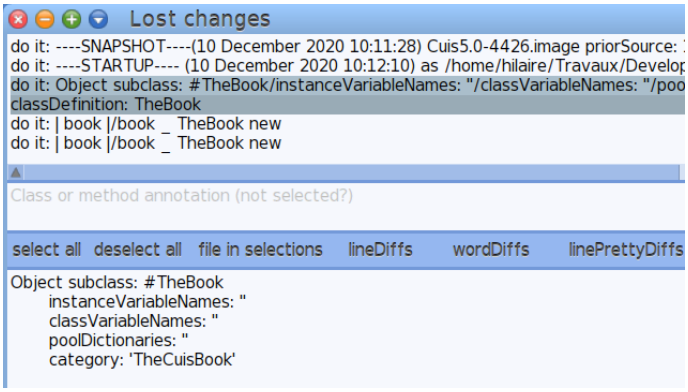

Figura 9.2: Selecciona manualmente los cambios que desea guardar en el archivo

Para introducir los cambios relacionados con la creación de la clase `TheBook`, pero no el código ejecutado en el espacio de trabajo, seleccione las dos líneas relacionadas con la definición de la clase.

El menú contextual (clic derecho del ratón) de la ventana `Lost changes` ofrece muchas opciones para filtrar los cambios. Es útil cuando el lote de cambios perdidos es importante.

- **Nada.** No se restauran los cambios. Ten en cuenta que los cambios no guardados no se descartan hasta que guardes la imagen.

  Si cambias de opinión y deseas recuperar los cambios, ve al menú World → `Changes...` → `Recently logged Changes...`.

  El sistema te presenta una lista de instantáneas de imágenes etiquetadas con una marca de fecha. Elige la que se produjo justo antes de perder tu código, probablemente la que aparece en la parte superior de la lista. A continuación, en la ventana `Recent changes` (Cambios recientes), procede como se ha descrito anteriormente para seleccionar los cambios que deseas restaurar.

9.3 El Change Set

En una instalación nueva de Cuis-Smalltalk, cada código que editas en el Browser del Sistema se registra en un *Change Set*.

Para examinar un //change set// (conjunto de cambios), utiliza la herramienta denominada *Change Sorter* (clasificador de cambios): ... Menú World (Mundo) → `Changes...` (Cambios...) → `Change Sorter...` (Clasificador de cambios...)

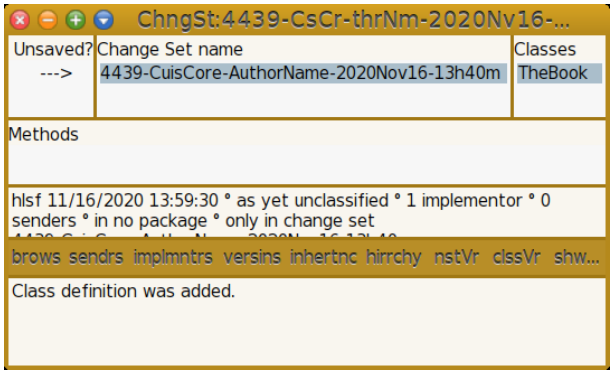

Figura 9.3: El *Change Sorter*, edición de clase

La clase `TheBook` que añadimos a Cuis-Smalltalk en la sección anterior es un cambio realizado en el núcleo del sistema. De forma predeterminada, se registra en un conjunto de cambios creado automáticamente por el sistema. En Figura 9.3 en la parte superior derecha, observa la clase `TheBook`, que pertenece a un conjunto de cambios denominados `4439-CuisCore-AuthorName-2020Nov16-13h40m`. En el panel izquierdo, cada conjunto de cambios no guardado está marcado con una `--->` . Aquí nos indica que el cambio no se ha guardado en el disco. Para guardar el conjunto de cambios, basta con utilizar su menú contextual y seleccionar una de las entradas `file out`. El conjunto de cambios se guardará junto con la imagen de Cuis-Smalltalk bajo su nombre de sistema, sustituyendo `AuthorName` por el nombre real del autor.

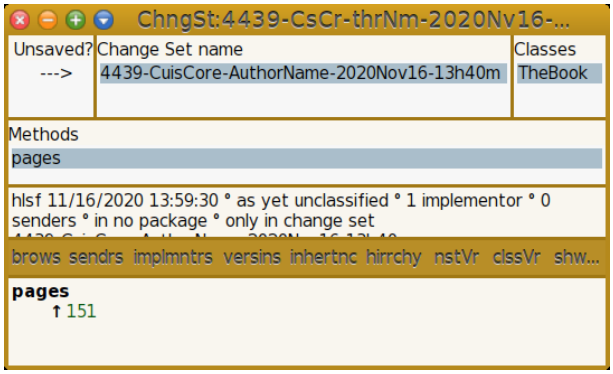

Figura 9.4: El *Change Sorter*, editar método

Fíjate en la Figura 9.4, después de añadir el método `pages` a la clase `TheBook`, el panel central muestra los métodos añadidos o modificados. Cuando se selecciona un método, su código fuente se muestra en el panel inferior.

Supongamos que guardamos el conjunto de cambios: entradas `File out` en el menú de la herramienta de clasificación de cambios. Esto crea un nuevo archivo `4451-CuisCore-HilaireFernandes-2020Nov14-21h08m-hlsf.001.cs.st` junto al archivo de imagen Cuis-Smalltalk:

```
From Cuis 5.0 [latest update: #4450] on 18 November 2020 at 9:05:09 am'!
!classDefinition: #TheBook category: 'TheCuisBook'!
Object subclass: #TheBook
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'TheCuisBook'!

!TheBook methodsFor: 'as yet unclassified' stamp: 'hlsf 11/18/2020 09:04:58'!
pages
   ↑ 151! !
```

Ejemplo 9.1: Contenidos de Change set

Para cargar este conjunto de cambios en una nueva imagen, utilice la herramienta *File List* (Lista de archivos)...Menú World (Mundo) → `Open` (Abrir) → `File List...` (Lista de archivos...) Busca la carpeta que contiene el archivo del conjunto de cambios que deseas cargar y selecciónalo. A partir de ahí, tienes tres opciones para manipularlo.

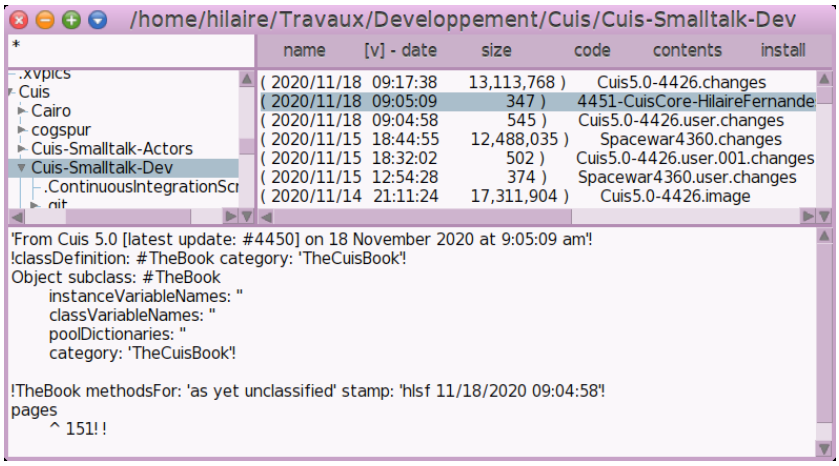

Figura 9.5: La herramienta File List, para instalar un conjunto de cambios y más

- `code`. Abre una especie de Browser del Sistema limitado al código del archivo del conjunto de cambios. Es una herramienta muy útil para leer y aprender el código del conjunto de cambios.
- `contents`. Se abre una herramienta denominada *Change List* (Lista de cambios) para revisar las modificaciones que este conjunto de cambios producirá en la imagen una vez instalado. También te permite seleccionar los cambios individuales que deseas instalar y descartar. Cada línea que seleccionas representa una clase o una adición/modificación de método. Una vez seleccionado el código que deseas instalar, pulsa el botón `file in selections` (archivo en selecciones) para continuar con la instalación.

  Supongamos que un compañero desarrollador modifica la clase `TheBook`, añade una variable de instancia `pages` y ajusta los métodos `pages` en consecuencia. Guarda los cambios y comparte el archivo contigo. Fíjate en la Figura 9.6 para ver cómo revisarás sus cambios con la herramienta *Change List*. Nuestro código que se eliminará de la imagen se muestra con trazos rojos, y su código que se instalará se muestra en verde.

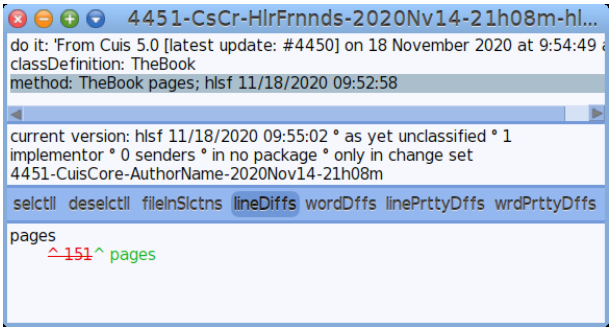

Figura 9.6: La herramienta *Change List* para revisar las modificaciones realizadas en la imagen

- `install`. Simplemente instala el conjunto completo de cambios sin interactividad.

Los desarrolladores de Cuis-Smalltalk utilizan el método de gestión del código fuente basado en conjuntos de cambios para trabajar en su imagen central. Cuando se desea escribir una aplicación, una herramienta específica o incluso un conjunto de clases que abarquen un dominio concreto, lo ideal es utilizar otra cosa para gestionar el código: un paquete.

9.4 El paquete

Un paquete puede contener un conjunto de clases que forman parte de la misma categoría de clases.

Guardemos nuestra categoría `Morphic-Learning` como un paquete utilizando nuestro navegador de paquetes instalados.

...Menú World → `Open...` → `Installed Packages`...

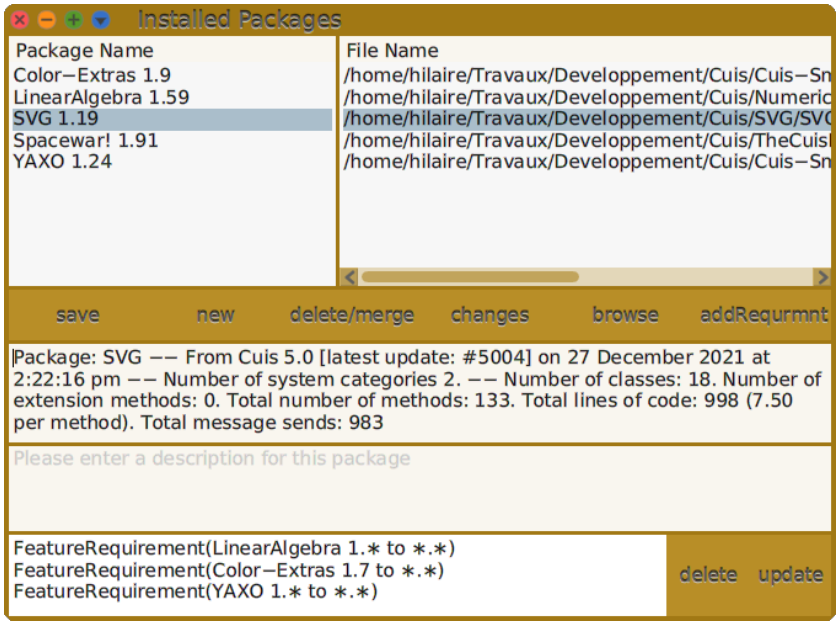

Figura 9.7: Browser de paquetes instalados

Fíjate en que antes hemos llamado a `Feature require: 'SVG'`. Al mirar los nombres de los paquetes, podemos observar varias cosas interesantes:

- Cada paquete tiene una versión; `SVG` tiene la versión 1, revisión 19.

- Hay paquetes llamados `YAXO`, `Color-Extras` y `LinearAlgebra` que nunca hemos solicitado.

- Si observas el panel inferior, verás que esos paquetes son **requeridos** por el paquete `SVG`.

Esto es importante. Cuando se requiere una función empaquetada, es posible que se especifique que también se necesitan otros paquetes para que funcione correctamente y, de hecho, que se especifique que dichos paquetes deben tener un nivel de versión específico.

Esta es la clave para poder componer de forma segura paquetes que tienen requisitos.

Vale, hagamos clic en el botón `New` y escribamos `Morphic-Learning` en el cuadro de diálogo. Esto da como resultado un nuevo paquete con el mismo nombre que nuestra categoría `Morphic-Learning`. Ten en cuenta que se trata de la versión 1, revisión 0 (1.0 a la derecha) y que el paquete aún no se ha guardado.

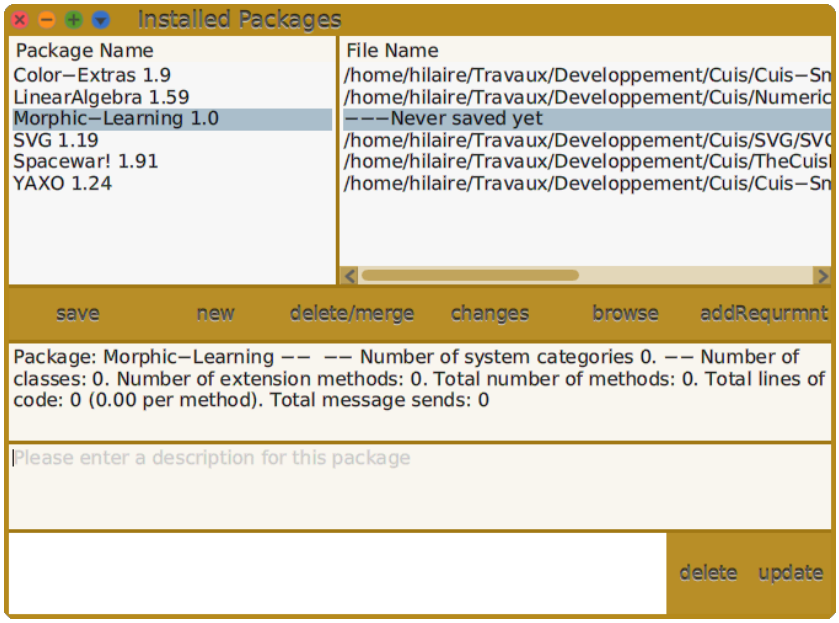

Figura 9.8: Nuevo paquete – `Morphic-Learning`

Supongamos que en nuestro morph de reloj queremos utilizar colores nombrados como en el paquete `Color-Extra`. Para poder cargar nuestro paquete `Morphic-Learning`, que hace uso de esto, debemos seleccionar nuestro nuevo paquete y hacer clic en el botón `add Requirement` (añadir requisito) situado en el centro, a la derecha.

Esto muestra una lista de paquetes cargados entre los que elegir.

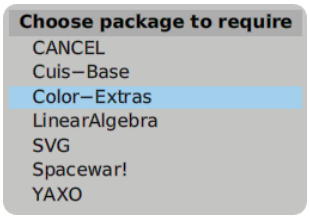

Figura 9.9: Seleccionar el paquete (o la versión base de Cuis) que se necesita

Ahora, cuando guardamos (`save`) nuestro paquete, vemos la ruta donde se ha creado el archivo del paquete. Ahora podemos enviar este archivo por correo electrónico de forma segura, registrarlo en un sistema de control de versiones y hacer una copia en nuestra memoria USB de respaldo.

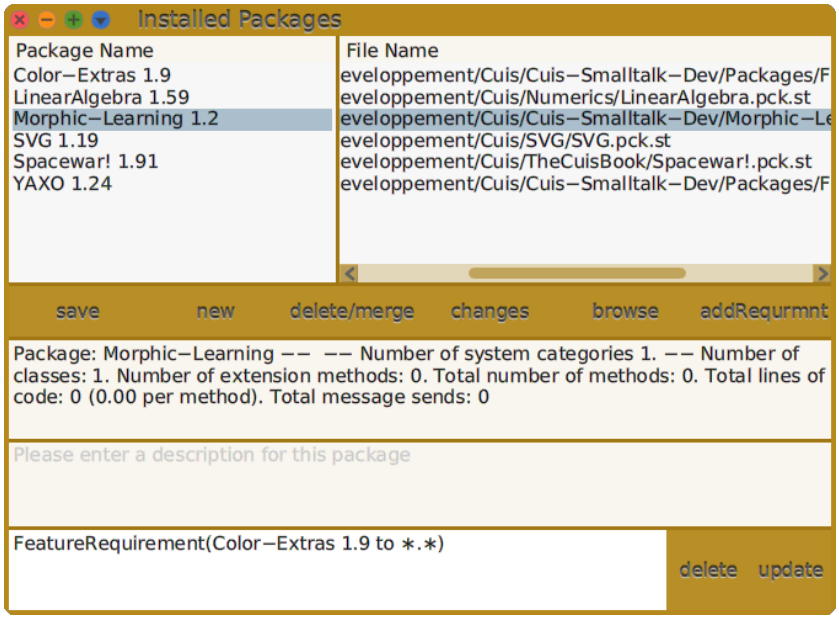

Figura 9.10: Paquete guardado – `Morphic-Learning`

Como se mencionó anteriormente, los archivos de paquetes son simplemente archivos de texto con un formato especial que Cuis-Smalltalk sabe cómo cargar. Si abre un navegador `File List` y ve el archivo del paquete, verá información sobre cómo se creó el paquete, qué proporciona y qué requiere y, si rellenó el cuadro de comentarios en el navegador `Installed Packages`, una descripción.

Escribe «Morphic Toys» en el cuadro de comentarios, vuelve a guardar el paquete y (re)selecciona el paquete en una lista de archivos para ver la descripción del paquete.

Ejercicio 9.1: Describir un paquete

Si aún no lo has hecho, crea y guarda un [Paquete Spacewar!], página 23. No hay requisitos adicionales que especificar.

Ejercicio 9.2: Guardar el paquete Spacewar!

Hay otras cosas interesantes que podemos hacer con los paquetes. Podemos incluir varias categorías de clases en un solo paquete. Supongamos que queremos dividir nuestras clases CuisBook en dos categorías `TheCuisBook-Models` y `TheCuisBook-Views`. Un nuevo

paquete creado con el nombre `TheCuisBook` incluye estas dos categorías de clases; esta etiqueta es un *prefijo* para buscar las categorías coincidentes que se incluirán en el paquete.

Normalmente, un paquete viene con varias categorías para organizar las clases en dominios que coincidan. Recomendamos hacerlo así. Cuando una aplicación o un framework crece, para mantener una organización sólida, es posible que sientas la necesidad de remodelar las categorías de clases: renombrar, dividir, fusionar, etc. Siempre que mantengas el mismo prefijo en las categorías de clases y el nombre del paquete, sus clases estarán seguras en el mismo paquete. En el Browser del sistema, puedes arrastrar y soltar cualquier clase en cualquier categoría de clases para reorganizarla.

Crea un paquete `TheCuisBook` *a partir de las dos categorías de clases* `TheCuisBook-Models` *y* `TheCuisBook-Views`*. La primera contiene una clase* `TheBook` *y la segunda una clase* `TheBookMorph`*. Guarda el paquete en el disco.*

Ejercicio 9.3: Dos categorías de clases, un paquete

Imaginemos que necesitamos imprimir el número de página del índice de `TheBook` en números romanos minúsculos, tal y como hacemos con la versión impresa de este libro. El código es muy sencillo:

```
4 printStringRoman asLowercase
⇒ 'iv'
```

En lugar de invocar esta secuencia de mensajes cada vez que la necesitamos, añadimos un mensaje específico a la clase `Integer`:

```
Integer>>printStringToc
   ↑ self printStringRoman asLowercase
```

Ahora, dentro de los métodos de nuestro `TheBook`, sólo haremos cosas como:

```
../..
 aPage ← Page new.
 aPage number: 1 printStringToc.
 ../..
```

Ahora nos enfrentamos a un problema. Para satisfacer las necesidades del paquete `TheBook`, ampliamos la clase `Integer` con un método `printStringToc`, sin embargo, esta adición al método forma parte del sistema central de Cuis-Smalltalk y su conjunto de cambios predeterminados asociado. Véase Figura 9.11, la herramienta `Change Sorter` lo muestra exactamente.

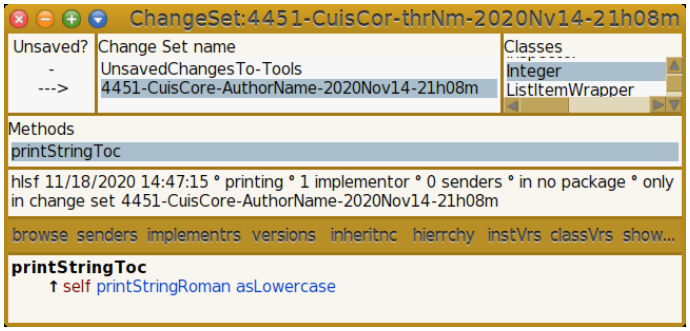

Figura 9.11: Change Sorter, método complementario al núcleo

Por lo tanto, al guardar nuestro paquete `TheBook`, este método no se incluye y se pierde al salir de Cuis-Smalltalk. Para incluirlo en nuestro paquete, lo clasificamos en una categoría de métodos con el prefijo `*TheCuisBook`. `*TheCuisBook-printing` es un buen candidato. En el panel de métodos del Browser del sistema, sobre `printStringToc`, haz ...Menú contextual → `more...` → `change category...` y escribe `*TheCuisBook-printing`.

Ahora el `Change Sorter` escribe sobre `Integer>>printStringToc`: *Method was moved to some other package* (método movido a algún otro paquete). Las herramientas de `Installed Packages` nos dicen ahora que hay una extensión, utiliza su botón `browse` para obtener una actualización de los contenidos del paquete.

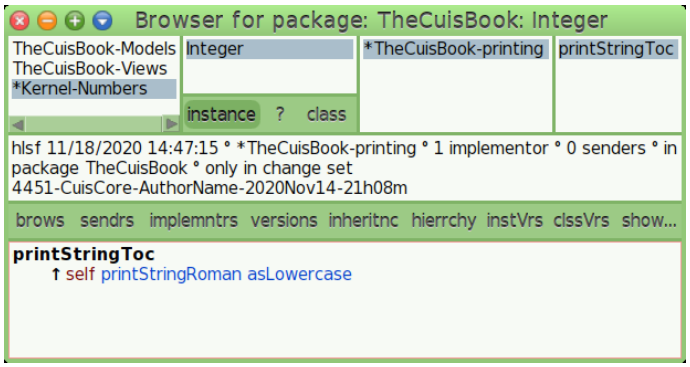

Figura 9.12: Paquete con extensión a la clase `Integer` de la categoría de clases del sistema `Kernel-Numbers`

Fíjate en cómo cada categoría –clase o método– de una extensión lleva un prefijo `*`.

Además de añadir un *requisito* de precarga de paquetes, también puedes seleccionar un requisito y borrar (`delete`) o actualizar (`update`) utilizando los botones de la parte inferior derecha. A veces, un paquete cambia y su código depende de él, por lo que tiene que cambiar su código para adaptarse. Cuando esto ocurre, es importante asegurarse de que se requiere la versión más reciente y modificada. Al seleccionar un requisito y pulsar `update`, se actualizará el requisito para utilizar la última versión del paquete cargado.

9.5 Flujo de trabajo diario

Para nuestro juego Spacewar!, creamos un archivo de paquete dedicado `Spacewar!.pck.st`. Esta es la forma de proceder al escribir un paquete externo: definir un paquete dedicado y, de vez en cuando, guardar el trabajo con el botón `save` de la herramienta Installed Packages (véase Véase Figura 2.3).

Cuis-Smalltalk utiliza GitHub para alojar, versionar y comparar su desarrollo principal, así como para gestionar un conjunto de paquetes externos (es decir, código que se mantiene de forma independiente y fuera de Cuis-Smalltalk, pero que está estrechamente relacionado con él).

Los archivos del paquete son archivos de texto simples, codificados para el alfabeto latino (ISO 8859-15) y gestionados sin problemas por GitHub. Cuis-Smalltalk utiliza la convención de salto de línea LF (código ASCII 10), tal y como prefiere GitHub. Esto permite a Git/GitHub comparar versiones y fusionar ramas.

Se utilizan repositorios GitHub independientes para los proyectos, es decir, paquetes o conjuntos de paquetes estrechamente relacionados que siempre se cargan y mantienen juntos como un todo.

Tu flujo de trabajo diario con Cuis-Smalltalk para desarrollar un paquete externo será similar al siguiente:

1. Comienza con una imagen Cuis estándar y nueva. No guardes nunca la imagen.
2. Configura tu sistema de control de versiones preferido para gestionar tus paquetes externos. Se recomienda utilizar un repositorio GitHub cuyo nombre comience por «Cuis-Smalltalk-», para que sea fácil de encontrar. Pero, aparte de esta consideración, puedes utilizar cualquier otro sistema de control de versiones.
3. Instala los paquetes necesarios desde los repositorios Git de Cuis-Smalltalk.
4. Desarrolla. Modifica y/o crea paquetes.
5. Guarda tus propios paquetes (en tus repositorios preferidos).
6. add / commit / push según corresponda a tu sistema de control de versiones
7. Cambios de archivo que no forman parte de ningún paquete. Estos se capturan automáticamente en conjuntos de cambios numerados, separados de los cambios en los paquetes.
8. Sal de la imagen. Normalmente sin guardar.


No guardar la imagen es simplemente un consejo de buenas prácticas para cuando tu objetivo principal sea crear código nuevo. Ya hemos hablado de las advertencias sobre guardar la imagen en relación con la gestión del código (Véase [La imagen], página 118). Pero, de vez en cuando, te encontrarás en la posición de un explorador cuando abras varios navegadores de código y lugares de trabajo para averiguar algo. En este caso, el estado del sistema, las ventanas abiertas y los fragmentos de código contienen el valor que te interesa, y guardar la imagen es la forma correcta de preservar el estado del sistema.[1]

9.5.1 Automatiza tu imagen

Como se describe en el flujo de trabajo diario, es una buena costumbre no guardar toda la imagen, sino solo el paquete modificado del código fuente editado. Sin embargo, cada vez que iniciamos una sesión de programación, resulta tedioso configurar la imagen para que se adapte a nuestras necesidades y gustos personales.

Las cosas que uno puede querer personalizar en la imagen son:

- Ajustes de preferencias,
- Colocación de herramientas como el Browser del sistema, el Workspace, el Transcript,
- Contenido predeterminado en el Workspace, listo para ser ejecutado.
- Instalación de paquetes.

Queremos registrar estas preferencias de imagen en un script `setUpEnvironment.st` que se ejecutará al inicio. En GNU/Linux, se le pide a Cuis-Smalltalk que ejecute un script con el `-s`, por ejemplo `squeakVM Cuis5.0.image -s setUpEnvironement.st`, donde `setUpEnvironement.st`. es un archivo que contiene código Smalltalk. Un ejemplo real podría ser:

```
../cogspur/squeak Cuis5.0-4426 -s ../scripts/setUpEnvironment.st
```

Describimos en detalle un ejemplo de script de configuración que organiza el entorno, tal y como se muestra en la Figura 9.13. Es interesante el código Smalltalk que se adentra en áreas heterogéneas de Cuis-Smalltalk, como las herramientas de desarrollo, el sistema Morph, las preferencias y el manejo de colecciones.

[1] Para obtener más información sobre la política de guardado de imágenes, lee este debate en la comunidad Cuis aquí (`https://lists.cuis.st/mailman/archives/cuis-dev/2023-July/007841.html`) y aquí (`https://lists.cuis.st/mailman/archives/cuis-dev/2023-August/007884.html`)

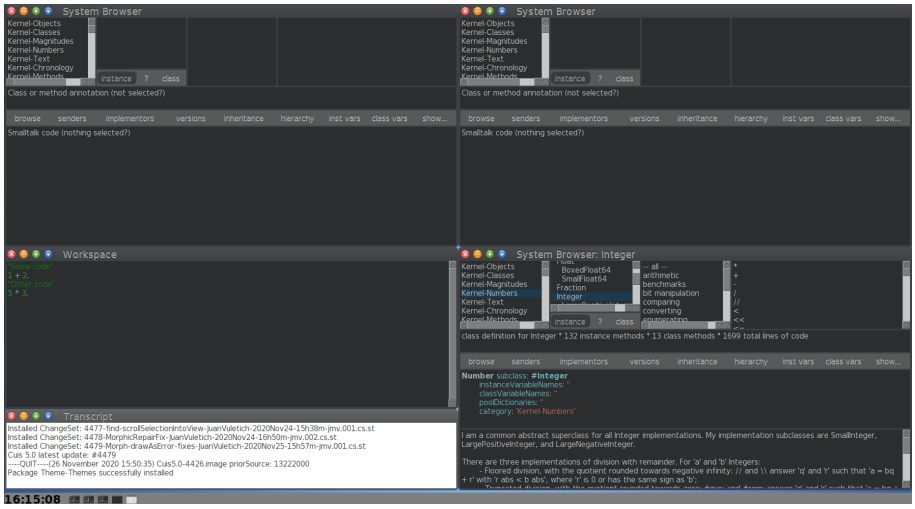

Figura 9.13: Entorno de una imagen iniciada con el script de configuración

Empecemos por cerrar las ventanas abiertas. El código se coloca en un bloque al que se le envía el mensaje `#fork` para esperar a que el entorno se inicialice correctamente.

```
| world morph area extent |
world ← UISupervisor ui.
[
    | area extent morph | "used later"
    UISupervisor whenUIinSafeState: [
        children ← world submorphs reject: [:aMorph | aMorph is: #TaskbarMorph].
        children do: [:child | child delete].
    ]
] fork
```

Todo el entorno de interfaz de usuario de Cuis-Smalltalk es un tipo de Morph, una instancia de `WorldMorph`. Sus submorphs son ventanas, menús, la barra de tareas o cualquier tipo de morph con el que el usuario pueda interactuar. Para acceder a esta instancia de `WorldMorph`, se solicita al `UISupervisor` con el mensaje `#ui`. Una vez seleccionados todos los morphs del mundo excepto la barra de tareas –en realidad, `#reject:`–, los `#delete` del mundo.

A continuación, cambiamos las preferencias. Coloca el siguiente código después de la línea anterior que elimina los hijos.

```
"Change to Dark theme"
Feature require: #'Theme-Themes'.
DarkTheme beCurrent.
"Adjust font size"
Preferences at: #defaultFontSize put: 12.
"Adjust taskbar size"
morph ← UISupervisor ui taskbar.
morph scale: 1 / 2.
Display fullScreenMode: true.
self runningWorld doOneCycleNow.
```

Necesitamos el paquete Theme-Themes; como no está instalado en la imagen predeterminada, se buscará en el disco para su instalación. En cuanto al acceso a la barra de tareas, recuerde que hemos eliminado todos los morphs excepto la barra de tareas del mundo, por lo que la barra de tareas es realmente la primera de la colección de submorphs del mundo. Por último, actualizamos la escala de la barra de tareas y solicitamos a Cuis-Smalltalk que se muestre a pantalla completa. Como hemos cambiado varias partes visuales, solicitamos un ciclo de actualización del entorno (es decir, el mundo en ejecución en la terminología de Cuis-Smalltalk).

Antes de instalar las herramientas, le preguntamos a un `RealEstateAgent` el área libre. Lamentablemente, este agente no tiene en cuenta el área ocupada por la barra de tareas, por lo que debemos ajustar su respuesta. A continuación, calculamos una cuarta parte de esta extensión de área libre (la mitad del ancho y la mitad del alto constituyen una cuarta parte del área libre total):

Coloca el siguiente código después del código anterior.

```
"Compute the available free space for windows placement"
area ← RealEstateAgent maximumUsableArea
   extendBy: 0 @ morph morphHeight negated.
extent ← area extent // 2.
```

Ahora estamos listos para instalar algunas herramientas. Los tres primeros navegadores ocupan cada uno una cuarta parte de la pantalla:

```
"Open a few System Browsers"
Browser open
   morphPosition: 0 @ 0;
   morphExtent: extent.
Browser open
   morphPosition: area width // 2 @ 0;
   morphExtent: extent.
"Open a System Browser on a specific class"
morph ← Browser open
   morphPosition: area extent // 2;
   morphExtent: extent.
morph model setClass: Integer selector: nil.
```

A continuación, en el cuarto restante libre, instalamos un espacio de trabajo que ocupa dos tercios del área y una transcripción que ocupa un tercio. El espacio de trabajo se instala con algunos contenidos predeterminados. Tenemos que hacer un pequeño truco porque, al solicitar un nuevo espacio de trabajo, Cuis-Smalltalk no responde a la instancia creada, por lo que tenemos que buscarla en las ventanas del mundo.

Coloca el siguiente código después del código anterior.

```
"Open a Workspace with some default contents"
morph ← Workspace open.
morph model actualContents: '"Some code"
1 + 2.
"Other code"
5 * 3.'.
morph
   morphPosition: 0 @ (area height // 2);
   morphExtent: extent x @ (2 / 3 * extent y).
"Open a transcript for logs"
Transcript open
```

```
morphPosition: 0 @ (area height // 2 + (2 / 3 * extent y));
morphExtent: extent x @ (1 / 3 * extent y).
```

Por supuesto, debes ajustar el argumento del mensaje `#actualContents:` a un código significativo para tu uso.

9.6 Archivo de código fuente

En este capítulo, has encontrado archivos creados por las herramientas de gestión de Cuis-Smalltalk. Algunos tienen la extensión `.pck.st` y otros la extensión `.st`. Ambos contienen código Smalltalk, pero tienen diferentes propósitos.

Los archivos con la extensión `.pck.st` son archivos de paquetes Cuis-Smalltalk. Tienen un preámbulo con información sobre el paquete, como las características que ofrece, la información sobre la versión, una descripción del paquete y los requisitos del paquete.

Los archivos de código Smalltalk se pueden instalar seleccionándolos en el explorador de la lista de archivos y haciendo clic en el botón `install`.

Cuando se instala un paquete, también se instalan sus dependencias y el paquete cargado aparece en la herramienta Package List (Lista de paquetes). Los archivos de paquetes `.pck.st` se crean utilizando la herramienta Packages List (Lista de paquetes) en Cuis-Smalltalk.

Los archivos con la extensión `.st` contienen código Smalltalk serializado: clases y métodos. Se crean cuando se archiva una categoría, clase o método del sistema Smalltalk. Puedes *serializar* código desde el navegador del sistema seleccionando una **system category**, **class** o **method** → haz clic derecho y selecciona `fileOut` en el menú contextual.

A diferencia de los archivos `.pck.st`, los archivos `.st` no contienen dependencias, descripciones ni información sobre paquetes, solo código Smalltalk. Los archivos `.st` existen desde los inicios de Smalltalk, mientras que los `.pkg.st` se añadieron con la funcionalidad de paquetes de Cuis-Smalltalk.

Además de `install`, al seleccionar un archivo de código en el explorador de File List también se añaden botones para inspeccionar el código (`code`) y tratar el contenido (`content`) del código como un Change Sorter. Al inspeccionar el código, un navegador muestra las diferencias entre el archivo y la imagen en ejecución y permite importar clases o métodos individuales con la ayuda del menú contextual.

10 Depuración y manejo de excepciones

> Principio reactivo: todos los componentes accesibles para el usuario deben poder presentarse de forma significativa para su observación y manipulación.
> —*Dan Ingalls*

Vale la pena repetir la cita anterior.

Vemos que los Morphs y los «objetos de datos» pueden presentarse para ser inspeccionados, pero el estado en tiempo de ejecución de Smalltalk también es accesible.

10.1 Inspeccionar lo inesperado

Hemos visto cómo diversas situaciones excepcionales provocan la aparición de una ventana de depuración. De hecho, las excepciones (`Exception`) también son objetos que recuerdan su contexto y pueden presentarlo. Anteriormente, hemos visto cómo generar instancias de excepción `MessageNotUnderstood` y `ZeroDivide`.

Esta es otra área en la que la mecánica real es compleja, pero las ideas básicas son sencillas.

Las instancias de excepciones, al ser objetos, también tienen clases. El `BlockClosure` tiene una categoría de métodos `exceptions` que reúne algunos métodos útiles que permiten `ensure:` limpiar o capturar y utilizar excepciones (`on:do:` y similares).

```
FileEntry>>readStreamDo: blockWithArg
  "Raise FileDoesNotExistException if not found."
  | stream result |
  stream ← self readStream.
  [ result ← blockWithArg value: stream ]
ensure: [ stream ifNotNil: [ :s | s close ]].
  ↑ result
```

Ejemplo 10.1: Asegurar que un `FileStream` está cerrado

Se crean y se avisan (*signal*) excepciones. Creemos una y veámosla.

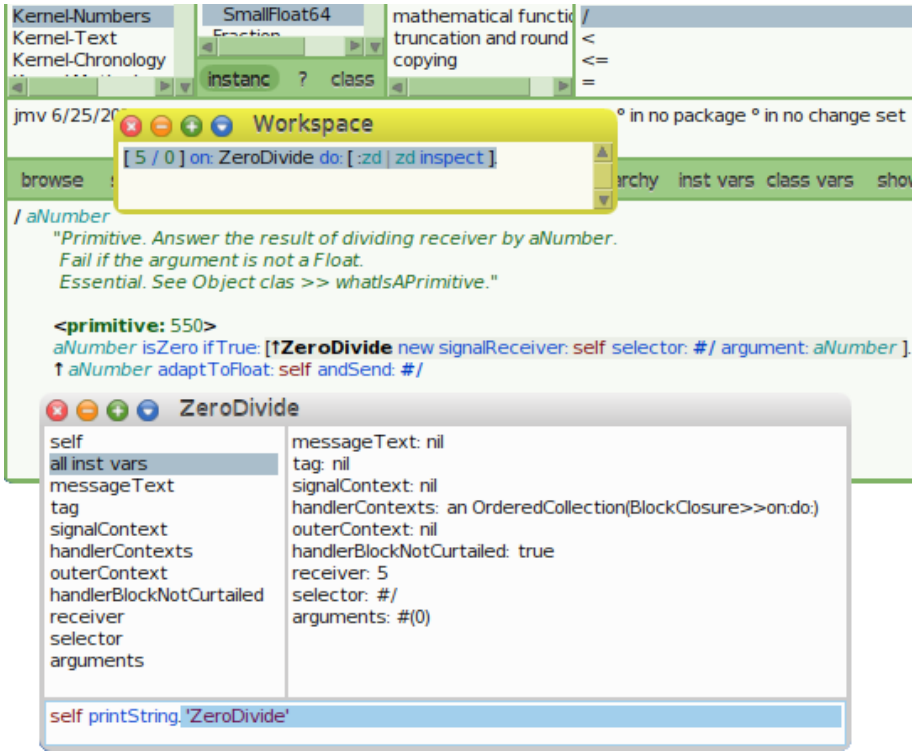

Figura 10.1: Inspección de una instancia `ZeroDivide`

Una vez más, podemos usar un Inspector en cualquier objeto, ¡y las `Exception`s no son una excepción! Ahora ya sabes cómo capturar una cuando lo necesites.

Las `Exception`s, al igual que los `MorphicEvent`s, son un cambio, una excepción, al flujo de control típico.

Anteriormente hemos mencionado la pseudovariable especial `thisContext`. La llamada de una excepción captura esto.

```
Exception>>signal

  ↑ self signalIn: thisContext
```

Ejemplo 10.2: Capturar `thisContext`

Al igual que el código Smalltalk tiene ventanas de visualización especiales que llamamos `Browser`, las `Exception` tienen un visor mejorado que llamamos `Debugger`. Veamos cómo utilizar este útil visor.

10.2 El Debugger

En primer lugar, necesitamos un ejemplo de código bastante sencillo para analizar. Escribe o copia lo siguiente en un Workspace.

```
| fileNames |
fileNames ← OrderedCollection new.
(DirectoryEntry smalltalkImageDirectory)
    childrenDo: [ :f | fileNames add: f name ].
fileNames asArray.
```

Ejemplo 10.3: Nombres de las entradas del directorio

Ahora, puedes pulsar `Ctrl-a` (*select* **A***ll*) y `Ctrl-p` (*select* **P***rint-it*) para ver el resultado.

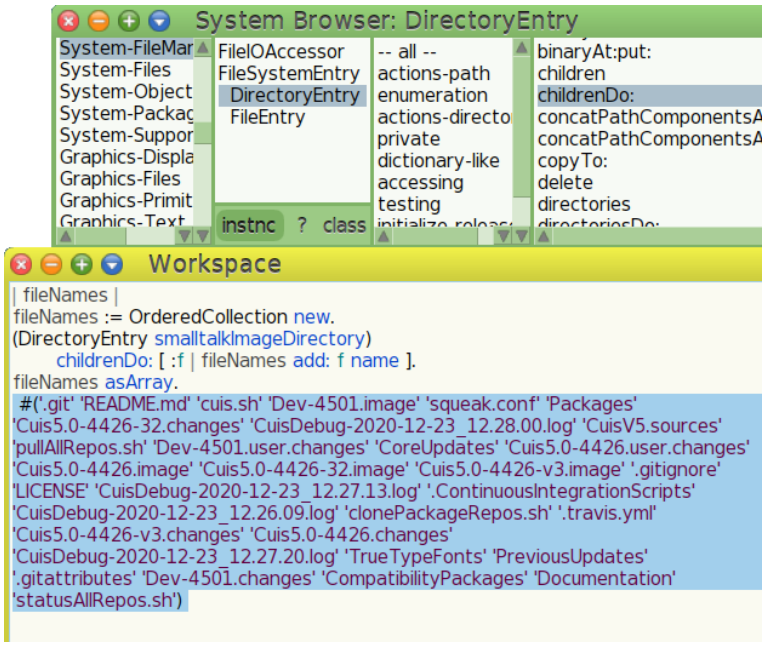

Figura 10.2: Nombres de archivos y directorios en un directorio

La clase `CharacterSequence` tiene varios nombres de categorías de métodos que comienzan por `fileman-` para convertir nombres de ruta (nombres del sistema para archivos y directorios) en objetos `FileEntry` y `DirectoryEntry`. `CharacterSequence>>asFileEntry` proporciona ejemplos.

Ahora que sabemos qué esperar, veamos paso a paso el procesamiento del código utilizando el debugger (depurador). Elimina el resultado, luego pulsa `Ctrl-a` (*select* **A***ll*) y `Ctrl-Shift-D` (**D***ebug-it*).

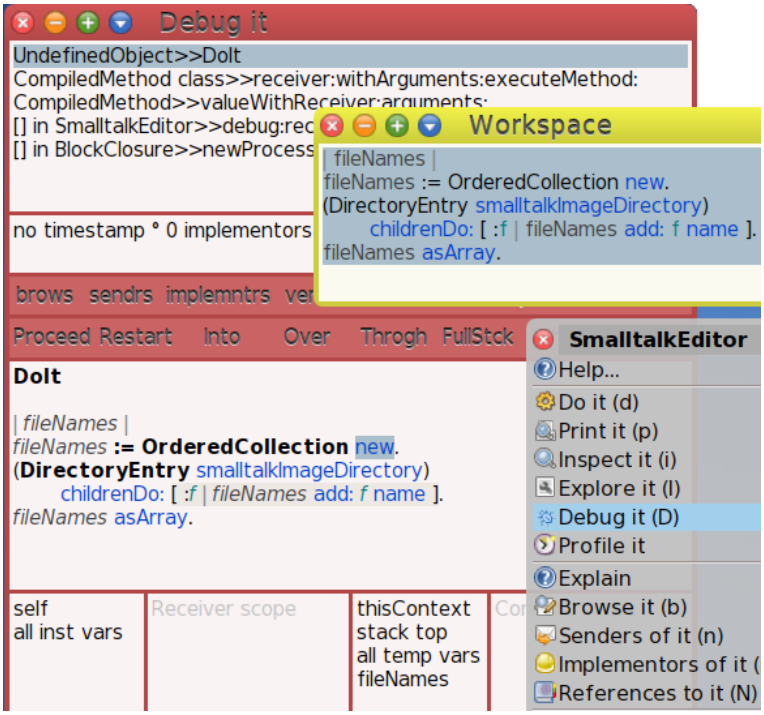

Figura 10.3: Depúralo

El panel superior del depurador muestra una vista de la *pila de ejecución* (*execution stack* para este fragmento del contexto de ejecución. La forma de entender esto, el *modelo de ejecución*, es que cada vez que un método envía un mensaje, este y su estado actual, argumentos y variables locales se colocan en una pila hasta que se recibe el resultado de ese mensaje. Si ese mensaje provoca el envío de otro mensaje, el nuevo estado se introduce en la pila. Cuando se devuelve un resultado, se extrae (*poped*) el *marco de la pila* y continúa el procesamiento. Esto funciona como una pila de bandejas en una cafetería.

Los marcos de pila se muestran para indicar el receptor y el método apilados. El objeto de enfoque, el receptor, para el marco de pila seleccionado tiene un inspector en los paneles inferiores izquierdos del depurador, en la parte inferior de la ventana.

Los dos paneles inferiores siguientes son un inspector para los argumentos y las variables locales, o *temporales*, del marco de contexto.

El área más grande muestra el código que se está procesando y resalta el siguiente mensaje que se enviará.

La pila de contextos de ejecución proporciona un historial del cálculo realizado hasta el momento. Puedes seleccionar cualquier marco, ver los valores de instancia en el receptor y ver los argumentos y las variables del método en ese punto.

Las dos filas de botones situadas encima del panel de código ofrecen vistas adicionales y control sobre cómo debe realizar el proceso la ejecución.

Botones destacados en la segunda fila:

- **Proceed.** Continuar la ejecución
- **Restart.** Inicia la ejecución del método actual desde el principio.

¡Puedes editar un método mostrado en el panel de código, guardarlo y reiniciarlo!

- **Into.** Pasa al código del siguiente mensaje enviado.

- **Over.** Pasar por alto el envío del mensaje.
  Realiza el siguiente envío de mensaje, pero permanece en el método actual.

- **Through.** Entra en un bloque de código pasando por alto los envíos de mensajes intermedios.
  Es útil cuando necesitas examinar lo que ocurre en un bloque de código, argumento del mensaje paso a paso, por ejemplo, el mensaje `#do:`.

Ahora vamos a jugar un poco. Si te desincronizas con las instrucciones aquí, simplemente cierra el depurador y vuelve a empezar con Debug-It.

Figura 10.4: Step Into

A medida que avanzas *paso a paso* en el depurador, cambia el resaltado del *siguiente* mensaje enviado. Pulsa tres veces la tecla `Over`. Deberías ver resaltada la línea que comienza por *childrenDo:*. Ahora pulsa la tecla `Into`.

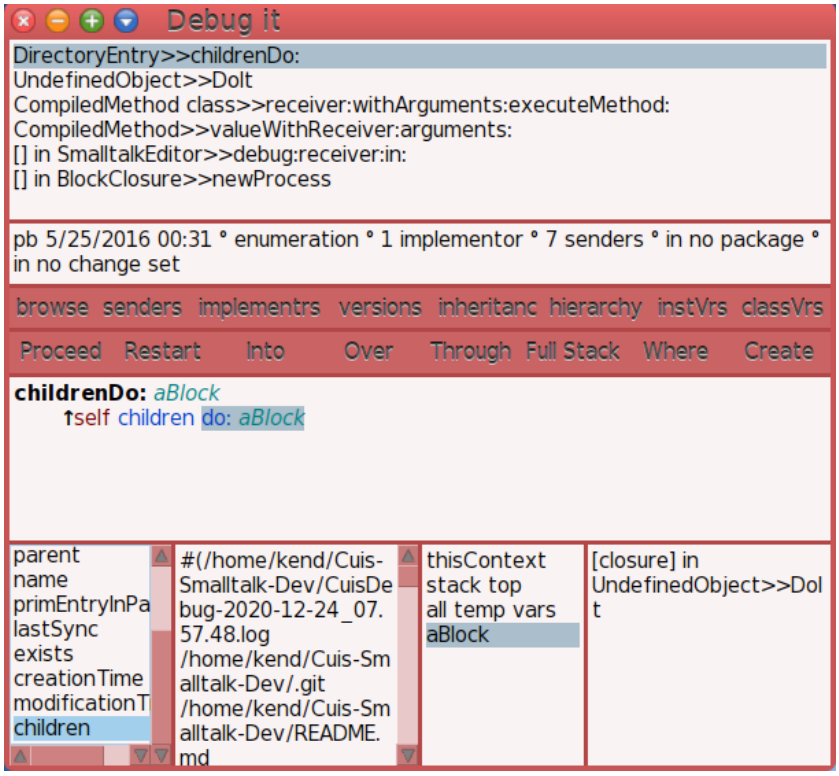

Figura 10.5: Visualización del objeto de foco y los temporales

El área de pila muestra que el objeto enfocado es un `DirectoryEntry`. Inspecciona sus valores de instancia seleccionando líneas en el panel inferior izquierdo.

El área de pila muestra que el método de enfoque es `DirectoryEntry>>childrenDo:`. Este es el método que se muestra en el panel de código.

El argumento para `childrenDo:` es `aBlock`. No hay variables de método que mostrar.

Si pulsas ***`Over`*** de nuevo e ***`Into`***, deberías ver el contexto en el que se está procesando `do:`.

Este podría ser un buen momento para investigar los inspectores, examinar la pila de arriba abajo y juega un poco con ella. A estas alturas, ya deberías sentirte seguro de que comprendes los conceptos básicos de lo que se muestra aquí.

¡Tienes el control!

Veamos brevemente otra forma de hacerlo.

10.3 ¡Alto!

Un *breakpoint* es un punto del código en el que se desea detener el procesamiento del código y examinarlo. No siempre se desea avanzar paso a paso para encontrar un problema, especialmente uno que solo se produce de vez en cuando. Es muy útil establecer un *breakpoint* en el punto en el que se produce el problema.

En Smalltalk, se utiliza el método `halt` para establecer un punto de interrupción. El mensaje `#halt` se envía a un objeto que es el foco inicial del depurador.

Cambia el código del Workspace para añadir un `#halt` como se indica a continuación.

```
| fileNames |
fileNames ← OrderedCollection new.
(DirectoryEntry smalltalkImageDirectory)
    childrenDo: [ :f | fileNames add: f name. fileNames halt ].
fileNames asArray.
```

Ejemplo 10.4: Halt: establece un Breakpoint

El objeto que recibe el mensaje `#halt` se convierte en el objeto de enfoque del depurador.

Volvamos a pulsar `Ctrl-a`. (*select* **A***ll*) y `Ctrl-p` (**P***rint-it*).

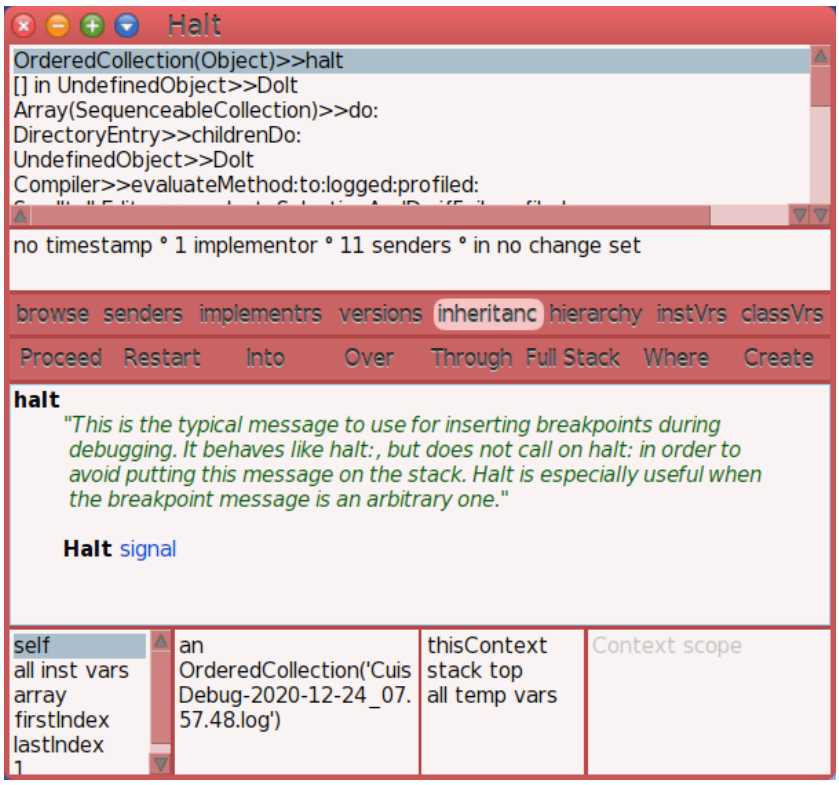

Figura 10.6: Halt

Presiona dos veces `Over` para sobrepasar el *breakpoint*.

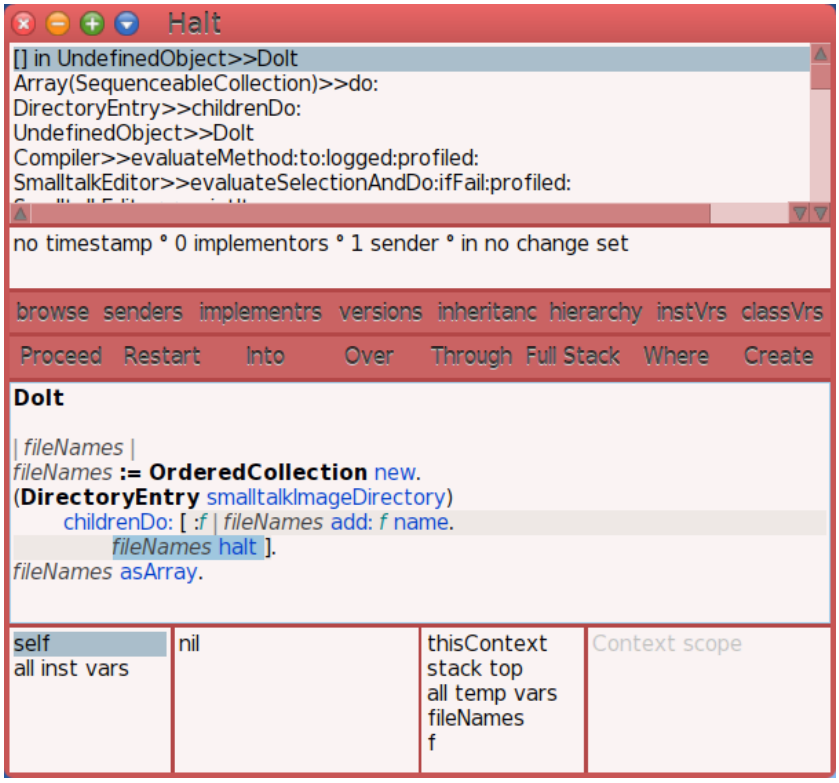

Figura 10.7: Step Over Breakpoint

Bueno, esto me resulta familiar. Sé lo que hay que hacer aquí.

Fíjate en que el comando `halt` está dentro de un bucle. Mientras estés en el bucle, cada vez que pulses ***`Proceed`***, *alcanzarás* el punto de interrupción en la siguiente iteración del bucle.

En muchos entornos de programación, para realizar un cambio es necesario eliminar un proceso, recompilar el código y, a continuación, iniciar un nuevo proceso.

Smalltalk es un entorno *vivo*. Puedes interrumpir, cambiar o escribir código (botón ***`Create`*** situado en la parte central derecha), reiniciar el marco de la pila y continuar con el procesamiento, ¡todo ello sin deshacer la pila de contexto!

A modo de analogía, en muchos lenguajes de programación, es como si te golpearas el dedo del pie y fueras al médico. El médico te dice: «Sí, te has golpeado el dedo del pie», saca una pistola, te dispara y le envía una nota a tu madre diciendo: «Ten otro hijo». ¡Smalltalk es mucho más amigable!

Ten en cuenta que un gran poder conlleva una gran responsabilidad.[1]. En un sistema abierto, se puede colocar un punto de interrupción en cualquier lugar, ¡incluso en lugares que pueden interrumpir la interfaz de usuario! Por ejemplo, ¡podría ser perjudicial colocar un punto de interrupción en el código del depurador!

[1] `https://quoteinvestigator.com/2015/07/23/great-power/`

11 Compartir Cuis

> La programación casi nunca es una comunión solitaria entre un hombre y una máquina. Preocuparse por los demás es una decisión consciente que requiere práctica.
>
> —*Kent Beck, "Smalltalk Best Practice Patterns"(1997)*

> La programación sigue siendo un proceso intensamente colaborativo entre grupos de lectores y escritores de programas.
>
> —*Dave Thomas, "Smalltalk With Style"(1996)*

> Deja que tu código hable: los nombres importan. Deja que el código diga lo que significa. Introduce un método para todo lo que haya que hacer. No tengas miedo de delegar, ni siquiera en ti mismo.
>
> —*Oscar Nierstrasz, "Best Practice Patterns - talk slides"(2009)*

Este libro es una invitación.

Esperamos que estés utilizando Cuis-Smalltalk para descubrir caminos de interés y que estés disfrutando del viaje. Si es así, en algún momento habrás hecho algo maravilloso y probablemente quieras compartirlo.

Compartir requiere comunicar la intención.

Para escribir bien hay que practicar. Los buenos escritores leen.

11.1 Reglas de oro del Gremio de Smalltalk

Preguntas básicas, que parecen ser las reglas de oro del gremio intergaláctico Smalltalk[1]:

- ¿Son los métodos breves y comprensibles?
- ¿Una línea de código se lee como una frase?
- ¿Los nombres de los métodos describen lo que hacen, en lugar de cómo lo hacen?
- ¿Los nombres de las variables de clase e instancia indican su función o funciones?
- ¿Hay comentarios útiles sobre la clase?
- ¿Podemos simplificarlo? ¿Dejar algo fuera?

Llevamos ya un tiempo trabajando con código Smalltalk, por lo que solemos seguir las buenas prácticas, pero echemos otro vistazo a nuestro código y veamos si podemos facilitar su comprensión al lector.

11.2 Refactorizar para mejorar la comprensión

Al examinar el código, observo un método denominado `#left`, que parece ser una abreviatura. Puedo solicitar a los *emisores* que me muestren cómo se utiliza `#left` en otras partes del código.

[1] *No te asustes*, en esta fase del libro, los autores aún están buscando todas las preguntas que realmente importan.

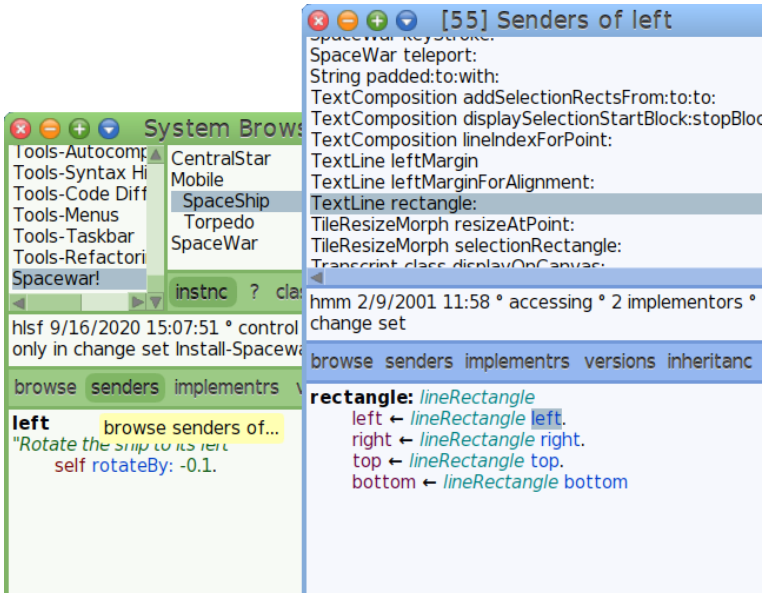

Figura 11.1: Emisores de `left`

He observado que la mayoría de los usos de `#left` son para indicar una posición, no para realizar una acción. ¿Cómo puedo solucionarlo?

Dado que las personas suelen querer cambiar las cosas para mejor, existen varias herramientas útiles que ayudan a hacerlo.

Ahora podría ver nuestros usos de `#left` en Spacewars!, pero el IDE de Cuis ya sabe cómo hacerlo.

Si hago clic con el botón derecho del ratón en el panel de métodos del Browser, aparece un menú contextual con opciones que me ayudan. Aquí elijo **Rename**.

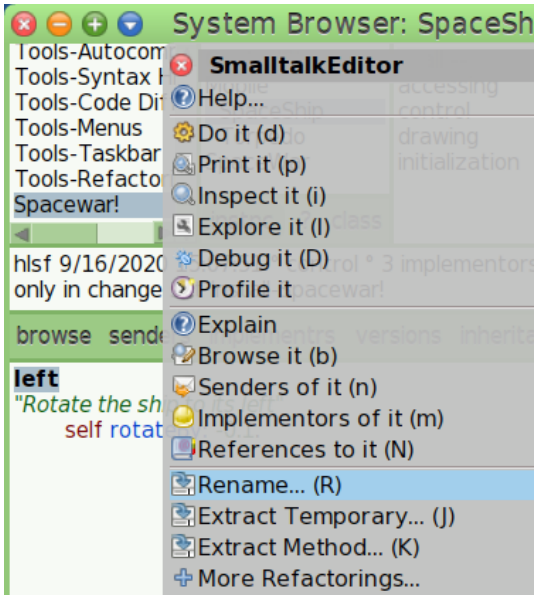

Figura 11.2: Renombrar `left`

Ahora bien, las herramientas que nos ayudan a refactorizar el código son bastante potentes, por lo que hay que actuar con moderación. No quiero cambiar todos los usos de `#left` en el sistema Cuis-Smalltalk, solo en la categoría `Spacewar!`.

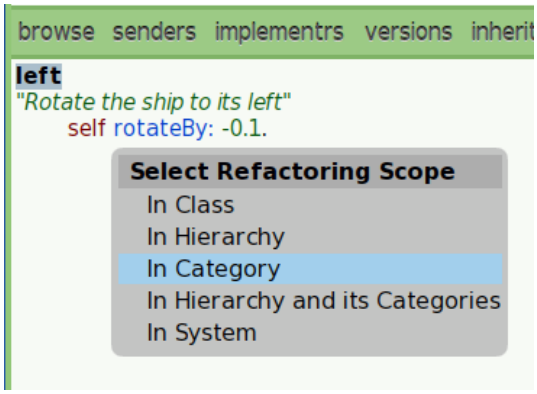

Figura 11.3: Renombrar en una categoría

Por supuesto, cuando se realizan cambios, uno quiere ver que el resultado es el esperado.

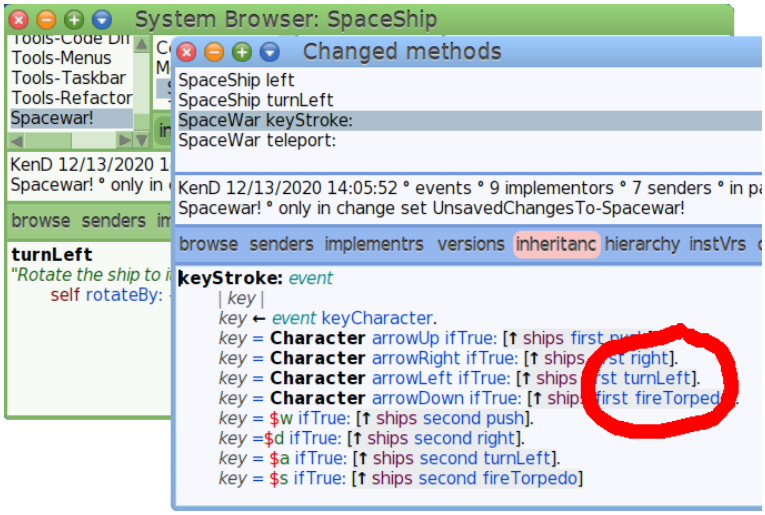

Figura 11.4: Resultados de renombrar

Como no soy perfecto, suelo guardar la imagen de Cuis-Smalltalk antes de realizar cambios importantes con herramientas potentes. Si ocurre algo que no deseaba, puedo salir de la imagen sin guardar y reiniciar la imagen guardada, que recuerda el estado anterior al cambio.

Renombrar `#right` *a* `#turnRight`.

Ejercicio 11.1: Renombrar un método

Al seguir explorando en el Browser, me fijo en el método `SpaceShip>>nose`.

¿Dónde lo utilicé? Ah, emisores...

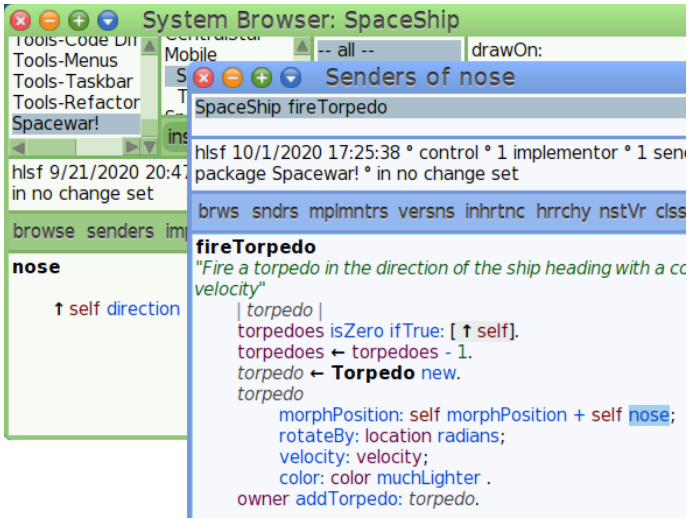

Figura 11.5: Emisores de `nose`

Mmmm, quizá algo más específico. ¿Qué tal `#noseDirection`? ¿Qué te parece?

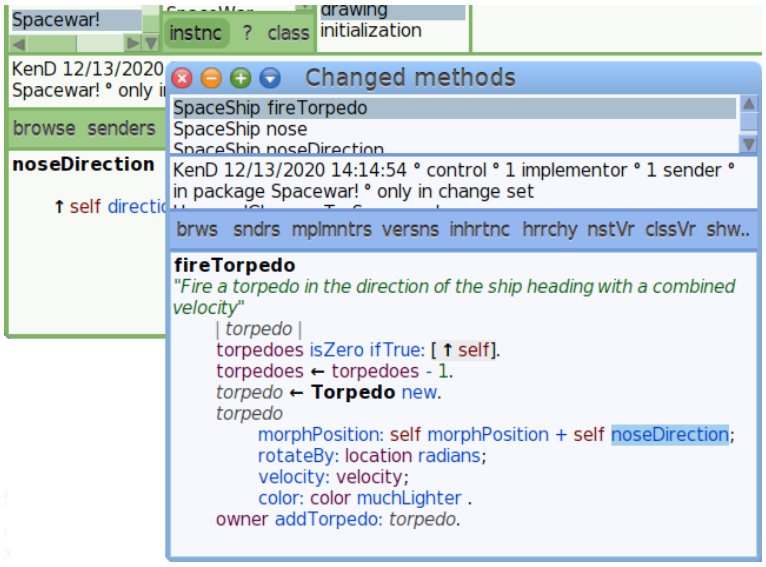

Figura 11.6: Renombrar `nose` a `noseDirection`


Menú World → `Help` es tu aliado. La `Terse Guide to Cuis` (La Guía Concisa de Cuis) te da acceso a una gran variedad de ejemplos de uso del código. El `Class Comment Browser` (El Navegador de Comentarios de Clase) es otra forma de encontrar información interesante sobre las clases. También hay más notas sobre la gestión del código y cómo usamos GitHub.


¡Queremos compartirlo con vosotros! Visitad los paquetes en el repositorio principal de Cuis-Smalltalk en `https://github.com/Cuis-Smalltalk`, buscad en GitHub los repositorios cuyos nombres empiecen por *Cuis-Smalltalk-* y echad un vistazo a los tutoriales y la información disponibles en `https://github.com/Cuis-Smalltalk/Learning-Cuis`.

Hay mucho más por explorar, pero este libro es una *introducción* y tenemos que dejar de escribir en algún momento. Este es un buen momento. ¡Queremos volver a escribir código! ¡Y estamos deseando ver *tus* proyectos!

¡Bienvenido a Cuis-Smalltalk!

Apéndice A Copyright de documentos

Parte del capítulo sobre sintaxis del libro *Squeak by Example* ha sido tomado y editado en el presente libro.

Copyright © 2007, 2008, 2009 by Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz and Damien Pollet.

Figura 1.4

Spacewar corriendo en un PDP-1, Joi Ito, 12 May 2007, redimensionado,
https://www.flickr.com/photos/35034362831@N01/494431001
https://creativecommons.org/licenses/by/2.0/deed.en

[Adele Goldberg quote], página 26

Historia oral de Adele Goldberg, Museo de Historia de la Informática, 10 de mayo de 2010.

Cuis-Smalltalk mascot

La cobaya australiana (Microcavia australis) es una especie de roedor sudamericano de la familia Caviidae.
Copyright © Euan Mee

Apéndice B Resumen de sintaxis

Sintaxis	Qué representa
`startPoint`	un nombre de variable
`Transcript`	un nombre de variable global
`self`	pseudo-variable
`1`	entero (decimal)
`2r101`	entero (base 2)
`16r1a`	entero (base 16)
`1.5`	número de coma flotante
`2.4e7`	notación exponencial
`$a`	el carácter 'a'
`'Hello'`	La cadena "Hello"
`#Hello`	el símbolo #Hello
`#(1 2 3)`	un array literal
`{1. 2. 1 + 2}`	un array dinámico
`"a comment"`	un comentario
`| x y |`	declaración de variables x e y
`x ← 1`, `x := 1`	asignar 1 a x
`[x + y]`	un bloque que se evalúa como x+y
`<primitive: 1>`	primitiva de la máquina virtual o anotación
`3 factorial`	un mensaje unitario
`3 + 4`	un mensaje binario
`2 raisedTo: 6 modulo: 10`	un amensaje de palabra clave
`↑ true`, `^ true`	devuelve el valor true
`Transcript show: 'hello'. Transcript cr`	separador de expresión (.)
`Transcript show: 'hello'; cr`	mensaje en cascada (;)
`ColoredBoxMorph new :: color: Color blue; openInWorld`	mensaje en cadena (::)
`` `{ 3@4 . 56 . 'click me'} ``	la composición del literal `#(3@4 56 'click me')`

Variables locales.

`startPoint` es un nombre de variable o identificador. Por convención, los identificadores se componen de palabras en «camelCase» (es decir, cada palabra, excepto la primera, comienza con una letra mayúscula). La primera letra de una variable de instancia, método o argumento de bloque, o variable temporal debe ser minúscula. Esto indica al lector que la variable tiene un ámbito privado.

Shared variables

Los identificadores que comienzan con letras mayúsculas son variables globales, variables de clase, diccionarios de grupo o nombres de clase. `Smalltalk` es una variable global, una instancia de la clase `SystemDictionary`.

El receptor.

`self` es una palabra clave que hace referencia al objeto dentro del cual se está ejecutando el método actual. Lo llamamos «the receiver» (el receptor) porque este objeto normalmente habrá recibido el mensaje que provocó la ejecución del método. self se denomina «pseudovariable» ya que no podemos asignarle ningún valor.

Enteros.

Además de los números enteros decimales comunes como `42`, Cuis-Smalltalk también ofrece una notación de base. `2r101` es 101 en base 2 (es decir, binario), que es igual a 5 en decimal.

Números de coma flotante.

Los números de coma flotante se pueden especificar con su exponente en base diez: `2.4e7` es 2.4×10^7.

Caracteres.

El signo del dólar introduce un carácter literal: `$a` es el literal para 'a'. Las instancias de caracteres no imprimibles se pueden obtener enviando mensajes con los nombres adecuados a la clase `Character`, como son `Character space` y `Character tab`.

Cadenas.

Las comillas simples se utilizan para definir una cadena literal. Si desea una cadena con una comilla dentro, simplemente duplique la comilla, como en `'G''day'`.

Símbolos.

Los símbolos son como cadenas, en el sentido de que contienen una secuencia de caracteres. Sin embargo, a diferencia de una cadena, un símbolo literal tiene garantizada su unicidad global. Solo hay un objeto Símbolo `#Hello` pero puede haber varios objetos Cadena con el valor `'Hello'`.

Arrays estáticos.

Los arrays estáticos o arrays en tiempo de compilación se definen mediante `#( )`, , que rodea a los literales separados por espacios. Todo lo que se encuentre dentro de los paréntesis debe ser una 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 que contiene los dos booleanos y el símbolo `#abc`.

Arrays dinámicos.

Arrays dinámicos o arrays en tiempo de ejecución. Las llaves { } definen un array (dinámico) en tiempo de ejecución. Los elementos son expresiones separadas por puntos. Así, `{ 1. 2. 1+2 }` define un array con los elementos `1`, `2`, y el resultado de evaluar `1+2`. (¡La notación con llaves es propia del dialecto Squeak de la familia Smalltalk! En otros Smalltalks, debes crear matrices dinámicas de forma explícita).

Comentarios.

Los comentarios se escriben entre comillas dobles. `"hello"` es un comentario, no una cadena, y el compilador Cuis-Smalltalk lo ignora. Los comentarios pueden abarcar varias líneas.

Declaración 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. En la versión impresa del libro escribimos ← instead. en su lugar. Dado que este carácter no está presente en el teclado, se escribe con la tecla del carácter de subrayado. Por lo tanto, `x := 1` es lo mismo que `x ← 1` o `x _ 1`.

Bloques.

Los corchetes `[ ]` definen un bloque, también conocido como closure de bloque o closure léxico, que es un objeto de primera clase que representa una función. Como veremos, los bloques pueden tomar argumentos y tener variables locales.

Primitivas.

`<primitive: ...>` denota una invocación de una primitiva de máquina virtual. (`<primitive: 1>` es la primitiva de VM para `SmallInteger>>+`.) Cualquier código que siga a la primitiva se ejecuta solo si la primitiva falla. La misma sintaxis se utiliza también para las anotaciones de métodos.

Mensajes unarios.

Los mensajes unarios consisten en una sola palabra (como `#factorial`), que se envía a un receptor (como 3).

Mensajes binarios.

Los mensajes binarios son operadores (como `+`) que se envían a un receptor y toman un único argumento. En `3 + 4`, el receptor es `3` y el argumento es `4`.

Mensajes de palabra clave.

Los mensajes de palabras clave constan de varias palabras clave (como `#raisedTo:modulo:`), cada una de las cuales termina con dos puntos y toma un único argumento. En la expresión `2 raisedTo: 6 modulo: 10`, el selector de mensajes `raisedTo:modulo:` toma los dos argumentos 6 y 10, uno después de cada dos puntos. Enviamos el mensaje al receptor 2.

Retorno de método.

Se utiliza ↑ para devolver un valor desde un método. (Debes escribir `^` para obtener el carácter ↑.)

Secuencias de declaraciones.

Un punto o parada completa (.) es el separador de sentencias. Al poner un punto entre dos expresiones, estas se convierten en sentencias independientes.

Cascadas.

Cadenas.

Hay dos tipos de composición de mensajes, en cascada y en cadena

El punto y coma «;» se utilizan para enviar series de mensajes al mismo receptor original. En `Transcript show: 'hello'; cr` primero envíamo el mensaje `#show: 'hello'` al receptor `Transcript`, y después se envía el mensaje unitario `#cr` al mismo receptor.

A veces resulta útil enviar una serie de mensajes al *resultado* del envío de un mensaje. En `ColoredBoxMorph new :: color: Color blue; openInWorld.` enviamos mensajes sucesivos a la nueva instancia ColoredBoxMorph, no a la clase ColoredBoxMorph.

Para comprender mejor las diferencias entre la cascada y la cadena de mensajes, fíjate en el resultado de las tres instrucciones siguientes:

```
3 + 4 squared
⇒ 19
```

```
3 + 4 ; squared
⇒ 9

3 + 4 :: squared
⇒ 49
```

Literal compuesto

Las comillas invertidas (`) se pueden utilizar para crear literales compuestos en tiempo de compilación. Todos los componentes de un literal compuesto deben ser conocidos cuando se compila el código.

Apéndice C Los ejercicios

Ejercicio 1: Soy un ejemplo de ejercicio ... 3
Ejercicio 1.1: Situar en el centro ... 9
Ejercicio 1.2: Concatenación y mayúsculas ... 10
Ejercicio 1.3: Suma de inversos ... 11
Ejercicio 1.4: Número en palabras en mayúsculas ... 11
Ejercicio 2.1: de Hello a Belle ... 17
Ejercicio 2.2: Suma de los cuadrados ... 18
Ejercicio 2.3: Recuento de métodos ... 21
Ejercicio 3.1: Información de la clase Float ... 31
Ejercicio 3.2: Tabla de cosenos ... 36
Ejercicio 3.3: Multiplicar por 1024 ... 37
Ejercicio 3.4: Errores de cálculo con números decimales ... 37
Ejercicio 3.5: Hacia el infinito ... 38
Ejercicio 3.6: Arreglando errores ... 38
Ejercicio 3.7: Seleccionar manzanas ... 39
Ejercicio 3.8: Formatea una cadena ... 40
Ejercicio 3.9: Variables de instancia de los protagonistas de Spacewar! ... 42
Ejercicio 3.10: Mensajes *getter* de `SpaceShip` ... 43
Ejercicio 3.11: Mensajes *setter* de `SpaceShip` ... 44
Ejercicio 3.12: Métodos para controlar el rumbo de la nave ... 44
Ejercicio 3.13: Métodos para controlar la aceleración de la nave ... 45
Ejercicio 3.14: Inicializar la estrella central ... 45
Ejercicio 4.1: Cortar una cadena ... 47
Ejercicio 4.2: Números negativos enteros ... 51
Ejercicio 4.3: Agujero en un conjunto ... 51
Ejercicio 4.4: Enteros impares ... 53
Ejercicio 4.5: Números primos entre 101 y 200 ... 53
Ejercicio 4.6: Múltiplos de 7 ... 53
Ejercicio 4.7: Impares y no primos ... 53
Ejercicio 4.8: Decodificar cifrado ... 54
Ejercicio 4.9: Alfabeto del cifrado Caesar ... 54
Ejercicio 4.10: Codifica con el cifrado Caesar ... 55
Ejercicio 4.11: Decodifica con el cifrado de Caesar ... 55
Ejercicio 4.12: Acceder a parte de una colección ... 57
Ejercicio 4.13: Rellenar un array ... 57
Ejercicio 4.14: Añadir un elemento después ... 58
Ejercicio 4.15: Letras ... 59
Ejercicio 4.16: Color por nombre ... 60
Ejercicio 4.17: Colecciones para contener las naves y torpedos ... 61
Ejercicio 4.18: Actualizar todas las naves y torpedos ... 63
Ejercicio 5.1: Calcular divisores con un bloque ... 67
Ejercicio 5.2: Implementar `and:` y `or:` ... 69
Ejercicio 5.3: Clasificar un método ... 71
Ejercicio 5.4: Clasificar los métodos de control ... 72
Ejercicio 5.5: Colisión de naves ... 73
Ejercicio 5.6: Colisiones entre los torpedos y el Sol ... 73
Ejercicio 6.1: Hacer todos los Morphs ... 84
Ejercicio 6.2: Refactorizar `SpaceShip` y `Torpedo` ... 86
Ejercicio 7.1: Morph cruz ... 92
Ejercicio 7.2: Morph rectángulo ... 94

Ejercicio 7.3: Un reloj elegante 101
Ejercicio 7.4: Dibujo del chorro de gases 104
Ejercicio 7.5: Extensión del torpedo 105
Ejercicio 7.6: Dibujo de torpedo 106
Ejercicio 7.7: Acceso de la nave espacial a su diagrama en el lado de la clase 108
Ejercicio 7.8: Dibujar en `Mobile` 109
Ejercicio 7.9: Detección precisa de colisiones 111
Ejercicio 8.1: Recibir notificaciones del evento de desplazamiento del ratón 114
Ejercicio 8.2: Gestionar el evento de entrada del ratón 115
Ejercicio 8.3: Manejar el evento de salida del ratón 115
Ejercicio 8.4: Recibir notificaciones de eventos del teclado 116
Ejercicio 8.5: Teclas para controlar la nave del segundo jugador 117
Ejercicio 9.1: Describir un paquete 126
Ejercicio 9.2: Guardar el paquete Spacewar! 126
Ejercicio 9.3: Dos categorías de clases, un paquete 127
Ejercicio 11.1: Renombrar un método 145

Apéndice D Soluciones a los ejercicios

Preface

Ejercicio 1

En los años setenta se desarrollaron cuatro veriones: Smalltalk-71, Smalltalk-72, Smalltalk-76 y Smalltalk-80.

Filosofía Smallltalk

Ejercicio 1.1

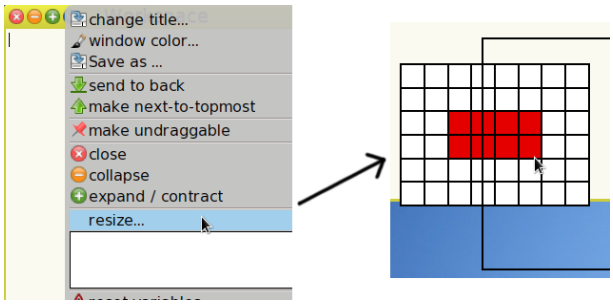

Figura D.1: Emplazamiento

Ejercicio 1.2

```
Transcript show: 'Hello ', 'my beloved ' asUppercase, 'friend'
```

Ejercicio 1.3

```
1 + (1/2) + (1/3) + (1/4)
⇒ 25/12
```

Ejercicio 1.4

Algunos mensajes se pueden enviar uno tras otro:

```
Transcript show: 2020 printStringWords capitalized
```

El estilo de vida del mensaje

Ejercicio 2.1

```
'Hello'
   at: 1 put: $B;
   at: 2 put: $e;
   at: 3 put: $l;
   at: 4 put: $l;
   at: 5 put: $e;
   yourself
```

Ejercicio 2.2

```
1 + (1/2) squared + (1/3) squared + (1/4) squared
⇒  205 / 144
```

Ejercicio 2.3

Desde un Browser del sistema, ve desde el panel izquierdo hacia el derecho ...`Kernel-Text` → `CharacterSequence` → `arithmetic`... El número de métodos en el último panel derecho es 6: `*`, `+`, `-`, `/`, `//` y `\\`.

Una alternativa es buscar una clase con un atajo de teclado: *`Ctrl-Espacio`* desde el entorno Cuis-Smalltalk o *`Ctrl-f`* desde la categoría de clases del *Browser*.

Clase – modelo de comunicación de entidades

Ejercicio 3.1

Cuando está seleccionado `Float`, el texto ancho muestra: "class definition for Float ° 92 instance methods ° 34 class methods ° 1280 total lines of code"

Ejercicio 3.2

```
0 to: Float twoPi by: 1/10 do: [:i |
   Transcript show: i cos; cr]
```

Ejercicio 3.3

1024 no es un número aleatorio. Es 2^{10}, escrito en base 2: 10000000000, también es `1 << 10`:

```
2↑10 ⇒ 1024
1024 printStringBase: 2 ⇒ '10000000000'
1 << 10 ⇒ 1024
```

Por lo tanto, para multiplicar un número entero por 1024, desplazamos sus dígitos 10 posiciones hacia la izquierda:

```
360 << 10 ⇒ 368640
360 * 1024 ⇒ 368640
```

Ejercicio 3.4

```
5.2 + 0.9  - 6.1
⇒ 8.881784197001252e-16

5.2 + 0.7 + 0.11
⇒ 6.010000000000001

1.2 * 3  - 3.6
⇒ -4.440892098500626e-16
```

Ejercicio 3.5

El sistema devuelve el error `ZeroDivide`, división por cero.

Ejercicio 3.6

```
(52/10) + (9/10)  - (61/10)
⇒ 0

(52/10) + (7/10)  + (11/100)
⇒ 601/100 soit 6.01

(12/10) * 3  - (36/10)
⇒ 0
```

Ejercicio 3.7

Hay diferentes opciones, con resultados significativamente distintos:

```
'There are 12 apples' select: [:i |i isLetter].
```

```
⇒ 'Thereareapples'
```

Realmente no es satisfactorio. Por lo tanto otra opción:

```
'There are 12 apples' select: [:i |i isDigit not].
⇒ 'There are  apples'
```

O incluso una opción más corta con el mensaje `#reject:`:

```
'There are 12 apples' reject: [:i |i isDigit].
⇒ 'There are  apples'
```

Ejercicio 3.8

En `String`, busca la categoría de método `format`, allí encontrarás el método `format:`:

```
'Joe bought {1} apples and {2} oranges' format: #(5 4)
⇒  'Joe bought 5 apples and 4 oranges'
```

Ejercicio 3.9

Las definiciones `SpaceWar`, `CentralStar` y `SpaceShip` con su variable de instancia añadida deberían tener el siguiente aspecto:

```
Object subclass: #SpaceWar
   instanceVariableNames: 'centralStar ships torpedoes'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

Object subclass: #CentralStar
   instanceVariableNames: 'mass'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

Object subclass: #SpaceShip
   instanceVariableNames: 'position heading velocity
      fuel torpedoes mass acceleration'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Ejercicio 3.10

```
SpaceShip>>position
   ↑ position

SpaceShip>>velocity
   ↑ position

SpaceShip>>mass
   ↑ mass
```

Ejercicio 3.11

```
SpaceShip>>position: aPoint
   position ← aPoint

SpaceShip>>velocity: aPoint
   velocity ← aPoint

Torpedo>>heading: aFloat
   heading ← aFloat
```

Ejercicio 3.12

```
SpaceShip>>left
```

```
"Rotate the ship to its left"
   heading ← heading + 0.1

SpaceShip>>right
"Rotate the ship to its right"
   heading ← heading - 0.1
```

Ejercicio 3.13

```
SpaceShip>>push
"Init an acceleration boost"
   acceleration ← 10

SpaceShip>>unpush
"Stop the acceleration boost"
   acceleration ← 0
```

Ejercicio 3.14

```
CentralStar>>initialize
   super initialize.
   mass ← 8000.
```

El estilo de vida de la colección

Ejercicio 4.1

Abre el browser de protocolo en la clase `String`, busca el método `allButFirst:` implementado en `SequenceableCollection`. Lee en su código fuente el comentario.

```
'Hello My Friend' allButFirst: 6
⇒ 'My Friend'
```

Ejercicio 4.2

```
(-80 to: 50) asArray
```

Ejercicio 4.3

```
(1 to: 100) difference: (25 to: 75)
⇒ #(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
92 93 94 95 96 97 98 99 100)
```

Ejercicio 4.4

```
(-20 to: 45) select: [:z | z odd]
```

Ejercicio 4.5

```
((101 to: 200) select: [:n | n isPrime]) size
⇒ 21
```

Ejercicio 4.6

```
(1 to: 100) select:[:n | n isDivisibleBy: 7]
⇒ #(7 14 21 28 35 42 49 56 63 70 77 84 91 98)
```

Ejercicio 4.7

Esta solución, basada en operaciones con conjuntos y el uso múltiple del mensaje `#select:`, es muy compatible con los conocimientos adquiridos hasta este punto del libro.

```
| primeNumbers nonPrimeNumbers |
primeNumbers ← (1 to: 100) select: [:n | n isPrime].
nonPrimeNumbers ← (1 to: 100) difference: primeNumbers.
nonPrimeNumbers select: [:n | n odd]
```

```
⇒ #(1 9 15 21 25 27 33 35 39 45 49 51 55 57 63 65 69 75
77 81 85 87 91 93 95 99)
```

Una solución más corta con operaciones lógicas se discutirá más adelante:

```
(1 to: 100) select:[:n | n isPrime not and: [n odd]]
```

Ejercicio 4.8

```
'Zpv!bsf!b!cptt' collect: [:c |
   Character codePoint: c asciiValue - 1]
⇒ 'You are a boss'
```

Ejercicio 4.9

```
($A to: $Z) collect: [:c |
   Character codePoint: c asciiValue - 65 + 3 \\ 26 + 65]
```

Cada carácter de la A a la Z se referencia con el parámetro `c` del bloque y se convierte a su valor ASCII[1]: `c asciiValue`. Se desplaza 65, por lo que la letra A cuenta como 0 y la Z como 25 en la lista alfabética. Para aplicar el cifrado César, enviamos el mensaje `+ 3` al resultado anterior.

Ya casi lo tenemos, solo los caracteres X, Y, Z desbordarán el alfabeto, de hecho estos caracteres se cifrarán como 26, 27, 28. Para solucionarlo, enviamos `\\ 26` al resultado anterior, es un viejo truco de programación: el resto de la división euclidiana por 26 enmarca el valor entre 0 y 25.[2].

Por último se desplaza de nuevo 65 antes de convertir el valor ASCII al carácter.

Ejercicio 4.10

En la solución del Ejercicio 4.9 sólo necesitamos reemplazar el intervalo de caracteres por una cadena:

```
'SMALLTALKEXPRESSION' collect: [:c |
   Character codePoint: c asciiValue - 65 + 3 \\ 26 + 65]
⇒ 'VPDOOWDONHASUHVVLRQ'
```

Ejercicio 4.11

```
  'DOHDMDFWDHVW' collect: [:c |
    Character codePoint: c asciiValue - 65 - 3 \\ 26 + 65]
⇒ 'ALEAJACTAEST'
```

Ejercicio 4.12

El mensaje apropiado es `#first:`, definido en la clase madre `SequenceableCollection`. Necesitas utilizar el visor de protocolo o jerarquía para descubrirlo en `Array`:

```
array1 first: 2
⇒ #(2 'Apple')
```

Ejercicio 4.13

Simplemente hacer un conteo:

```
array1 at: 1 put: 'kiwi'.
array1 at: 2 put: 'kiwi'.
array1 at: 3 put: 'kiwi'.
array1 at: 4 put: 'kiwi'.
```

[1] `https://en.wikipedia.org/wiki/ASCII`

[2] Se generaliza al campo matemático de la aritmética modular, `https://www.wikiwand.com/en/Modular_arithmetic`

O incluso algo menos que un conteo:

```
1 to: array1 size do: [:index |
   array1 at: index put: 'kiwi']
```

Pero si buscas cuidadosamente en el protocolo `Array`, puedes hacer nada más que:

```
array1 atAllPut: 'kiwi'.
```

Ejercicio 4.14

En el protocolo `OrderedCollection` busca el método `add:after:`.

```
coll1 ← {2 . 'Apple' . 2@1 . 1/3 } asOrderedCollection .
coll1 add: 'Orange' after: 'Apple'; yourself.
⇒  an OrderedCollection(2 'Apple' 'Orange' 2@1 1/3)
```

Ejercicio 4.15

```
Set  new
   addAll: 'buenos días';
   addAll: 'bonjour';
   yourself.
⇒  a Set($e $j $o $a $u $b $  $í $r $d $n $s)
```

Ejercicio 4.16

```
colors keysDo: [:key |
   colors at: key put: key asString capitalized].
colors
⇒ a Dictionary(#blue->'Blue' #green->'Green' #red->'Red'
#yellow->'Yellow' )
```

Ejercicio 4.17

Cuando comienza el juego no hay torpedos disparados, por lo que `torpedoes` es una `OrderedCollection` vacía, instanciada con el mensaje de clase `#new`.

Por otro lado, `ships` es un `Array` que contiene solo dos elementos, las naves de jugador. Usamos el mensaje de clase `#with:with` para instanciar y poblar el array con dos naves creadas en el mensaje de argumento.

Por legibilidad, separamos el código en varias líneas con la indentación apropiada.

```
torpedoes ← OrderedCollection new.
ships ← Array
   with: SpaceShip new
   with: SpaceShip new.
```

Ejercicio 4.18

```
SpaceWar>>stepAt: msSinceLast
  ships do: [:each | each update: msSinceLast / 1000 ].
  ships do: [:each | each unpush ].
  torpedoes do: [:each | each update: msSinceLast / 1000 ].
```

Mensajes de control de flujo

Ejercicio 5.1

```
| divisors |
divisors ← [:x | (1 to: x) select: [:d | x \\ d = 0] ].
divisors value: 60.
⇒ #(1 2 3 4 5 6 10 12 15 20 30 60)
divisors value: 45
⇒ #(1 3 5 9 15 45)
```

Ejercicio 5.2

Comprueba las implementaciones en `Boolean`, `True` y `False`.

Ejercicio 5.3

Una vez editado y guardado el método, en el panel `Method`, seleccione su nombre `teleport:` luego haz ...click derecho → `more...` → `change category...` → `events`...

Ejercicio 5.4

En el panel de métodos, selecciona un método de control sin categoría, luego haz ...click derecho → `more...` → `change category` → `new...` teclea `control`.

Para categorizar el resto de los métodos de control sin categoría, repite el proceso pero simplemente selecciona `control` en el último paso porque la categoría ya existe.

Ejercicio 5.5

No necesitamos un iterador para detectar una colisión entre dos naves. Sin embargo, utilizamos un iterador para actuar sobre cada nave cuando se detecta una colisión.

```
SpaceWar>>collisionsShips
| positionA position B |
   positionA ← ships first morphPosition.
   positionB ← ships second  morphPosition.
   (positionA dist: positionB) < 25 ifTrue: [
      ships do: [:each |
         each flashWith: Color red.
         self teleport: each]
   ]
```

Las variables locales solo se utilizan para facilitar el formato del código fuente en el libro impreso.

Ejercicio 5.6

Solo tienes que seleccionar los fragmentos de código adecuados del ejercicio y los ejemplos referenciados.

```
SpaceWar>>collisionsTorpedoesStar
| position |
   position ← centralStar morphPosition.
   torpedoes do: [:each |
      (each morphPosition dist: position) < 8 ifTrue: [
         each flashWith: Color orange.
         self destroyTorpedo: each]]
```

Visual con Morph

Ejercicio 6.1

Simplemente sustituye todas las ocurrencias de `Object` con `PlacedMorph`:

```
PlacedMorph subclass: #SpaceWar
   instanceVariableNames: 'centralStar ships torpedoes'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

PlacedMorph subclass: #CentralStar
   instanceVariableNames: 'mass'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

```
PlacedMorph subclass: #SpaceShip
   instanceVariableNames: 'name position heading velocity
      fuel torpedoes mass acceleration'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Ejercicio 6.2

```
Mobile subclass: #SpaceShip
   instanceVariableNames: 'name fuel torpedoes'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'

Mobile subclass: #Torpedo
   instanceVariableNames: 'lifeSpan'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Spacewar!'
```

Los fundamentos de Morph

Ejercicio 7.1

El método `drawOn:` se modifica para dibujar dos líneas distintas, sin conexión:

```
LineExampleMorph>>drawOn: aCanvas
   aCanvas strokeWidth: 20 color: Color green do: [
      aCanvas
         moveTo: 0 @ 0;
         lineTo: 200 @ 200;
         moveTo: 200 @ 0;
         lineTo: 0 @ 200 ]
```

Fíjate cómo el mensaje `#moveTo:` mueve el lápiz a una posición determinada con el lápiz levantado y, a continuación, el mensaje `#lineTo:` le pide al lápiz que dibuje desde la posición anterior hasta la nueva posición.

Ejercicio 7.2

Creamos un `RectangleExampleMorph`, subclase de `PlacedMorph`:

```
PlacedMorph subclass: #RectangleExampleMorph
   instanceVariableNames: 'fillColor'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Morphic-Learning'
```

A continuación, los métodos necesarios para inicializar y dibujar el Morph:

```
initialize
   super initialize .
   fillColor ← Color random alpha: 0.5

drawOn: aCanvas
   aCanvas
      strokeWidth: 1
      color: Color blue
      fillColor: fillColor
      do: [
         aCanvas moveTo: 0 @ 0;
            lineTo: 200 @ 0;
            lineTo: 200 @ 100;
            lineTo: 0 @ 100;
```

```
lineTo: 0 @ 0]
```

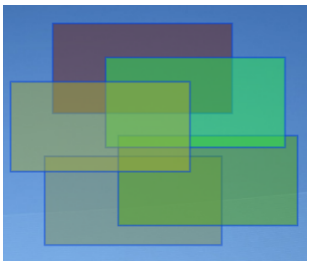

Figura D.2: Varios morphs rectangulares

Ejercicio 7.3

Entre las piezas del reloj (submorphs), sólo necesitamos modificar el dibujo de la clase `ClockSecondHandMorph`. El disco está rodeado por una fina línea roja y relleno de amarillo.

```
ClockSecondHandMorph>>drawOn: aCanvas
   aCanvas strokeWidth: 1.5 color: Color red do: [
      aCanvas
         moveTo: 0 @ 0;
         lineTo: 85 @ 0 ].
   aCanvas ellipseCenter: 0 @ -70 radius: 3 @ 3
      borderWidth: 1
      borderColor: Color red fillColor: Color yellow
```

Ejercicio 7.4

Además del método `drawOn:`, también queremos que la variable de aceleración tome un rango de valores entre 0 y 50.

```
SpaceShip>>push
"Init an accelaration boost"
   fuel isZero ifTrue: [^ self].
   fuel := fuel - 1.
   acceleration := (acceleration + 10) min: 50

SpaceShip>>unpush
"Stop the accelaration boost"
   acceleration := acceleration - 5 max: 0

drawOn: canvas
../..
"Draw gas exhaust"
   acceleration ifNotZero: [
      canvas
         line: c
         to: 0 acceleration
         width: 1 + acceleration / 8
         color: Color orange].
```

Ejercicio 7.5

El ancho del torpedo es de 4 píxeles y su altura de 8 píxeles:

```
Torpedo>>morphExtent
   ↑ `4 @ 8`
```

Ejercicio 7.6

El método `drawOn:` de `Torpedo` es muy similar al de la clase `SpaceShip`:

```
Torpedo>>drawOn: canvas
   | a b c |
   a ← 0 @ -4.
   b ← -2 @ 4.
   c ← 2 @ 4.
   canvas line: a to: b width: 2 color: color.
   canvas line: c to: b width: 2 color: color.
   canvas line: a to: c width: 2 color: color.
```

Ejercicio 7.7

Utilizamos una variable local porque usamos dos veces los vértices, una para dibujar la nave y otra para dibujar el escape de gas.

```
SpaceShip>>drawOn: canvas
   | vertices  |
   vertices ← self class vertices.
   canvas line: vertices first to: vertices second width: 2 color: color.
   canvas line: vertices second to: vertices third width: 2 color: color.
   canvas line: vertices third to: vertices fourth width: 2 color: color.
   canvas line: vertices fourth to: vertices first width: 2 color: color.
   "Draw gas exhaust"
   acceleration ifNotZero: [
      canvas line: vertices third to: 0@35 width: 1 color: Color gray]
```

Ejercicio 7.8

Necesitas tanto iterar cada vértice de la matriz `vertices` como acceder al vértice siguiente por índice. La operación aritmética de resto `#\\` es necesaria para mantener el índice dentro de los límites de la colección.

Cuando `size` es 4 (diagrama de la nave espacial), el argumento `(i \\ size + 1)` toma alternativamente los siguientes valores:

- **i = 1** => 1 \\ 4 + 1 = 1 + 1 = **2**
- **i = 2** => 2 \\ 4 + 1 = 2 + 1 = **3**
- **i = 3** => 3 \\ 4 + 1 = 3 + 1 = **4**
- **i = 4** => 4 \\ 4 + 1 = 0 + 1 = **1**

```
Mobile>>drawOn: canvas polygon: vertices
   | size |
   size ← vertices size.
   vertices withIndexDo: [: aPoint :i |
      canvas
         line: aPoint
         to: ( vertices at: (i  \\ size + 1) )
         width: 2
         color: color]
```

Ejercicio 7.9

Simplemente sustituye cada aproximación de distancia de posición de morph por la detección de intersección entre los límites de visualización de los morphs:

```
SpaceWar>>collisionsShips
   (ships first collides: ships second)
   ../..

SpaceWar>>collisionsShipsTorpedoes
   ships do: [:aShip |
      torpedoes do: [:aTorpedo |
         (aShip collides: aTorpedo)
   ../..
```

```
SpaceWar>>collisionsTorpedoesStar
   torpedoes do: [:each |
      (each collides: centralStar)
   ../..
```

Eventos

Ejercicio 8.1

El método `handlesMouseOver:`, implementado en la clase morph `SpaceWar`, devuelve verdadero para que el juego sea informado de los eventos de paso del ratón en métodos dedicados.

```
SpaceWar>>handlesMouseOver: event
   ↑ true
```

Ejercicio 8.2

Debes buscar el método `Morph>>handlesMouseOver:` y leer el comentario. Habla sobre un mensaje `#mouseEnter:`; implementamos el método correspondiente en la clase `SpaceWar` con los comportamientos descritos anteriormente:

```
SpaceWar>>mouseEnter: event
   event hand newKeyboardFocus: self.
   self startStepping
```

Ejercicio 8.3

El mensaje `#mouseLeave:` se envía a nuestra instancia `SpaceWar` cada vez que el cursor del ratón sale (abandona) el juego. Por lo tanto, añadimos el método homónimo a la clase `SpaceWar`:

```
SpaceWar>>mouseLeave: event
   event hand releaseKeyboardFocus: self.
   self stopStepping
```

Ejercicio 8.4

El mensaje `#handlesKeyboard` se envía a un morph para saber si desea recibir eventos del teclado. El morph responde `true` a este mensaje para indicar su interés en los eventos del teclado. Implementamos el método en la clase `SpaceWar`:

```
SpaceWar>>handlesKeyboard
   ↑ true
```

Ejercicio 8.5

Designamos los caracteres como `$w $a $s $d`. Añadimos el siguiente código al método `SpaceWar>>keyStroke:`

```
key = $w ifTrue: [↑ ships second push].
key = $d ifTrue: [↑ ships second right].
key = $a ifTrue: [↑ ships second left].
key = $s ifTrue: [↑ ships second fireTorpedo]
```

Gestión del código

Ejercicio 9.3

1. En el Browser del sistema, crea dos categorías de clases: en el menú del panel más a la izquierda, selecciona `add item... (a)` y escribe un nombre de categoría cada vez.
2. Crea dos clases, en cada categoría: la subclase `TheCuisBook` de `Object` y la subclase `TheBookMorph` de `PlacedMorph`.

Las dos operaciones anteriores se pueden realizar de una sola vez. Selecciona cualquier categoría existente e introduce la definición de clase con el nombre de la categoría:

```
PlacedMorph subclass: #TheBookMorph
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'TheCuisBook-Views'
```

Una vez guardada la definición de la clase, se crea la clase `TheBookMorph`, –como es lógico–, ¡pero también la categoría `TheCuisBook-Views`!

3. Ve a la herramienta Paquetes instalados ...Menú World → `Open...` → `Installed Packages`..., pulsa el botón `new` y escribe `TheCuisBook`.
4. Pulsa el botón `save` y ¡lo has hecho!

Apéndice E Los Ejemplos

Ejemplo 1: Soy un ejemplo con cabecera y resultado . . . 3
Ejemplo 1.1: El tradicional programa 'Hello World!' . . . 9
Ejemplo 1.2: Varias líneas . . . 9
Ejemplo 1.3: Concatenar cadenas . . . 10
Ejemplo 2.1: Calculando el número de entidades . . . 14
Ejemplo 2.2: Calcular el número de clases . . . 14
Ejemplo 2.3: Velocidad de la nave . . . 15
Ejemplo 2.4: Cascada de mensajes . . . 16
Ejemplo 2.5: Detener y teletransportar una nave espacial a una posición aleatoria . 16
Ejemplo 2.6: Pruebas con enteros . . . 19
Ejemplo 2.7: Calcular el vector de la fuerza de la gravedad . . . 19
Ejemplo 3.1: Preguntando la clase de una instancia . . . 30
Ejemplo 3.2: Alineando un torpedo con su dirección de velocidad . . . 33
Ejemplo 3.3: Redondeando números, pruebas en el Workspace . . . 34
Ejemplo 3.4: Bucles de intervalo (for-loop) . . . 34
Ejemplo 3.5: Lanzar un dado 5 veces . . . 35
Ejemplo 3.6: Intervalos . . . 35
Ejemplo 3.7: Teletransportar una nave . . . 35
Ejemplo 3.8: Entero representado en diversas bases . . . 36
Ejemplo 3.9: Contando como los antiguos . . . 36
Ejemplo 3.10: Desplazando bits . . . 36
Ejemplo 3.11: ¡Discalculia del ordenador! . . . 37
Ejemplo 3.12: ¡El cálculo es correcto utilizando fracciones! . . . 38
Ejemplo 3.13: Doce manzanas . . . 39
Ejemplo 3.14: La clase `Torpedo` con sus variables de instancia . . . 42
Ejemplo 3.15: Plantilla de método . . . 43
Ejemplo 3.16: Un método devolviendo una constante . . . 44
Ejemplo 3.17: Inicializar la nave espacial . . . 45
Ejemplo 4.1: Colección de tamaño dinámico . . . 51
Ejemplo 4.2: Operaciones de conjunto . . . 51
Ejemplo 4.3: Seleccionar los números primos entre el 1 y 100 . . . 52
Ejemplo 4.4: Cuenta cuántos números primos hay entre 1 y 100 . . . 53
Ejemplo 4.5: Recoger cubos . . . 54
Ejemplo 4.6: Cifrado sencillo . . . 54
Ejemplo 4.7: Un bloque *for* . . . 55
Ejemplo 4.8: Un bucle *repeat* . . . 55
Ejemplo 4.9: Colección con un tamaño fijo . . . 56
Ejemplo 4.10: Acceso a elementos de colección . . . 57
Ejemplo 4.11: Colección con un tamaño variable . . . 58
Ejemplo 4.12: Añadir, eliminar elementos de un array dinámico . . . 58
Ejemplo 4.13: Colección Set . . . 58
Ejemplo 4.14: Set, sin duplicados . . . 58
Ejemplo 4.15: Convertir un array dinámico . . . 59
Ejemplo 4.16: Diccionario de colores . . . 59
Ejemplo 4.17: Incomplete game initialization . . . 61
Ejemplo 4.18: Mecánicas del torpedo . . . 61
Ejemplo 4.19: Mecánicas de la nave espacial . . . 62
Ejemplo 4.20: Método dirección de la nave espacial . . . 62
Ejemplo 4.21: Gravedad de la nave espacial . . . 62
Ejemplo 4.22: Actualización periódica del juego . . . 63

Ejemplo 4.23: Colisión entre las naves y el Sol 63
Ejemplo 5.1: SpaceWar! pulsación de tecla (*key stroke*) 65
Ejemplo 5.2: Calcular divisores 67
Ejemplo 5.3: Método `teleport:` 67
Ejemplo 5.4: Implementaciones de ifTrue:ifFalse: 68
Ejemplo 5.5: Implementación de la negación 68
Ejemplo 5.6: Nave perdida en el espacio 69
Ejemplo 5.7: Inicializar `SpaceWar` 71
Ejemplo 5.8: Controles de la nave 71
Ejemplo 5.9: Disparar un torpedo desde una nave espacial en movimiento 72
Ejemplo 5.10: Colisión entre las naves y los torpedos 73
Ejemplo 6.1: Modificar el comportamiento de este morph desde su Inspector 79
Ejemplo 6.2: Modificar el estado de esta elipse desde su Inspector 81
Ejemplo 6.3: Completar el código para inicializar los actores de Spacewar! 85
Ejemplo 6.4: `Mobile` en el juego 86
Ejemplo 6.5: Calcular la fuerza de la gravedad 87
Ejemplo 6.6: Método `update:` de `Mobile` 88
Ejemplo 6.7: Límites de nuestros objetos Morph 88
Ejemplo 6.8: Comprobar que un mobile está «fuera del espacio» 88
Ejemplo 6.9: Sobreescribir Initialize en la jerarquía `Mobile` hierarchy 90
Ejemplo 7.1: Borrar todas las instancias de un morph dado 92
Ejemplo 7.2: Dibujando el reloj de esfera 98
Ejemplo 7.3: Extensión de la estrella central 103
Ejemplo 7.4: Una estrella con un tamaño fluctuante 103
Ejemplo 7.5: Dibujado de nave espacial 104
Ejemplo 7.6: `vertices` como variable instanciada en la `Mobile class` 107
Ejemplo 7.7: Inicializar una clase 107
Ejemplo 7.8: El valor de una variable de clase no es compartido por las subclases . 108
Ejemplo 7.9: `Vertices` una variable de clase `Mobile` 109
Ejemplo 7.10: Vértices devueltos por un método de instancia 109
Ejemplo 7.11: Colisión (superposición de rectángulos) entre las naves y el Sol 110
Ejemplo 7.12: Colisión (precisión de píxel) entre las naves y el Sol 110
Ejemplo 8.1: Efecto del foco del teclado en Spacewar! 116
Ejemplo 8.2: Teclas para controlar la nave del primer jugador 117
Ejemplo 9.1: Contenidos de Change set 122
Ejemplo 10.1: Asegurar que un `FileStream` está cerrado 134
Ejemplo 10.2: Capturar `thisContext` 135
Ejemplo 10.3: Nombres de las entradas del directorio 136
Ejemplo 10.4: Halt: establece un Breakpoint 140

Apéndice F Las Figuras

Figura 1: Cuis . . . 2
Figura 1.1: Establecer preferencias . . . 8
Figura 1.2: Opciones de la ventana . . . 8
Figura 1.3: Ventana Transcript con salida . . . 9
Figura 1.4: El juego Spacewar! en un microordenador DEC PDP-1 . . . 12
Figura 1.5: El juego Spacewar! . . . 13
Figura 2.1: El Browser del sistema . . . 20
Figura 2.2: Categoría de clases Spacewar! . . . 23
Figura 2.3: Ventana Installed Package . . . 23
Figura 2.4: Ecuaciones de las aceleraciones, velocidad y posición . . . 25
Figura 3.1: Métodos de clase en `Float` . . . 31
Figura 3.2: Métodos de instancia en `Float` . . . 32
Figura 4.1: Ojear el protocolo de Browse `String` . . . 46
Figura 4.2: Ojear la jerarquía de `String` . . . 47
Figura 5.1: Spacewar! torpedos alrededor . . . 72
Figura 6.1: Selecciona un `EllipseMorph` desde el menú . . . 75
Figura 6.2: Arrastra el controlador de construcción para cambiar el tamaño . . . 76
Figura 6.3: Una elipse más grande . . . 76
Figura 6.4: Obtener un `BoxedMorph` . . . 77
Figura 6.5: Hacer el rectángulo submorph de la elipse . . . 77
Figura 6.6: Clic-central para controles de construcción . . . 78
Figura 6.7: Click-central de nuevo para descender en los submorphs . . . 78
Figura 6.8: Añadir comportamiento específico de la instancia . . . 79
Figura 6.9: Mover un submorph dentro de su padre . . . 80
Figura 6.10: Sacar un submorph de su padre . . . 80
Figura 6.11: Inspeccionar variables de instancia de la elipse . . . 81
Figura 6.12: Utilizar el Inspector para establecer el color y el ancho del borde . . . 82
Figura 6.13: Obtener una `ColorClickEllipse` . . . 83
Figura 6.14: Botón `inheritance` de `update:` . . . 89
Figura 7.1: Detalle de nuestro morph de línea . . . 92
Figura 7.2: Varios morphs triangulares, uno decorado con su halo y sistema de coordenadas . . . 94
Figura 7.3: Morph animado . . . 96
Figura 7.4: Un submorph tirangular animado y recortado . . . 97
Figura 7.5: Un morph reloj . . . 97
Figura 7.6: Declarar variables desconocidas como variables de instancia en la clase actual . . . 100
Figura 7.7: `ClockMorph` con variables de instancia añadidas . . . 100
Figura 7.8: Un elegante morph de reloj . . . 101
Figura 7.9: Una estrella con un tamaño fluctuante . . . 103
Figura 7.10: Diagrama de la nave espacial al inicio del juego . . . 104
Figura 7.11: Diagrama de torpedo al inicio del juego . . . 105
Figura 7.12: El lado de clase del Browser del sistema . . . 107
Figura 7.13: Los límites de visualización de una nave espacial . . . 110
Figura 8.1: Efecto Spacewar! dependiendo del foco del teclado . . . 116
Figura 9.1: Cuis-Smalltalk informa sobre los cambios perdidos . . . 119
Figura 9.2: Selecciona manualmente los cambios que desea guardar en el archivo . 120
Figura 9.3: El *Change Sorter*, edición de clase . . . 121
Figura 9.4: El *Change Sorter*, editar método . . . 121
Figura 9.5: La herramienta File List, para instalar un conjunto de cambios y más . 122

Figura 9.6: La herramienta *Change List* para revisar las modificaciones realizadas en la imagen 123
Figura 9.7: Browser de paquetes instalados 124
Figura 9.8: Nuevo paquete – `Morphic-Learning` 125
Figura 9.9: Seleccionar el paquete (o la versión base de Cuis) que se necesita 125
Figura 9.10: Paquete guardado – `Morphic-Learning` 126
Figura 9.11: Change Sorter, método complementario al núcleo 128
Figura 9.12: Paquete con extensión a la clase `Integer` de la categoría de clases del sistema `Kernel-Numbers` 128
Figura 9.13: Entorno de una imagen iniciada con el script de configuración 131
Figura 10.1: Inspección de una instancia `ZeroDivide` 135
Figura 10.2: Nombres de archivos y directorios en un directorio 136
Figura 10.3: Depúralo 137
Figura 10.4: Step Into 138
Figura 10.5: Visualización del objeto de foco y los temporales 139
Figura 10.6: Halt 140
Figura 10.7: Step Over Breakpoint 141
Figura 11.1: Emisores de `left` 143
Figura 11.2: Renombrar `left` 144
Figura 11.3: Renombrar en una categoría 144
Figura 11.4: Resultados de renombrar 145
Figura 11.5: Emisores de `nose` 146
Figura 11.6: Renombrar `nose` a `noseDirection` 146
Figura D.1: Emplazamiento 155
Figura D.2: Varios morphs rectangulares 163

Apéndice G Spacewar! Source Code

En el código fuente que aparece abajo, interpreta el carácter "_" como "←".

```
'From Cuis6.0 [latest update: #6154] on 27 January 2024 at 5:42:28 pm'!
'Description SpaceWar!! game code'!
!provides: 'Spacewar!!' 1 97!
SystemOrganization addCategory: #'Spacewar!!'!


!classDefinition: #Mobile category: #'Spacewar!!'!
PlacedMorph subclass: #Mobile
          instanceVariableNames: 'acceleration color velocity'
          classVariableNames: ''
          poolDictionaries: ''
          category: 'Spacewar!!'!
!classDefinition: 'Mobile class' category: #'Spacewar!!'!
Mobile class
          instanceVariableNames: 'vertices'!

!classDefinition: #SpaceShip category: #'Spacewar!!'!
Mobile subclass: #SpaceShip
          instanceVariableNames: 'name fuel torpedoes'
          classVariableNames: ''
          poolDictionaries: ''
          category: 'Spacewar!!'!
!classDefinition: 'SpaceShip class' category: #'Spacewar!!'!
SpaceShip class
          instanceVariableNames: ''!

!classDefinition: #Torpedo category: #'Spacewar!!'!
Mobile subclass: #Torpedo
          instanceVariableNames: 'lifeSpan'
          classVariableNames: ''
          poolDictionaries: ''
          category: 'Spacewar!!'!
!classDefinition: 'Torpedo class' category: #'Spacewar!!'!
Torpedo class
          instanceVariableNames: ''!

!classDefinition: #SpaceWar category: #'Spacewar!!'!
PlacedMorph subclass: #SpaceWar
          instanceVariableNames: 'color ships torpedoes centralStar'
          classVariableNames: ''
          poolDictionaries: ''
          category: 'Spacewar!!'!
!classDefinition: 'SpaceWar class' category: #'Spacewar!!'!
SpaceWar class
          instanceVariableNames: ''!

!classDefinition: #CentralStar category: #'Spacewar!!'!
Morph subclass: #CentralStar
          instanceVariableNames: ''
          classVariableNames: ''
          poolDictionaries: ''
          category: 'Spacewar!!'!
!classDefinition: 'CentralStar class' category: #'Spacewar!!'!
CentralStar class
          instanceVariableNames: ''!


!Mobile commentStamp: 'hlsf 10/25/2020 09:16:19' prior: 0!
An abstract mobile with a mass and inner acceleration, subject to a gravity pull.!

!SpaceShip commentStamp: 'hlsf 10/25/2020 09:19:28' prior: 0!
```

```
A ship comes with a torpedo inventory and a fuel tank.
Player controls its heading and acceleration, and he can fire torpedo.!

!Torpedo commentStamp: 'hlsf 10/25/2020 09:18:37' prior: 0!
A torpedo comes with a lifespan and an automatic acceleration handling!

!SpaceWar commentStamp: 'hlsf 11/1/2020 10:06:46' prior: 0!
I am the view and controler of the game.
My submorphs are the elements of the game.

To start the game execute: "SpaceWar new openInWorld"

Ship Controls
Green: left, right, up down arrows
Red: a, d, w, s keys!

!CentralStar commentStamp: '<historical>' prior: 0!
I am the central start of the game generating a gravity field.!

!Mobile methodsFor: 'accessing' stamp: 'hlsf 10/1/2020 08:34:04'!
color: aColor
          "Set the receiver's color. "
          color = aColor ifFalse: [
                    color _ aColor.
                    self redrawNeeded ]! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 2/14/2022 13:48:54'!
direction
"I am an unit vector representing the noze direction of the mobile"
          ^Point rho: 1 theta: self heading! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 2/14/2022 13:48:44'!
heading
          ^ location radians - Float halfPi! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 2/14/2022 13:53:10'!
heading: aHeading
          self rotation: aHeading + Float halfPi! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 10/1/2020 08:34:23'!
mass
          ^ 1! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 1/27/2024 15:47:36'!
starMass
          ^ owner starMass! !

!Mobile methodsFor: 'accessing' stamp: 'hlsf 8/23/2021 11:55:02'!
velocity: aVector
          velocity _ aVector ! !

!Mobile methodsFor: 'computing' stamp: 'hlsf 1/27/2024 15:47:23'!
gravity
"Compute the gravity acceleration vector"
          | position |
          position := self morphPosition.
          ^ [-10 * self mass * self starMass / (position r raisedTo: 3) * position]
                    on: Error do: [0 @ 0]! !

!Mobile methodsFor: 'computing' stamp: 'hlsf 8/23/2021 11:52:21'!
update: t
"Update the mobile position and velocity "
          | ai ag newVelocity |
          "acceleration vectors"
          ai _ acceleration * self direction.
```

```
          ag _ self gravity.
          newVelocity _ (ai + ag) * t + velocity.
          self morphPosition: (0.5 * (ai + ag) * t squared)
                    + (velocity * t)
                    + self morphPosition.
          velocity _ newVelocity.

          "Are we out of screen?
          If so we move the mobile to the other corner and slow it down by a factor of 2"
          (self isInOuterSpace and: [self isGoingOuterSpace]) ifTrue: [
                    velocity _ velocity / 2.
                    self morphPosition: self morphPosition negated]! !

!Mobile methodsFor: 'initialization' stamp: 'hlsf 10/1/2020 08:37:29'!
initialize
          super initialize.
          color _ Color gray.
          velocity _ 0 @ 0.
          acceleration _ 0.! !

!Mobile methodsFor: 'testing' stamp: 'hlsf 10/8/2020 22:21:33'!
isGoingOuterSpace
"is the mobile going crazy in direction of the outer space?"
          ^ (self morphPosition dotProduct: velocity) > 0 ! !

!Mobile methodsFor: 'testing' stamp: 'jmv 10/28/2021 15:04:22'!
isInOuterSpace
"Is the mobile located in the outer space? (outside of the game play area)"
          ^(owner localBounds containsPoint: self morphPosition) not! !

!Mobile methodsFor: 'drawing' stamp: 'hlsf 8/23/2021 11:50:43'!
drawOn: canvas polygon: vertices
          | size |
          size _ vertices size.
          vertices withIndexDo: [: aPoint :i |
                    canvas
                              line: aPoint
                              to: ( vertices at: (i  \\ size + 1) )
                              width: 2
                              color: color] ! !

!Mobile methodsFor: 'geometry' stamp: 'jmv 10/28/2021 14:53:01'!
localBounds

          ^Rectangle encompassing: self class vertices! !

!Mobile class methodsFor: 'as yet unclassified' stamp: 'hlsf 12/19/2020 23:21:09'!
vertices
          ^ vertices ! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 2/14/2022 13:53:33'!
fireTorpedo
"Fire a torpedo in the direction of the ship heading with a combined velocity"
          | torpedo |
          torpedoes isZero ifTrue: [ ^ self].
          torpedoes _ torpedoes - 1.
          torpedo _ Torpedo new.
          torpedo
                    morphPosition: self morphPosition + self nose;
                    heading: self heading;
                    velocity: velocity;
                    color: color muchLighter .
          owner addTorpedo: torpedo.! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 2/14/2022 14:09:51'!
```

```
left
"Rotate the ship to its left"
          self heading: self heading - 0.1! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 1/27/2024 15:39:46'!
push
"Init an accelaration boost"
          fuel isZero ifTrue: [^ self].
          fuel := fuel - 1.
          acceleration := (acceleration + 10) min: 50! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 2/14/2022 14:10:08'!
right
"Rotate the ship to its right"
          self heading: self heading + 0.1! !

!SpaceShip methodsFor: 'control' stamp: 'hlsf 1/27/2024 15:39:56'!
unpush
"Stop the accelaration boost"
          acceleration := acceleration - 5 max: 0! !

!SpaceShip methodsFor: 'drawing' stamp: 'hlsf 1/27/2024 17:42:24'!
drawOn: canvas
          | vertices  |
          vertices := self class vertices.
          super drawOn: canvas polygon: vertices.
          "Draw gas exhaust"
          acceleration ifNotZero: [
                    canvas line: vertices third to: 0@acceleration  width: 1 + acceleration / 8 color: Color

!SpaceShip methodsFor: 'drawing' stamp: 'hlsf 12/15/2020 21:16:09'!
nose
          ^ self direction * 40! !

!SpaceShip methodsFor: 'initialization' stamp: 'hlsf 2/14/2022 13:50:54'!
initialize
          super initialize.
          self resupply! !

!SpaceShip methodsFor: 'initialization' stamp: 'hlsf 2/16/2023 11:54:41'!
resupply
          fuel _ 500.
          torpedoes _ 10! !

!SpaceShip class methodsFor: 'as yet unclassified' stamp: 'hlsf 12/19/2020 23:21:16'!
initialize
"SpaceShip initialize"
          vertices _ {0@-15 . -10@15. 0@10. 10@15}! !

!Torpedo methodsFor: 'computing' stamp: 'hlsf 2/16/2023 11:56:14'!
update: t
"Update the torpedo position"
          super update: t.
          "orientate the torpedo in its velocity direction, nicer effect while inaccurate"
          self heading: (velocity y arcTan: velocity x).
          lifeSpan _ lifeSpan - 1.
          lifeSpan isZero ifTrue: [owner destroyTorpedo: self].
          acceleration > 0 ifTrue: [acceleration _ acceleration - 500].! !

!Torpedo methodsFor: 'drawing' stamp: 'hlsf 12/20/2020 13:19:16'!
drawOn: canvas
          self drawOn: canvas polygon: self class vertices! !

!Torpedo methodsFor: 'initialization' stamp: 'hlsf 12/15/2020 21:12:29'!
initialize
```

```
          super initialize.
          lifeSpan _ 500.
          acceleration _ 3000! !

!Torpedo class methodsFor: 'as yet unclassified' stamp: 'hlsf 12/19/2020 23:21:22'!
initialize
"Torpedo initialize"
          vertices _ {0@-4 . -2@4 . 2@4}! !

!SpaceWar methodsFor: 'accessing' stamp: 'hlsf 9/16/2020 16:41:40'!
starMass
          ^ centralStar mass! !

!SpaceWar methodsFor: 'actions' stamp: 'hlsf 9/21/2020 20:56:33'!
addTorpedo: aTorpedo
          torpedoes add: aTorpedo.
          self addMorph: aTorpedo ! !

!SpaceWar methodsFor: 'actions' stamp: 'hlsf 9/21/2020 21:10:33'!
destroyTorpedo: aTorpedo
          aTorpedo delete.
          torpedoes remove: aTorpedo ! !

!SpaceWar methodsFor: 'actions' stamp: 'hlsf 10/1/2020 17:29:37'!
teleport: aShip
"Teleport a ship at a random location"
          | area randomCoordinate |
          aShip resupply.
          area _ self localBounds insetBy: 20.
          randomCoordinate _ [(area left to: area right) atRandom].
          aShip
                    velocity: 0@0;
                    morphPosition: randomCoordinate value @ randomCoordinate value! !

!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 10/1/2020 15:04:27'!
collisions
          self collisionsShipsStar.
          self collisionsTorpedoesStar.
          self collisionsShipsTorpedoes.
          self collisionsShips ! !

!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:53:43'!
collisionsShips
          (ships first collides: ships second) ifTrue: [
                    ships do: [:each |
                              each flashWith: Color red.
                              self teleport: each] ]! !

!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:28:09'!
collisionsShipsStar
          ships do: [ :aShip |
                    (aShip collides: centralStar) ifTrue: [
                              aShip flashWith: Color red.
                              self teleport: aShip ]]! !

!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:30:08'!
collisionsShipsTorpedoes
          ships do: [:aShip |
                    torpedoes do: [:aTorpedo |
                    (aShip collides: aTorpedo) ifTrue: [
                              aShip flashWith: Color red.
                              aTorpedo flashWith: Color orange.
                              self destroyTorpedo: aTorpedo.
                              self teleport: aShip]]
          ]! !
```

```
!SpaceWar methodsFor: 'collisions' stamp: 'hlsf 2/16/2023 12:30:19'!
collisionsTorpedoesStar
          torpedoes do: [:each |
                    (each collides: centralStar) ifTrue: [
                              each flashWith: Color orange.
                              self destroyTorpedo: each]]! !

!SpaceWar methodsFor: 'drawing' stamp: 'hlsf 10/1/2020 15:45:54'!
drawOn: aCanvas
          aCanvas
                    fillRectangle: self localBounds
                    color: color! !

!SpaceWar methodsFor: 'event handling testing' stamp: 'hlsf 11/1/2020 10:03:57'!
handlesKeyboard
          ^ true! !

!SpaceWar methodsFor: 'event handling testing' stamp: 'hlsf 11/1/2020 10:04:18'!
handlesMouseOver: event
          ^ true! !

!SpaceWar methodsFor: 'events' stamp: 'hlsf 2/22/2023 21:31:48'!
keyStroke: event
          | key |
          event isArrowUp ifTrue: [^ ships first push].
          event isArrowRight ifTrue: [^ ships first right].
          event isArrowLeft ifTrue: [^ ships first left].
          event isArrowDown ifTrue: [^ ships first fireTorpedo].
          key _ event keyCharacter.
          key = $w ifTrue: [^ ships second push].
          key =$d ifTrue: [^ ships second right].
          key = $a ifTrue: [^ ships second left].
          key = $s ifTrue: [^ ships second fireTorpedo]! !

!SpaceWar methodsFor: 'events' stamp: 'hlsf 9/11/2020 10:07:17'!
mouseEnter: evt
          evt hand newKeyboardFocus: self.
          self startStepping! !

!SpaceWar methodsFor: 'events' stamp: 'hlsf 9/11/2020 10:07:22'!
mouseLeave: evt
          evt hand releaseKeyboardFocus: self.
          self stopStepping! !

!SpaceWar methodsFor: 'focus handling' stamp: 'hlsf 9/10/2020 16:04:35'!
keyboardFocusChange: gotFocus
          gotFocus
                    ifTrue: [color _ self defaultColor ]
                    ifFalse: [color _ self defaultColor alpha: 0.5].
          self redrawNeeded.! !

!SpaceWar methodsFor: 'geometry testing' stamp: 'hlsf 12/11/2020 13:00:49'!
clipsSubmorphs
          ^ true! !

!SpaceWar methodsFor: 'initialization' stamp: 'hlsf 9/10/2020 15:21:22'!
defaultColor
          ^ `Color black`! !

!SpaceWar methodsFor: 'initialization' stamp: 'hlsf 11/1/2020 10:04:39'!
initialize
"We want to capture keyboard and mouse events,
start the game loop(step) and initialize the actors."
          super initialize.
```

```
          color _ self defaultColor.
          self startSteppingStepTime: self stepTime.
          self initializeActors.! !

!SpaceWar methodsFor: 'initialization' stamp: 'hlsf 8/23/2021 11:53:32'!
initializeActors
          centralStar _ CentralStar new.
          self addMorph: centralStar.
          centralStar morphPosition: 0@0.
          torpedoes _ OrderedCollection new.
          ships _ Array with: SpaceShip new with: SpaceShip new.
          self addAllMorphs: ships.
          ships first
                    morphPosition: 200 @ -200;
                    color: Color green darker.
          ships second
                    morphPosition: -200 @ 200;
                    color: Color red darker.! !

!SpaceWar methodsFor: 'stepping' stamp: 'hlsf 12/16/2020 19:45:30'!
stepAt: millisecondSinceLast
          ships do: [:each | each update: millisecondSinceLast / 1000 ].
          ships do: [:each | each unpush ].
          torpedoes do: [:each | each update: millisecondSinceLast / 1000 ].
          self collisions.
          self redrawNeeded
! !

!SpaceWar methodsFor: 'stepping' stamp: 'hlsf 9/11/2020 13:19:07'!
stepTime
"millisecond"
          ^ 20! !

!SpaceWar methodsFor: 'stepping' stamp: 'hlsf 9/10/2020 18:03:59'!
wantsSteps
          ^ true! !

!SpaceWar methodsFor: 'geometry' stamp: 'hlsf 2/16/2023 11:38:27'!
localBounds

          ^ -300 @ -300 extent: 600@600! !

!CentralStar methodsFor: 'accessing' stamp: 'hlsf 10/8/2020 22:20:17'!
mass
          ^ 8000! !

!CentralStar methodsFor: 'accessing' stamp: 'hlsf 12/15/2020 21:04:13'!
morphExtent
          ^ `30 @ 30`! !

!CentralStar methodsFor: 'drawing' stamp: 'KenD 2/15/2023 15:40:04'!
drawOn: canvas
          | radius |
          radius _ self morphExtent // 2.
          canvas ellipseCenter: 0 @ 0
                    radius: (radius x + (2 atRandom - 1)) @ (radius y + (2 atRandom - 1))
                    borderWidth: 3
                    borderColor: Color orange
                    fillColor: Color yellow ! !

!CentralStar methodsFor: 'geometry' stamp: 'jmv 10/28/2021 14:55:55'!
localBounds

          ^Rectangle center: 0@0 extent: self morphExtent! !
SpaceShip initialize!
```

```
Torpedo initialize!
```

Apéndice H Índice temático

•

`.pck.st` 133
`.st` 133

A

archivo 136
archivo extensión,
  `.st` 133
  `.st.pck` 133
`Array` 49, 56
array,
  dinámico 50, 150
  estático 150
  operación 50
  size 50
  statistic 50
  tamaño 50
assignment, *Véase* variable
atajo, *Véase* atajo de teclado
Atajo de teclado,
  buscar una clase (*Ctrl-f*) 21
atajo de teclado,
  completar código (*tab*) 32
  ejecutar código (*Ctrl-d*) 9
  guardar código (*Ctrl-s*) 22
  ojear jerarquía (*Ctrl-h*) 46
  ojear protocolo (*Ctrl-p*) 46
  ojear una clase (*Ctrl-b*) 21
  seleccionar todo el código (*Ctrl-a*) 9
atajo de telado,
  ejecutar e imprimir resultado (*Ctrl-p*) 10
atajo de telcado,
  implementadores de (*Ctrl-m*) 30

B

backtick 103
block,
  parameter 52
bloque 52, 66, 151
  asignar a una variable 66
  `ensure:` 134
  parámetro 66
  variable local 66
boolean 67
breakpoint, *Véase* Herramientas, debugger
browser 19
  categoría de clases 20
  categoría de clases (nueva) 22
  invocar desde Workspace 21
  jerarquía 46
  protocolo 46
bucle 69
  for 34, 55
    paso (step) 55
    step 34
  repetir 34, 55
bucle for, *Véase* loop

C

cadena, *Véase* string
cadena 150
  `asArray` 18
  entrada de fichero 136
  file entry 136
  `shuffled` 17
  `sorted` 18
cadena de mensajes 151
carácter 38, 150
  ascii 17
  unicode 17
cascada de mensajes 16, 151
change log 118
change set 120
cifrado Caesar 54
clase 14, 26
  abstracta 56
  categoría 20, 21, 47, 123
  categoría (nueva) 22
  `class` 108
  comentario 21
  crear (nueva) 22
  declaración 20
  herencia 27, 46
  inicializar 107
  método, *Véase* método
  protocolo 46
  variable, *Véase* variable
  variable de instancia, *Véase* variable
colección 56
  acceder elemento 57
  `add:` 51
  `at:` 51
  `collect:` 54, 55
  convertir 58
  `Dictionary` 59
  dinámica 51
  `indexOf:` 51
  `inject:into:` 14
  instanciar array 56
  instanciar arrays de tamaño variable 58
  `last` 51
  mecanismo enumerador 52
  operaciones de conjunto (union, intersection, difference) 51
  `OrderedCollection` 51
  `pairsDo:` 56
  `select:` 52
  `Set` 58
  `shuffled` 17
  tamaño fijo 56
  tamaño variable 57
collection,
  `squared` 50
comentario 150

control de flujo,
bucle ... 69
control flow ... 67
test ... 67
coordenadas ... 41

D

debugger, *Véase* herramientas
decimal (ver número) ... 150
desplazamiento de bits ... 36
devolver valor ... 29
`Dictionary` ... 59

E

entero, *Véase* número
entero (ver número) ... 150
event,
handling ... 83
evento,
clases ... 112
comprobar ... 82
manejo (handling) ... 113
mouse enter ... 114
mouse-enter ... 115
teclado ... 116
prueba (testing) ... 113
mouse over ... 114
teclado ... 116
ratón ... 114
teclado ... 116
excepciones ... 134
exception ... 134
execution stack ... 137

F

`false` ... 64
file ... 136
fracción, *Véase* número

G

garbage collection ... 14
recolección de basura ... 14

H

`halt` ... 139
herramientas,
browser del sistema ... 19
cambios perdidos ... 119
cambios recientes ... 120
debugger ... 135
breakpoint ... 139
inspector ... 78
transcript ... 9
workspace ... 8

I

`initialize` ... 45
inspector, *Véase* herramientas
instancia ... 14, 26
creación ... 27
método, *Véase* método
`Interval` ... 35

L

línea de comandos opción,
-s (correr un script) ... 130
literal,
número ... 12

M

método ... 14
categoría ... 21, 34
crear ... 70
devolver valor (explícito) ... 29
devolver valor (implícito) ... 30
método de clase ... 27, 31
método de instancia ... 21, 29, 32
overriding (sobreescritura) ... 29
variable, *Véase* variable
mensaje,
binario ... 15, 151
cascada ... 16
composición ... 18
emisor ... 14
envío ... 15
getter ... 42
palabra clave ... 15, 151
precedencia ... 15
receptor ... 14
setter ... 42
unario ... 151
unitario ... 15
method ... 65
category ... 39
morph,
animado ... 95
`clipsSubmorphs` ... 97
`delete` ... 92
`drawOn:` ... 91, 93, 98, 103
ellipse ... 75
halo ... 75
legado ... 91
`location` ... 93, 105
móvil ... 92
`morphPosition` ... 95
move/pick up ... 79
propiedades ... 79
rectangle ... 76
`rotateBy:` ... 95
`rotation:` ... 105
`rotationCenter` ... 95
`step,wantsSteps` ... 95
subclase ... 82
submorph ... 76, 96

vector,
área de relleno ... 93
line drawing ... 91
world ... 131

N

número,
abs ... 29
conversión ... 38
decimal ... 37, 150
división decimal ... 18
entero ... 10, 150
atRandom ... 35, 67, 103
base ... 11, 36
división ... 19
en palabras ... 11
even ... 19
gcd: (máximo común divisor) ... 19
isDivisibleBy: ... 19
isPrime ... 19
lcm: (mínimo común múltiplo) ... 19
odd ... 19
resto de la división ... 19
romano ... 11
timesRepeat: ... 34
intervalo ... 35
literal ... 12
raíz ... 18
racional (fracción) ... 10, 26, 37
operaciones ... 18
roundTo: ... 34
roundUpTo: ... 34
sqrt ... 18
squared ... 18, 29
to: ... 35
to:by: ... 35
to:do ... 55
to:do: ... 34
to:do:by ... 34
to:do:by: ... 55
nil ... 64

O

OrderedCollection ... 58
overriding ... 29, 88

P

package, *Véase* paquete
package ... 123
paquete ... 123
ampliación del sistema ... 127
cargar,
mediante código ... 74
mediante interfaz gráfica ... 24
crear (nuevo) ... 23
guardar ... 23
herramienta ... 23
prefijo ... 126
requisito ... 124
pila de ejecución ... 137
Point ... 41
polimorfismo ... 29
polymorphism ... 27
primitiva ... 151
protocolo ... 46
pseudovariable,
false ... 64
nil ... 64
self ... 64
super ... 64
thisContext ... 64
true ... 64
pseudovariables ... 64

R

racional (fracción), *Véase* número
ratón ... 114
receptor ... 149
refactorizar ... 85, 142
repetir, *Véase* bucle
returned value ... 2

S

símbolo ... 48, 150
script de inicio ... 130
secuencia ... 151
secuencia de Fibonacci ... 55
selector ... 26
self ... 64
Set ... 58
sobreescribir ... 88
string ... 10, 39
acceso a caracteres ... 16
asUppercase ... 10
at:put: ... 17
capitalized ... 10
concatenar ... 10
indexOf: ... 17
unicode ... 16
subclase ... 26
super ... 45, 64
superclase ... 27

T

test ... 67
thisContext ... 64
true ... 64

U

unicode ... 16, 17, 38

V

valor devuelto ... 151
variable ... 48
:= ... 43
← ... 43
asignación ... 43, 150
assignment ... 2
clase ... 109
compartida ... 149
instancia ... 26
instancia de clase ... 106
local ... 149
declaración ... 150
método ... 48
variable de instancia ... 26

Y

`yourself` ... 17