- BORRADOR FINAL -

7 de Febrero de 2006

Programando con Smalltalk

Autor: Diego Gómez Deck

Editorial: Edit Lin Editorial S.L.

PONER LA LICENCIA CREATIVE COMMONS ELEGIDA.

Se permite la copia y distribución de la totalidad o parte de esta obra sin ánimo de lucro. Toda copia total o parcial deberá citar expresamente el nombre del autor, nombre de la editorial e incluir esta misma licencia, añadiendo, si es copia literal, la mención "copia literal".

Se autoriza la modificación y traducción de la obra sin ánimo de lucro siempre que se haga constar en la obra resultante de la modificación el nombre de la obra originaria, el autor de la obra originaria y el nombre de la editorial. La obra resultante también será libremente reproducida, distribuida, comunicada al público y transformada en términos similares a los expresados en esta licencia.

A todos aquellos que saben que la verdadera revolución de la información no ocurrirá hasta que seamos capaces de romper unas cuantas barreras.

Índice

Prólogo (por Alan Kay) 6

Introducción 7
- Enfoque 7
- Metodología 8
- Alcance del libro 8
- Audiencia 8
- Licencia 9
- Sitio web 9
- Agradecimientos 9

¿Qué es Smalltalk? 10
- Conceptos 11
  - Objetos 11
  - Mensajes 11
  - Interfaz 11
  - Encapsulación 12
  - Polimorfismo 12
  - Clases 12
  - Variables de Instancia 12
  - Métodos 12
  - Herencia 13
  - Programar es simular 13
- Historia del Smalltalk 13
  - Aportes del Smalltalk 14
- ¿Qué es Squeak? 16

Programando con Smalltalk 18
- La curva de aprendizaje 18
- Prepararse para un shock cultural 18
  - Tirar código 18
  - Trabajo incremental 18
  - No hay archivos fuentes 18
- El camino es largo, mejor no ir solo 18
- Sintaxis 20
  - Literales 20
  - Mensajes 22
  - Precedencia 24
  - Variables 25
  - Los nombres de clases son, también, variables globales 26
  - Bloques 27
  - Comentarios 30
  - Método de Ejemplo 30
- Herramientas 31
- Librería de Clases y Frameworks 31
- Máquina Virtual 32

¡Manos a la Obra! 33
- Modificando Objetos Vivos 34
  - El Mundo 34
  - Browser de Clases 36
  - Tipos de Browser de Clases 36
  - Categorías de Clase y Métodos 37
  - Squeak y el Ratón 38
  - Foco de teclado 39
  - Workspace 42
  - Tipos de Workspace 42
  - Evaluando Código 43
  - Sentencias de Ejemplo para evaluar, imprimir, inspeccionar o explorar 44
  - Inspector 45
  - Hot-Keys 46
- Parser de XML basado en una Pila 49
  - Smalltalk con Estilo 50
  - NombreDeClase>>nombreDeMétodo 50
- Importador de Wikipedia 55
  - Wikipedia 55
  - Métodos de Clase vs. Métodos de Instancia 57
  - Plantilla para nuevos métodos 62
  - Archivos de ejemplo 63
  - Pre-depurador 64
  - Depurador 66
  - Depurador 100% en Smalltalk 68
  - Convención de nombres 73
  - Colecciones 84
  - Colecciones – Set 85
  - Valor de retorno por defecto 86
  - Colecciones – mensaje #add: 87
  - Explorador 89
  - Colecciones – OrderedCollection 98
  - Inicialización de objetos 110
  - MessageTally 112
  - TimeProfilerBrowser 115
- Motor de Workflow 117
  - Workflow 117
  - Test Driven Development 117
  - SUnit 118
  - Refactoring Browser 119
  - Refactoring 119
  - Los métodos de testing comienzan por #test 120
  - Consejo: Pensar primero en la interfaz pública 121
  - Estructura de los test 121
  - SUnit Test Runner 123
  - Colecciones – mensaje #collect: 130
  - Consejo: Es más barato escribir código limpio 132
  - Explaining Temporary Variable 132
  - Colecciones – mensaje #, 135
  - Consejo: Probar las situaciones límite 139

Mensaje #halt 145
Consejo – El depurador nos brinda más información a la hora de implementar 146
Colecciones – mensaje #includes: 147
Senders 150
Los tests son, también, documentación 159
Colecciones – mensaje #anySatisfy: 161
Composed Method 162
Los tests aumentan la confianza 168
Colecciones – mensaje #select:thenCollect: 170
Colecciones – mensaje #select: 171
Colecciones – mensaje #at:ifAbsentPut: 172
Colecciones – Dictionary 173
Transcript 173
Patrón de Diseño – Adapter 188
Métodos privados 190
Browser Jerárquico 196
Usar el fuente de un método para crear otro método parecido 199
Mensaje #subclassResponsibility 201

La yapa 203

Futuro 212
Traits 212
Tweak 212
64 bits 212
OpenCroquet 212

Como continuar 214
Libros 214
Papers o artículos 214
Grupos de Usuarios 215

Conclusión 216

Bibliografía 217

Herramientas usadas en el libro 221

Prólogo (por Alan Kay)

Llega el 20/Enero.

Introducción

Smalltalk es mucho más que un lenguaje de computación.

Smalltalk es uno de los productos de un proceso de investigación y desarrollo, liderado por Alan Kay, que ya lleva más de 30 años tratando de inventar la computadora personal. Smalltalk, además de ser un lenguaje de computación muy poderoso y versátil, es una interpretación de como debieran utilizarse las computadoras: Las computadoras deberían ser herramientas que sirvan para amplificar el espíritu creativo de las personas.

Smalltalk no es la culminación de esa visión, es sólo un paso. Si Smalltalk tiene un objetivo en si mismo, ese es el de servir de herramienta para crear el producto que vuelva obsoleto al mismo Smalltalk.

Lamentablemente las ideas detrás de Smalltalk no son las más extendidas en el mundo de la informática, y aunque muchas innovaciones del proyecto llegaron a nuestros días, estas llegaron despojadas de lo fundamental y sólo se copió la cascara de las ideas.

Eso hace que la primera experiencia con Smalltalk, para una persona acostumbrada a usar computadoras o incluso a programarlas, sea desconcertante por lo distinto que resulta.

De todas formas, el esfuerzo de entrar en un área desconocido tiene sus recompensas. Muchos somos los que sentimos que Smalltalk nos devolvió la fascinación que nos produjo usar una computadora por primera vez. No es raro ver como gente que comienza a programar, con Smalltalk, cambia su concepción de los porqués de las computadoras. Prácticamente todos los programadores de Smalltalk que yo conozco utilizan el ambiente, no sólo para trabajar y producir software, sino que también lo utilizan como una herramienta de investigación. Es muy común encontrarse con proyectos muy “raros” e innovadores desarrollados con Smalltalk, porque justamente, Smalltalk sirve como amplificador del espíritu creativo que todos llevamos dentro.

Enfoque

Este libro pretende mostrar a personas que ya sepan programación, como el hecho de vivir en un ambiente de objetos, impacta en el ciclo de desarrollo de software.

La programación con Smalltalk es muy diferente al clásico ciclo edición/compilación/ejecución que domina a la mayoría de las herramientas usadas hoy en día. Con Smalltalk se da un proceso mucho más interactivo. No se penaliza la investigación ni se castiga más fuertemente las primeras decisiones. Al minimizar el costo de los cambios, se pueden posponer las decisiones más importantes del diseño hasta el momento que sepamos lo suficiente. Se promueve un método de desarrollo donde el software cambia conforme las personas que lo desarrollan lo hacen.

Este libro pretende mostrar algo de ese proceso; para eso desarrollaremos algunas aplicaciones de ejemplo paso a paso e iremos introduciendo conceptos, descripciones, etc. conforme lo necesitemos para el desarrollo de esos ejemplos. Los ejemplos son una escusa para mostrar el uso del ambiente de Smalltalk, no son objetivos en si mismos y no se van a desarrollar al 100%.

Metodología

Aunque todos los programadores Smalltalk compartimos ciertas costumbres, programar con Smalltalk es una tarea muy personal y cada programador configura y usa el ambiente de forma diferente. No es de extrañar que la programación se convierta en una tarea tan personal cuando se desarrolla en un entorno nacido como la concepción de la computadora personal.

En este libro les voy a mostrar como yo uso el ambiente. Eso no quiere decir que yo crea que esta es la única o la mejor forma. El método que les muestro es sólo el que a mi más me gusta y no pretendo imponer mi forma de trabajo ni nada que se le parezca. Lo que si deseo es que cada uno de ustedes use al Smalltalk de la forma que más le guste, y que compartan con la comunidad las mejoras que hagan en sus ambientes.

Probablemente reconocerán algunas cosas de las, ahora denominadas, metodologías ligeras; y eso no debería ser de extrañar ya que la principal metodología ágil, que es la Extremme Programming (Programación Extrema), se formalizó en un proyecto desarrollado con Smalltalk donde Kent Beck era parte.

Alcance del libro

Este libro no es una guía completa de Smalltalk, sólo pretende mostrar la filosofía de uso del ambiente y ser un punto de entrada a personas que deseen aprender Smalltalk.

La mayoría de las respuestas, a las preguntas que nos hacemos cuando programamos en Smalltalk, están dentro del mismo ambiente. Este libro pretende enseñar a buscar las respuestas en el ambiente en lugar de contestarlas directamente.

En el capítulo de bibliografía incluyo una lista de libros de Smalltalk que pueden ser de complemento a este libro. Muchos de esos libros pueden ser bajados de forma gratuita de Internet.

Audiencia

Este libro no pretende enseñar programación sino que se asume que se sabe programar y que, incluso, se está familiarizado con los conceptos de la programación orientada a objetos.

Este libro es ideal (mejor dicho: pretende serlo) para programadores que estén buscando nuevas y mejores formas de producir software y que crean que Smalltalk puede ser una opción. Incluso, aunque no se tenga oportunidad de usar Smalltalk en el día a día, aprender Smalltalk y el paradigma de objetos tal y cual fue creado los convertirá en mejores programadores.

Licencia

Pretendimos ser fieles al ideal de completa libertad que rigió el desarrollo del proyecto Smalltalk y escogimos una licencia que permite la libre circulación de los contenidos de este libro.

PONER LA LICENCIA CREATIVE COMMONS ELEGIDA.

Se permite la copia y distribución de la totalidad o parte de esta obra sin ánimo de lucro. Toda copia total o parcial deberá citar expresamente el nombre del autor, nombre de la editorial e incluir esta misma licencia, añadiendo, si es copia literal, la mención "copia literal".

Se autoriza la modificación y traducción de la obra sin ánimo de lucro siempre que se haga constar en la obra resultante de la modificación el nombre de la obra originaria, el autor de la obra originaria y el nombre de la editorial. La obra resultante también será libremente reproducida, distribuida, comunicada al público y transformada en términos similares a los expresados en esta licencia.

Sitio web

El enfoque de este libro hace que sea prioritario reducir al mínimo el tiempo de espera antes de meterse en el ambiente. La experiencia de primera mano en el ambiente es fundamental para aprender a usar Smalltalk.

Para acelerar el proceso inicial hice una imagen de Squeak pre-configurada con las herramientas más habituales a la hora de programar. En uno de los capítulos finales se explica en detalle que cosas se instalaron y cambiaron de un Squeak 'virgen'.

Pueden encontrar la imagen, como así también otras cosas relacionadas con este libro, en el wiki:

http://smalltalk.consultar.com

Agradecimientos

Disculpen mi falta de originalidad, pero tengo que agradecer especialmente a mi núcleo familiar por el soporte y comprensión que me han brindado. Mil gracias a mi compañera de viaje Raquel y a mis hijos Nicolás y Nahuel; ellos son los que realmente han hecho sacrificios para que este libro llegue a buen término.

¿Qué es Smalltalk?

Es muy difícil explicar, a personas que no lo conozcan, que es Smalltalk y porque es tan diferente a otras herramientas más populares.

El origen de ese des-entendimiento viene a razón de que el proyecto Smalltalk estuvo - y sigue estando - regido por una concepción diferente de como debieran usarse las computadoras:

Las computadoras deben ser herramientas que sirvan como amplificadores del espíritu creativo de las personas.

Eso implica, entre otras cosas, que:

- Todo el sistema tiene que ser entendible, y por ende modificable, por una sola persona.
- El sistema tiene que estar construido con un mínimo juego de partes intercambiables. Cada parte puede ser cambiada sin que el resto del sistema se modifique.
- Todo el sistema tiene que estar construido basado en una metáfora que pueda ser aplicada en todas las partes.
- Cada componente del sistema tiene que poder ser inspeccionado y cambiado.

Smalltalk es un ambiente de objetos. Los objetos interactúan enviándose mensajes entre si.

Smalltalk incluye un lenguaje de programación y este fue desarrollado usando los principios que ya se enumeraron. Esto quiere decir, por ejemplo, que el lenguaje Smalltalk puede ser cambiado - como cualquier otra parte - y reemplazado por otro; o que pueden convivir diferentes lenguajes de programación en el mismo ambientes, etc.

Un ambiente de Smalltalk típico también incluye muchas herramientas que asisten en la tarea de programación. Pero estas, como todo, también pueden ser modificadas o reemplazadas por otras.

A su vez, un ambiente de Smalltalk, incluye objetos para crear interfaces de usuarios gráficas. Todo escrito en Smalltalk, todo a la vista y todo modificable.

Este es un libro de programación para programadores, así que nos vamos a centrar en las capacidades de un ambiente de Smalltalk para la creación de software. Sin embargo tenemos que saber que las ideas detrás de Smalltalk son lo suficientemente poderosas como para poder hacer un sistema operativo completo. Estas ideas tienen el potencial para convertir el uso de los ordenadores en una actividad mucho más creativa que la experiencia que podemos obtener con el software que

usamos normalmente.

Centrándonos en el punto de vista de meros programadores mortales, podemos decir que Smalltalk:

- Incluye un lenguaje de programación.
- Incluye una librería de clases
- Incluye un entorno de desarrollo

Tanto el lenguaje, como la librería de clases y el entorno de desarrollo llevan más de 30 años de uso y depuración. No en vano Smalltalk ha sido la referencia para el desarrollo de lenguajes y entornos de desarrollo.

Conceptos

Smalltalk está definido con un conjunto muy pequeño de conceptos pero son un significado muy específico. Es fácil confundir los conceptos porque estos son usados, en diferentes lenguajes, con definiciones muy diferentes.

Objetos

Todo en Smalltalk es un *objeto*. Un objeto es una parte indistinguible del resto del ambiente, con características y responsabilidades bien demarcadas.

Mensajes

Un *mensaje* es un requerimiento, que se hace a un objeto determinado, para que este lleve a cabo algunas de sus responsabilidades.

Un mensaje especifica que es lo que se espera del *receptor* (el objeto que recibe el mensaje) pero no fuerza una determinada forma de resolverlo. El receptor es el responsable de decidir como se lleva a cabo la operación para responder al mensaje.

Interfaz

El juego de mensajes que un determinado objeto puede entender se denomina su *interfaz*. La única forma de interactuar con un objeto es a través de su interfaz.

Encapsulación

Se llama *encapsulación* al hecho que ningún objeto puede acceder a la estructura interna de otro objeto. Sólo el objeto conoce, y puede manipular, su propia estructura interna.

Ese hecho, sumado a que los mensajes sólo especifican que se espera del receptor pero no especifica como se debe realizar la tarea, aseguran que ningún objeto dependa de la estructura interna de otro.

El envío de mensajes sumado a la encapsulación permiten el desarrollo de sistemas muy *modulares* ya que cualquier parte puede ser reemplazada por otra mientras la nueva parte respete la interfaz de la parte reemplazada.

Polimorfismo

Dos objetos son *polimórficos* entre sí cuando un determinado emisor no puede distinguir a uno del otro. Dicho de otra forma: si podemos intercambiar un objeto por otro, en un determinado contexto, es porque son polimórficos.

Clases

Una *clase* describe la implementación de un conjunto de objetos. Los objetos individuales, descritos por las clases, se llaman *instancias*. La clase describe la estructura interna de los objetos y describe, también, como se responde a los mensajes.

Variables de Instancia

La estructura interna de los objetos está compuesta por *variables de instancia*. La variables de instancia son nombres que el objeto puede usar para hacer referencia a otros objetos.

Métodos

Los *métodos* son la forma de especificar como los objetos de una determinada clase responden a los mensajes. Cada método especifica como se lleva a cabo la operación para responder a un determinado mensaje. Un método puede acceder a la estructura interna del objeto como así también enviar mensajes a si mismo o a otros objetos. Los métodos especifican, también, cual es la respuesta que el *emisor* (el objeto que envía el mensaje) recibe.

Todos los objetos de Smalltalk son instancia de alguna clase. La programación en Smalltalk consiste en crear clases, crear instancias de esas clases y especificar la secuencia de envío de mensajes entre esos objetos.

Herencia

Las clases, en Smalltalk, están organizadas jerárquicamente. Una clase refina el concepto de otra clase más abstracta. La clase más abstracta es `Object`. Todas las clases son herencias de `Object` (porque todo es un objeto) o herencia de alguna clase que hereda de `Object`.

La relación que existe entre la clase más abstracta (la *superclase*) y la clase más concreta (la *subclase*) permite clasificar el conocimiento que tengamos del dominio modelado.

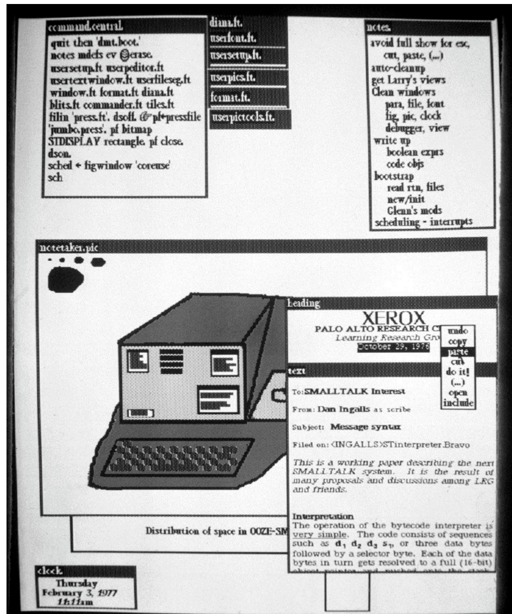
Programar es simular

Programar con objetos es programar una simulación.

La metáfora en la programación con objetos está basada en personificar a un objeto físico o conceptual del dominio real en objetos simulados del ambiente. Tratamos de reencarnar los objetos reales en el ambiente dotándolos de las mismas características y funcionalidades que los objetos reales a los que representan.

Historia del Smalltalk

Smalltalk (no se escribe SmallTalk, ni Small-talk, ni Small-Talk) es un proyecto que lleva más de 30 años de desarrollo. Fue inventado por un grupo de investigadores liderados por Alan Kay, en Xerox PARC (Palo Alto Research Center), durante los años 70s. El proyecto de investigación produjo varios resultados intermedios que fueron conocidos como Smalltalk/71, Smalltalk/72, Smalltalk/76 y Smalltalk/80. (http://en.wikipedia.org/wiki/Smalltalk)



Smalltalk en el año 1977

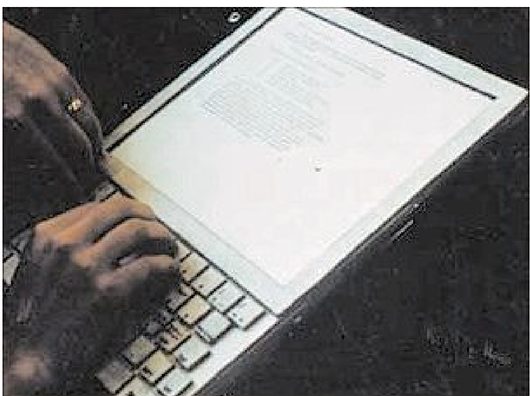
El Smalltalk usado hoy en día es un descendiente directo del Smalltalk/80 y conserva prácticamente todas sus características.

El proyecto ha generado una cantidad muy grande de invenciones que llegaron a nuestros días.

Aportes del Smalltalk

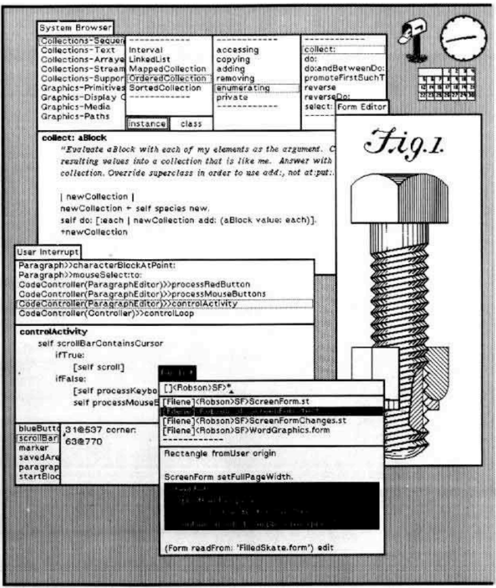
Por razones que desconozco el proyecto Smalltalk no cuenta con la reputación y la fama que se merece. Es muy frecuente encontrarse con profesionales de la informática que no saben del proyecto ni de sus aportes. Sin embargo utilizamos a diario varias ideas que fueron desarrolladas en el proyecto Smalltalk.

Computadora Personal: La idea de que cada persona pueda disponer de una computadora de uso personal, con una potencia de cálculo suficiente para ser la herramienta de acceso a la información, tiene sus raíces en las ideas de la Dynabook de Alan Kay (http://en.wikipedia.org/wiki/Dynabook).



Maqueta de la Dynabook

Interfaces gráficas de usuario: Smalltalk, desde sus inicios, contó con una rica interfaz de usuario gráfica manejada por un ratón. Las ventanas superpuestas, las barras de desplazamiento, el copiar y pegar, los menús de contexto, etc son todos desarrollos del proyecto Smalltalk que años más tarde Apple ayudó a masificar.



Smalltalk/80

Informática: Son innumerables la cantidad de aportes que el Smalltalk ha hecho a la informática. Empezando con el paradigma de orientación a objetos y terminando con cosas muy técnicas como la operación BitBlt (http://en.wikipedia.org/wiki/Bit_blit). La lista continua con las metodologías ágiles (http://en.wikipedia.org/wiki/Extreme_Programming), los patrones de diseño (http://en.wikipedia.org/wiki/Design_Patterns), el concepto de entorno de desarrollo, etc.

El proyecto Smalltalk también ayudó a afianzar el uso de los lenguajes dinámicos, los recolectores de basura, el uso de máquinas virtuales, etc.

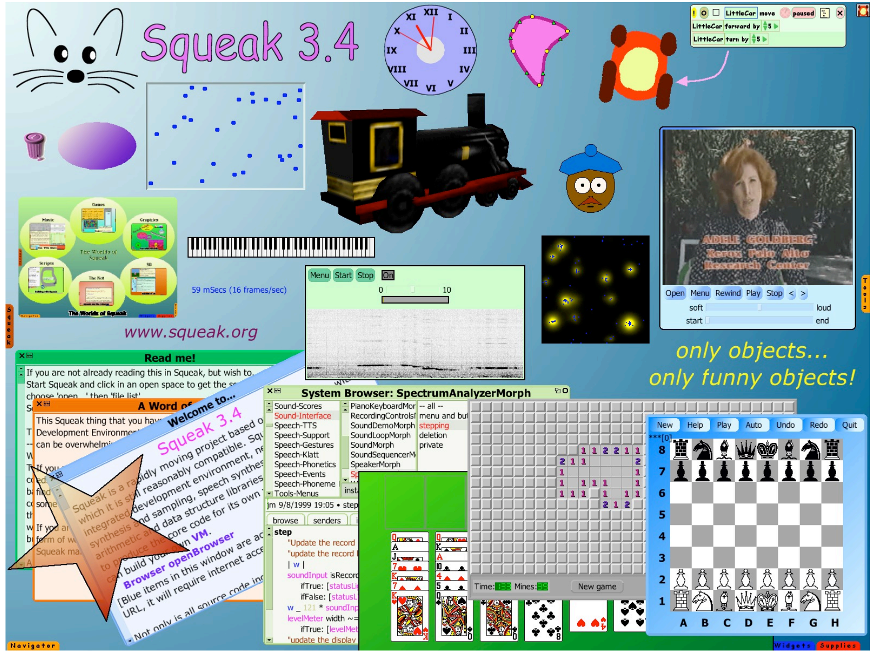
La lista continua: Unit Testing (http://en.wikipedia.org/wiki/Unit_test), Refactoring y Refactoring Browser (http://en.wikipedia.org/wiki/Refactoring), etc.

¿Qué es Squeak?

Squeak es un Smalltalk moderno, de código abierto, escrito en si mismo, muy portable y muy rápido. A la fecha Squeak puede correr en plataformas Windows, Apple, Linux, iPaqs, etc.

Squeak es la continuación del proyecto original de Smalltalk y cuenta con los aportes de varios de los desarrolladores originales de los años 70s. Entre ellos Alan Kay, Dan Ingalls, Ted Kaehler, etc.

La comunidad de desarrolladores y usuarios de Squeak se extiende por todo el mundo y cubre un abanico muy grande de usos, entre ellos: educación, investigación, desarrollo multimedia, desarrollo web, etc.



Squeak versión 3.4

El proyecto Squeak comenzó en Apple, en el año 1995, porque los autores necesitaban un entorno de desarrollo de software educativo que pueda se usado – e incluso programado – por personas no técnicas. Los detalles de la motivación que arrancó con el proyecto Squeak pueden leerse en el paper *"Back to the Future – The Story of Squeak, a practical Smalltalk written in itself"* (ver bibliografía).

Squeak, en cierta forma, es un regreso a las ideas que habían motivado todo el desarrollo del proyecto Smalltalk en los 70s. Algunos de los dialectos comerciales de Smalltalk han perdido parte de la concepción original y Squeak vuelve a tomar esos valores. Todo lo que pueda hacerse con una computadora tiene que poder hacerse con Squeak. Aplicar ese concepto convierte a Squeak en

un ambiente muy multimedia donde todos los medios de expresión y soporte de información pueden confluir en una única herramienta. Squeak cuenta con excelente soporte para gráficos 2D y 3D, procesamiento de texto y numérico, música (MP3, MIDI, generador de sonido FM), procesamiento de vídeo, etc.

Squeak también dispone de excelentes herramientas para la programación como el Refactoring Browser, editores de código con sintax-highlight, framework para unit-testing, etc.

Y cuenta con excelentes opciones para desarrollo de software comercial como Seaside (un frameworks de desarrollo de aplicaciones web basado en continuations), acceso a bases de datos ODBC, bases de objetos como Magma, rST – Remote Smalltalk (soporte de objetos distribuidos), etc.

Existen, a fines de enero de 2006, más de 600 paquetes con aplicaciones para instalar en Squeak registradas en SqueakMap (http://map1.squeakfoundation.org/sm).

Squeak es el dialecto de Smalltalk escogido para este libro. Sin embargo hay que saber que el 95% de lo que se aprenda en un determinado dialecto vale para el resto de los dialectos. Si Squeak no es la herramienta de su preferencia, puede probar alguna de las varias alternativas que Smalltalk brinda.

Programando con Smalltalk

Comenzar con Smalltalk no es fácil para personas que venimos de otros entornos. Antes de meternos de lleno con cosas como la sintaxis, estos son algunos consejos y comentarios que pretenden facilitar el acceso a Smalltalk.

La curva de aprendizaje

Prepárese para una curva de aprendizaje alta al comienzo. Smalltalk tiene una visión diferente de como debiera ser la experiencia con los ordenadores, esa diferencia de concepción hace que los inicios con Smalltalk sean un poco desconcertante para personas que ya tengan experiencia en informática.

Prepararse para un shock cultural

La mayoría de los desentendimientos iniciales tienen que ver con una forma diferente de ver y entender la informática. Smalltalk no es difícil, sólo es diferente.

Tirar código

El código escrito es importante, pero mucho más importante es el conocimiento que vamos obteniendo conforme programamos. Si aprendemos día a día, probablemente el código viejo no sea bueno. Tirar código no es malo, lo malo es no aprender a diario.

Trabajo incremental

Un ambiente como Smalltalk es ideal para trabajar de forma incremental. Los vicios que tenemos, producto del ciclo de desarrollo edición/compilación/prueba, no tienen sentido en Smalltalk y, en cierta forma, tenemos que aprender a programar de otra forma.

Por otro lado el costo de los cambios es mucho más bajo en Smalltalk que en otros lenguajes, eso permite relajarse y demorar las decisiones importantes de diseño hasta que sepamos suficiente del dominio. No es de extrañarse que las metodologías ágiles hayan sido desarrolladas en Smalltalk.

No hay archivos fuentes

En Smalltalk no hay un archivo con los fuentes y todo, incluso el código, está dentro del ambiente de objetos. Esto no quiere decir que no se pueda mover código entre imágenes de Smalltalk y existen distintas opciones para mover código de un Smalltalk a otro.

El camino es largo, mejor no ir solo

Quizás la forma más fácil de aprender Smalltalk sea con un tutor. Se que es difícil conseguir un

 Programando con Smalltalk

tutor a tiempo completo y de forma presencial, pero hay listas de correos y grupos de usuarios que pueden ayudar bastante en los comienzos.

 Programando con Smalltalk

Sintaxis

Prácticamente todo el paradigma de objetos se puede resumir en: Objetos que reciben mensajes. La sintaxis Smalltalk es una directa consecuencia de eso. La estructura básica de la sintaxis es:

```
objeto mensaje.
```

Literales

Algunos objetos son conocidos por el compilador y pueden ser instanciados con literales.

Números:

Smalltalk cuenta con una rica variedad de objetos numéricos. Enteros (`SmallInteger`, `LargePositiveInteger` y `LargeNegativeInteger`), Coma flotante (`Float`), Fracciones (`Fraction`), Decimales (`ScaledDecimal`), etc. Algunos de esos objetos puede instanciarse usando los siguientes literales.

```
"Enteros"
1.
-1.
12345678901234567890.
-12345678901234567890.

"Coma flotante"
1.1.
-1.1.
12345678901234567890.0.
-12345678901234567890.0.
```

Caracteres:

Los caracteres (`Character`) pueden instanciarse, también, usando literales.

```
"Caracteres"
$a.
$b.
$á.
$1.
```

 Programando con Smalltalk

```
$$.
```

Cadenas de Caracteres:

Las cadenas (`String`) son una secuencia de caracteres.

```
"Cadena de Caracteres"
'!Hola mundo!'.
'Smalltalk'.
'áéíóú'.
'Un string con una comilla simple ('')'
```

Símbolos:

Los símbolos (`Symbol`) son cadenas de caracteres (`String`) usadas por el sistema como nombre de clase, métodos, etc. Nunca habrá, en todo el sistema, 2 símbolos con los mismos caracteres, eso permite comparaciones muy rápidas.

```
"Símbolos"
#unSímbolo.
#'un símbolo con espacios'.
```

Array:

Un `Array` es una estructura de datos simple que permite acceder a los elementos contenidos indicando la posición con un número. Se pueden crear Arrays de literales con un literal. El literal para crear un array es una secuencia de literales encerradas entre `#(` y `)`.

```
"Array"
#(1 2 3 4).
#(1 1.0 $a 'un string' #unSímbolo).
#(#(1) #(2)).
```

En Squeak (y no en otros dialectos) se puede instanciar un Array con el resultado de la evaluación de expresiones Smalltalk separadas por un punto y encerradas entre llaves (`{` y `}`).

```
"Arrays en Squeak"
{1 + 1. 2 class}.
{#(1) class}.
```

Mensajes

Los mensajes representan la interacción entre los componentes de un sistema Smalltalk. Un mensaje es un pedido que un objeto le hace a otro objeto.

Un mensaje está compuesto por el **receptor** (el objeto al que se le hace el pedido), el **selector** (el nombre del mensaje) y, si corresponde, por los **argumentos**.

Desde el punto de vista sintáctico, hay 3 tipos de mensajes: **Unary**, **Binary** y **Keyword**.

Mensajes Unary

Los mensajes Unary son mensajes sin argumentos. Son los más simples y constan de:

```
receptor mensaje.
```

```
"Ejemplos de mensajes Unary"
-1 abs.
2 class.
1000 factorial.
'aeiou' size.
Date today.
Time now.
OrderedCollection new.
#símbolo class.
String category.
```

Mensajes Binary

Los mensajes Binary están compuestos de la siguiente forma:

```
receptor unOperador argumento.
```

 Programando con Smalltalk

Los operadores válidos están compuestos por 1 ó más de los siguientes caracteres:

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

```
"Ejemplos de mensajes Binary"
3 + 5.
3 > 7.
3 = 5.
2 @ 10.
'Un String ', 'concatenado a otro'.
'Alan Kay' -> Smalltalk.
```

Mensajes Keyword

Los mensajes Keyword están formados con 1 ó más palabras clave, con sus respectivos argumentos, con la forma:

```
receptor palabraClave1: argumento1.
```

O de la forma:

```
receptor palabraClave1: argumento1 palabraClave2: argumento2.
```

Y así con 3, 4 ó más palabras claves y sus argumentos.

```
"Ejemplos de mensajes Keyword"
'Un String' first: 3.
'Un String' allButFirst: 3.
'Un String' copyFrom: 2 to: 5.
5 between: 1 and: 10.
1 to: 5.
Array with: 1 with: nil with: 'string'
```

Valor de retorno

El lenguaje Smalltalk provee un mecanismo doble de comunicación. El selector y los argumentos del mensaje permiten que el emisor le envíe información al receptor. Por otro lado el receptor

devuelve información con un objeto como resultado del envío de mensajes.

Los métodos, que son la forma que tienen los objetos de responder a los mensajes, pueden especificar el valor de retorno usando el caracter ^.

```
unMétodoConValorDeRetorno1
   "Este es un ejemplo de método que responde nil como valor de retorno"

   ^ nil
```

```
unMétodoConValorDeRetorno2
   "Este es un ejemplo de método que responde al mismo receptor como valor
de retorno.

   Si el método no tiene un valor de retorno explícito, el receptor es el
resultado"
```

Cascading messages

A veces es necesario enviarle varios mensajes al mismo receptor. En el lenguaje Smalltalk hay una forma sintáctica de enviar más de un mensaje al mismo receptor.

La forma es terminar el envío del primer mensaje con el caracter punto y coma (;) y a continuación escribir el siguiente mensaje.

```
"Mensajes en cascada"
Transcript
   clear;
   show: 'algo en el Transcript'; cr;
   show: 'y algo más'; cr.
```

Precedencia

Los mensajes se evalúan de izquierda a derecha. Los mensajes Unary tienen precedencia sobre los Binary, y a su vez los Binary tienen precedencia sobre los mensajes Keyword. Siempre se pueden romper las reglas de precedencia utilizando paréntesis.

```
'string' at: -2 negated   >>>   'string' at: (-2 negated)

'string' at: 2 + -1 negated   >>>   'string' at: (2 + (-1 negated))
```

Las simples reglas de precedencia, de la sintaxis Smalltalk, tienen algunas implicaciones desconcertantes para personas acostumbradas a otros lenguajes de programación. En Smalltalk el compilador NO sabe de, por ejemplo, sumas y multiplicaciones. Eso quiere decir que el compilador no puede determinar que, cuando operamos con números, la multiplicación tiene precedencia sobre la suma.

Analicemos esta sentencia:

```
3 + 2 * 4   >>>   20
```

Según las reglas de precedencia de Smalltalk se envía primero el mensaje **+** (con el argumento 2) y al resultado (5) se le envía el mensaje * (con el argumento 4). De esa forma el resultado es 20 y no 11 como hubiese sido en otros lenguajes.

Siempre podemos utilizar paréntesis para forzar la precedencia que deseamos:

```
3 + (2 * 4)   >>>   11
```

Variables

La memoria disponible para un objeto se organiza en variables. Las variables tienen un nombre y cada una hace referencia a un único objeto en cada momento. El nombre de las variables puede ser usados en expresiones que quieran referir a ese objeto.

```
métodoDeEjemplo: argumento
    "Este método muestra el uso de diferentes tipos de variables.

    argumento: argumento al método
    variableTemporal: variable temporal al método
    variableDeInstancia: variable de instancia
    Smalltalk: variable global
    each: argumento para el bloque
    "

    | variableTemporal |
```

```
variableTemporal := Smalltalk allClasses.

variableDeInstancia := variableTemporal select:[:each |
                                  | variableTemporalAlBloque |
                                  variableTemporalAlBloque := 1.
                                  each name beginsWith: argumento
                              ].
```

El nombre de las variables está compuesto por una secuencia de letras y dígitos, empezando con una letra. Las variables temporales y de instancia comienzan con una letra minúscula, las globales comienzan con una letra mayúscula. Otra convención en lo que respecta al nombre de las variables es que si este está compuesto de varias palabras, cada una (excepto la inicial en algunos casos) debe comenzar por mayúscula.

Los nombres de clases son, también, variables globales

Cuando se crear una clase, el Smalltalk crea una variable global que referencia al objeto clase. De ahí se desprende que la convención de nombres de clase sea la misma que la convención de nombres de variables globales.

Un literal siempre se refiere a un único objeto pero una variable puede referirse a diferentes objetos en diferentes momentos.

Asignación:

El objeto referenciado por una variable cambia cuando una **asignación** es evaluada. Las asignaciones, en Smalltalk, tienen la forma:

```
variable := ExpresiónSmalltalk.
```

```
"Ejemplos de asignaciones"
x := 0.
y := 1.
punto := x @ y.
clases := Smalltalk allClasses.
```

Pseudo-variables:

Una pseudo-variable es un identificador que referencia a un objeto. La diferencia con las variables “normales” es que no se pueden asignar y siempre referencian al mismo objeto.

```
"Pseudo-variables constantes"

nil. "Referencia a un objeto usado cuando hay que representar el concepto de
'nada' o de 'vacío'. Las variables que no se asignaron nunca, referencian a nil"

true. "Referencia a un objeto que representa el verdadero lógico."

false. "Referencia a un objeto que representa el falso lógico."



"Pseudo-variables no-constantes"
self. "Referencia al receptor del mensaje."

super. "Referencia al receptor del mensaje, pero indica que no debe usarse la
clase del receptor en la búsqueda del método a evaluar. Se usa, sobre todo,
cuando se especializa un método en una subclase y se quiere invocar el método
de la superclase."

thisContext. "Referencia al objeto contexto-de-ejecución que tiene toda la
información referente a la activación del método."
```

Explicar la diferencia entre variables globales, temporales, de instancia, etc.

Bloques

En Smalltalk el comportamiento (el código) también es un objeto. Los bloques son una forma de capturar comportamiento (código) en un objeto para utilizarlo a nuestra discreción. La forma básica de un bloque es una lista de expresiones, separadas por un punto, encerradas entre corchetes (`[]`).

```
[ExpresiónSmalltalk1].
[ExpresiónSmalltalk1. ExpresiónSmalltalk2].
```

La forma de activar un bloque (de ejecutar el código que encierra) es enviándole el mensaje `#value`. La última expresión del bloque será el valor de retorno.

```
"Ejemplo con un bloque sin argumentos ni variables temporales"

| bloque |
bloque := [World flash]. "En este punto el mundo NO parpadea"
```

```
bloque value. "Ahora que evaluamos el bloque el mundo SI parpadea"
```

Una cosa muy interesante del lenguaje Smalltalk es que no existen, como parte de la sintaxis, las 'estructuras de control'. La funcionalidad que los lenguajes tradicionales se da con extensiones en la sintaxis, en Smalltalk se resuelve con mensajes a objetos.

```
"Estructuras tipo IF"
Smalltalk allClasses size > 2000
        ifTrue:[Transcript show: '¡Cuantas clases!'; cr].


Smalltalk allClasses size > 2000
        ifTrue:[Transcript show: '¡Cuantas clases!']
        ifFalse:[Transcript show: '¡No son tantas las clases!'].
Transcript cr.
```

```
"Estructuras tipo IF"
| número paridad |

número := 11.

(número \\ 2) = 0
    ifTrue:[paridad := 0]
    ifFalse:[paridad := 1].

Transcript show: 'el número ', número asString, ' tiene paridad igual a ', paridad
asString; cr.
```

```
"Estructuras tipo IF"
| número paridad |

número := 11.
paridad := (número \\ 2) = 0 ifTrue:[0] ifFalse:[1].

Transcript show: 'el número ', número asString, ' tiene paridad igual a ', paridad
asString; cr.
```

```
"Estructura tipo WHILE"
| index |
index := 1.
[index < 10]
    whileTrue:[
        Transcript show: index; cr.
```

```
    index := index + 1
].
```

```
"Estructuras tipo FOR"
1 to: 10 do:[:index | Transcript show: index; cr].
1 to: 10 by: 2 do:[:index | Transcript show: index; cr].

25 timesRepeat:[Transcript show: '.'].
Transcript cr.
```

Es muy interesante ver la implementación del los métodos `True>>ifTrue:ifFalse:` y `False>>ifTrue:ifFalse:` para entender como funcionan las 'estructuras de control' con objetos y mensajes.

A los bloque, también, se les puede activar con argumentos. La sintaxis para definir un bloque que necesita argumentos para evaluarse es la siguiente:

```
[:argumento1 | Expresiones].
[:argumento1 :argumento2 | Expresiones].
```

Para activar un bloque que requiere un argumento se usa el mensaje `#value:`, para activar uno que requiere 2 argumentos se usa el mensaje `#value:value:`, y así sucesivamente para bloques con 3 o más argumentos.

```
"El mensaje #do: requiere un bloque con un argumento"
| suma |
suma := 0.
#(2 3 5 7 11 13) do:[:primo | suma := suma + primo].
Transcript show: suma; cr.


"Lo mismo el mensaje #collect:"
| productos |
productos := #(2 3 5 7 11 13) collect:[:primo | primo * primo].
Transcript show: productos; cr.
```

```
"A las colecciones SortedCollection se les puede especificar un bloque con 2
argumentos como comparador"
| colección |
colección := SortedCollection sortBlock:[:x :y | x size < y size].
colección add: 'un string más largo'.
```

 Programando con Smalltalk

```
colección add: 'un string'.
colección add: #(#un #array #con @simbolos).
Transcript show: colección; cr.
```

Y, por último, se pueden declarar variables temporales al bloque con la siguiente sintaxis.

```
[| variableTemporal | Expresiones].
[:argumento1 | | variableTemporal | Expresiones].
```

Comentarios

Se pueden insertar comentario en cualquier parte del código. Los comentarios están encerrados entre comillas dobles. El primer comentario de un método se considera comentario del método.

Método de Ejemplo

El siguiente método de ejemplo muestra muchos de los aspectos de la sintaxis del lenguaje Smalltalk.

```
métodoDeEjemplo: argumento
   "Este es un pequeño método que muestra varias de las partes de la
   sintaxis del lenguaje Smalltalk.

   El método tiene envío de mensajes Unary, Binary y Keyword;
   declara argumentos y variables temporales; accede a una variable
   global; usa literales como array, character, symbol, string, integer
   y float; usa las pseudo-variables true, false, nil, self y super; usa
   bloques con y sin argumentos, con y sin variables temporales al
   bloque; hace una asignación; devuelve un resultado al finalizar y,
   fundamentalmente, no hace nada útil.

   Basado en el método encontrado en:
   http://wiki.cs.uiuc.edu/VisualWorks/A+small+method+that+uses+all+of+the+Smalltalk+syntax"

   | variableTemporal bloqueConVariableTemporal |

   true & false not & (nil isNil) ifFalse: [self halt].

   variableTemporal := self size + super size.

   bloqueConVariableTemporal := [
```

Programando con Smalltalk

```
                    | variableTemporalAlBloque |
                    variableTemporalAlBloque := 1.
                    variableTemporalAlBloque := variableTemporalAlBloque + 1
                  ].
bloqueConVariableTemporal value.

#($a #a 'a' 1 1.0)
  do: [:each | Transcript show: (each class name); show: ' '; cr].

^ argumento < variableTemporal
```

Herramientas

Un ambiente Smalltalk típico cuenta con muchas herramientas que asisten en la tarea de desarrollo de software. Según el estilo que escogimos para el libro, explicaremos las herramientas según lo requieran los ejemplos que desarrollamos en el capítulo "¡Manos a la Obra!"

Librería de Clases y Frameworks

La imagen de un Smalltalk cuenta con cientos, si no miles, de clases con funcionalidad aprovechable por nuestro desarrollo.

Las clases que vienen en Smalltalk nos brindan, entre otras cosas, la siguiente funcionalidad:

Números: Existen todo tipo de números. Enteros, coma flotante, fracciones, etc.

Colecciones: El framework de colecciones de Smalltalk es uno de los más antiguos y más funcionales que existen en la actualidad. La lista de colecciones incluye `Bag`, `Set`, `OrderedCollection`, `SortedCollection`, `Dictionary`, etc.

String: Soporte para cadenas de caracteres de bytes y cadenas que soportan caracteres unicode.

Boolean: Las clases `Boolean`, `True` y `False` se usan, entre otras cosas, para implementar algunas de las 'estructuras de control'.

Cronología: Clases como `Date`, `Time`, `DateAndTime`, `Month`, `Week`, `Year`,

Gráficos: Smalltalk tiene mucho que ver con el desarrollo de las interfaces de usuario gráficas. En los Smalltalk completamente auto-contenidos (como Squeak) todo lo referente al procesamiento gráfico está implementado en Smalltalk y, por ende, es inspeccionable y modificable por el usuario. Se cuentan con operaciones 2D básicas (como BitBtl) hasta soporte para gráficos 3D con OpenGL. El Squeak, a la fecha, tiene soporte para colores con el canal alfa (transparencia), anti-aliasing, renderizado de TTF (True Type Fonts), etc.

Stream: Hay veces que es necesario combinar operaciones de acceso a los elementos de una colección con operaciones de inserción de elementos. Los típicos mensajes de enumeración de las

colecciones de Smalltalk no permiten insertar elementos mientras sucede la iteración. La jerarquía de clases de Stream permite la iteración de colecciones a la vez que la inserción de elementos. La metafora de los "streams de objetos" funcionó tan bien en Smalltalk que, a partir de entonces, se usa para acceder a fuentes de datos externas en Smalltalk y en muchos lenguajes orientados a objetos.

Weak References: Se puede hacer un uso avanzado del recolector de basura utilizando referencias débiles a objetos. Las referencias débiles, al contrario de las referencia normales o fuertes, no evitan que un objeto sea reclamado por el recolector. Un objeto puede ser reclamado por el recolector cuando no tenga referencias en absoluto, o tenga sólo referencias débiles. Muy útil para implementar caches, pool de instancias, mecanismos de finalización de objetos, etc.

Multithreading: El Smalltalk soporta multithreading desde los inicios. Se cuenta con una rica variedad de clases para hacer computación concurrente de forma sencilla. Las clases `Process` (Proceso = Thread) y `Semaphore` (Semáforo) sirven de base para la programación con threads.

Excepciones: Smalltalk cuenta con un moderno esquema de excepciones. A diferencia de otros lenguajes, toda la implementación de excepciones está escrita en el mismo lenguaje. Entre otras cosas, el mecanismo de excepciones de Smalltalk permite continuar con la ejecución en el punto siguiente donde ocurrió la excepción.

Metaclases: Todo en Smalltalk es un objeto. Todos los objetos tienen una clase. Las clases, como todo, son objetos. Las clases tienen su clase, que se llama la Metaclase. Todo el mecanismo de herencia está modelado con clases y metaclases.

Seaside: El Seaside es un framework para hacer aplicaciones web basado en continuations. Un framework como Seaside simplifica muchísimo el manejo del flujo de un sitio web.

SUnit: Este es el framework 'padre' de todos los frameworks de unit-testing que existen.

Magma: Base de datos de objetos, multiusuario, que permite una completa transparencia a la hora de persistir objetos.

Máquina Virtual

La máquina virtual (virtual machine - VM) de Squeak está escrita en Squeak. El truco consistente en la utilización de un subconjunto del lenguaje Smalltalk que puede ser traducido a lenguaje C y compilado para la plataforma necesaria. Eso perminte, entre otras cosas, un nivel de portabilidad muy alto que convierte a Squeak en una de las piezas de software más portable contando, a la fecha, unas 25 plataformas diferentes. Todo el proceso de depuración de la máquina virtual puede hacerse en Squeak, esto facilita mucho la investigación con arquitecturas de VM diferentes, extensiones, etc.

La VM de Squeak, a la fecha, es de 32 bits (aunque ya exista una VM de 64 bits). Cuenta con un recolector de basura (garbage collector) generacional de muy altas prestaciones.

¡Manos a la Obra!

Vamos a introducirnos de lleno en la programación con Smalltalk. Aprender a programar con Smalltalk es mucho más que aprender una sintaxis y una librería de clases; es mucho más importante conocer como utilizar el ambiente en nuestro beneficio y entender que implica utilizar un ambiente de objetos vivos.

Smalltalk no es sólo un lenguaje de computación, Smalltalk es un ambiente donde conviven objetos y estos interactúan entre sí enviándose mensajes. Toda la computación ocurre como el resultado del envío de mensajes a los objetos. Cuando un usuario interactúa con un ambiente Smalltalk, este ambiente se ve modificado como efecto de esa interacción.

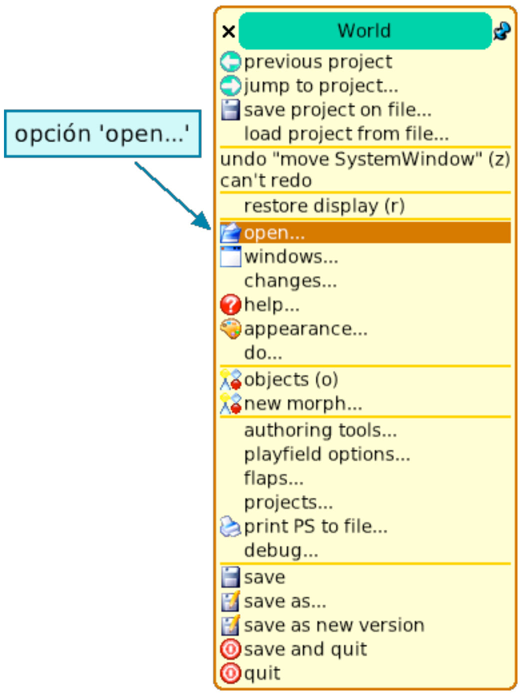
Recordemos que en Smalltalk todo es un objeto. Las clases, los métodos, etc (obviamente) también son objetos y para impactar sobre ellos debemos enviarles mensajes. Esto implica que la tarea de programación no es diferente a cualquier otra cosas que se haga en un ambiente Smalltalk: Objetos que reciben mensajes y que reaccionan como efecto de ese envío de mensajes.

En Smalltalk no existe diferencia entre “desarrollo” y “ejecución”, sino que la programación se hace modificando objetos mientras estos están funcionando. Para ilustrar mejor esta idea vamos a desarrollar, paso a paso, un ejemplo sencillo. El ejemplo, también, nos servirá para tomar un primer contacto con las principales herramientas de desarrollo con Smalltalk.

Modificando Objetos Vivos

Vamos a crear una clase de nombre `Cliente` que sería la representación en nuestro ambiente de los clientes reales de un supuesto sistema para facturación. Para eso nos valdremos de una las herramientas que Smalltalk nos brinda para interactuar con el ambiente: El Browser de Clases.

Para hacernos del Browser de Clases activamos el menú del Mundo haciendo clic sobre el Mundo de Squeak y seleccionamos la opción '`open...`' para obtener el submenú:



El Mundo

El objeto gráfico que contiene a todos los demás objetos gráficos se llama Mundo.

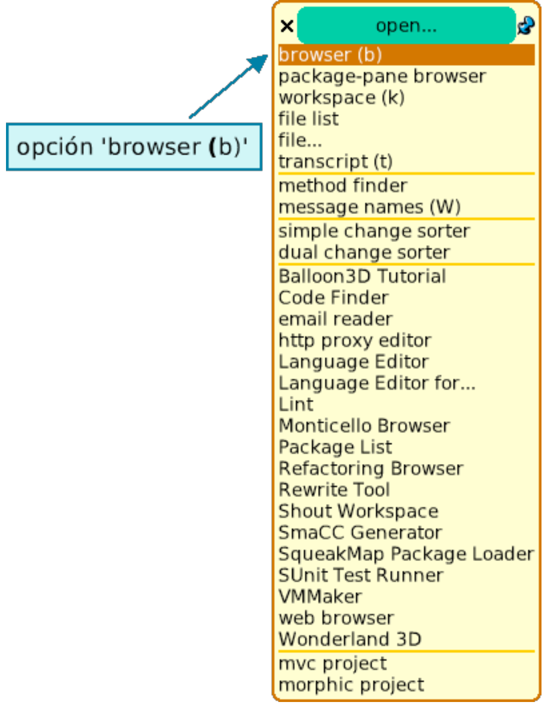
El Mundo de Squeak es similar al escritorio de los sistemas operativos actuales, pero respetando la regla básica de Smalltalk: Todo es un Objeto.

El Mundo es un objeto y disponemos de una variable global para acceder a el llamada `World`. Podemos evaluar algunas de las siguientes sentencias para ver como reacciona el mundo:

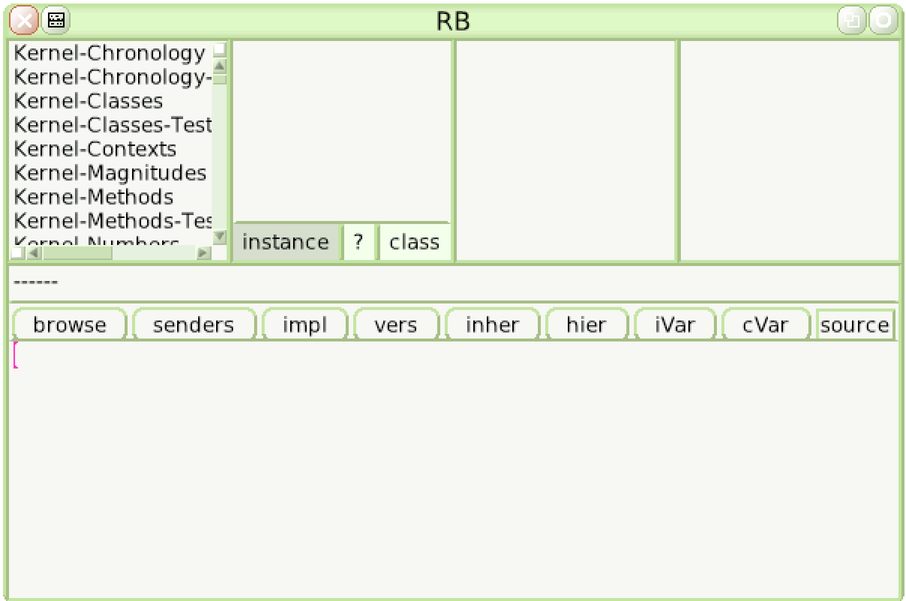
```
"Cambiar el color al Mundo"
World color: Color lightYellow muchLighter.
World color: Color white.

"El Mundo se oscurece por un instante"
```

```
World flash.

"Juguemos con el borde del Mundo"
World borderWidth: 4.
World borderColor: Color red.
```

Luego seleccionamos la opción '`browser (b)`':



Y obtenemos el Browser de Clases:



Browser de Clases

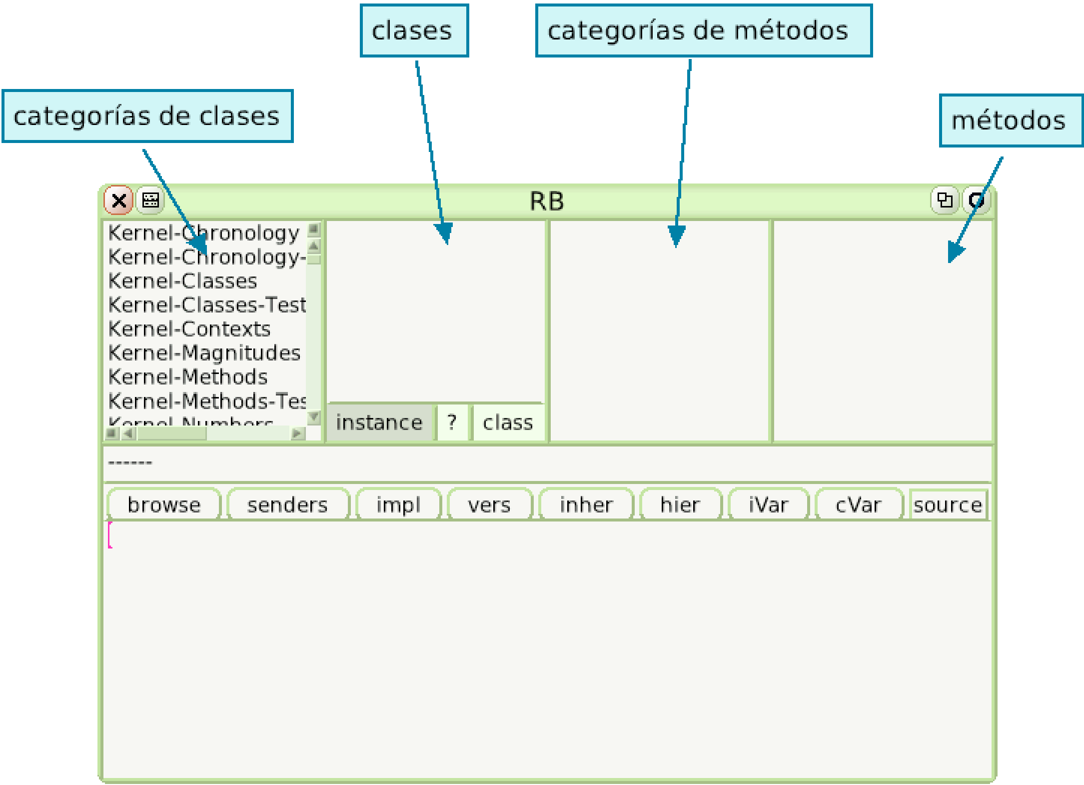
Esta herramienta nos permite ver y modificar todas las clases que tenemos en nuestro ambiente. Podemos también crear o eliminar categorías de clases, clases, categorías de métodos y métodos.

Como Squeak está escrito en si mismo, esta herramienta nos permite ver absolutamente todo el funcionamiento de Squeak. Desde el Compilador hasta las Ventanas, desde los Números Enteros hasta los tipos Booleanos, todo visible y todo modificable.

Tipos de Browser de Clases

Existen diferentes tipos de Browser de Clases, sin embargo prácticamente todos comparten la funcionalidad básica. En este libro usaremos el Refactoring Browser que, además de brindarnos las características del browser básico, nos brinda opciones para la refactorización del código.

El browser está compuesto principalmente por 4 paneles superiores y un panel inferior. En los paneles superiores encontraremos - de izquierda a derecha - las Categorías de Clases, las Clases, las Categorías de Métodos y los Métodos.

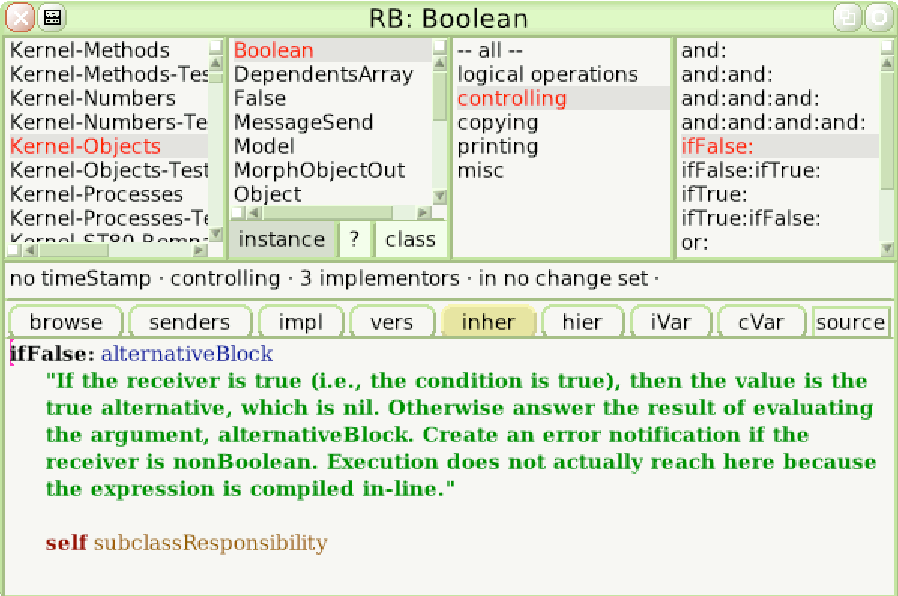


Categorías de Clase y Métodos

Tanto las categorías de clases como las categorías de métodos no tienen una semántica para el Smalltalk en si mismo, sino que sirven para documentar el diseño agrupando clases y métodos por funcionalidad apuntando a mejorar el entendimiento del sistema por las personas.

En Smalltalk se lee mucho más código del que se escribe, así que todo el entorno promueve la escritura de código limpio y documentado y todo el tiempo que usemos en escribir código limpio es recompensado.

Si seleccionamos una de las opciones del panel, en los paneles subsiguientes veremos la información correspondiente a la selección. Por ejemplo, si seleccionamos en el primer panel la categoría de clases `'Kernel-Objects'`, veremos en el segundo panel las clases dentro de esa categoría (como `Boolean`, `DependentsArray`, `False`, `MessageSend`, etc). De la misma forma, si ahora seleccionamos la clase `Boolean` en el segundo panel, veremos en el tercer panel las categorías de métodos de esa clase; y si seleccionamos una categoría de métodos como `'controlling'` veremos en el último panel los métodos. Por último, si seleccionamos uno de los métodos, veremos en el panel inferior el código de dicho método.



Ahora crearemos una categoría de clases para poner nuestra clase `Cliente` dentro. Para lograr eso pedimos el menú contextual, haciendo clic con el botón azul del ratón, del panel de categorías de clases y seleccionamos la opción '`add item...`'.

Squeak y el Ratón

Squeak es multiplataforma y a la fecha funciona en más de 20 plataformas diferentes. Linux, varios sabores de Unix, Windows, Apple, PDA como iPaq, y un largo etc. Tantas plataformas nos exponen a algunos problema, uno de ellos es como utilizar los diferentes tipos de ratones que las distintas plataformas poseen.

Por motivos históricos que se remontan al ratón que tenía la computadora Xerox Alto, los 3 botones del ratón se los nombra con colores: **Rojo**, **Amarillo** y **Azul**.

Botón Rojo:

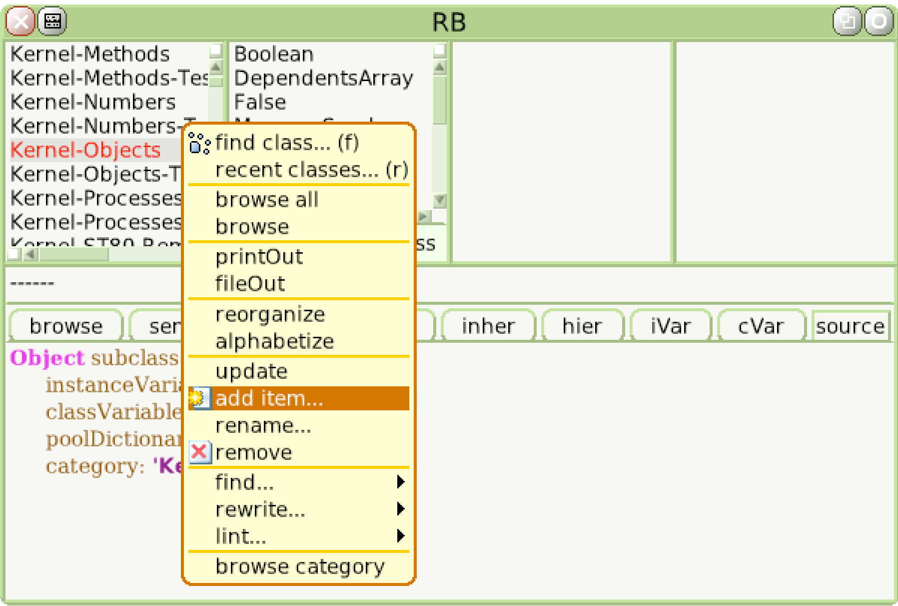
El Botón Rojo es el que se usa para seleccionar. En prácticamente todas las plataformas el Botón Rojo se mapea al botón principal de la plataforma (normalmente el botón izquierdo del ratón, o el 'tap' del lápiz).

Botón Amarillo:

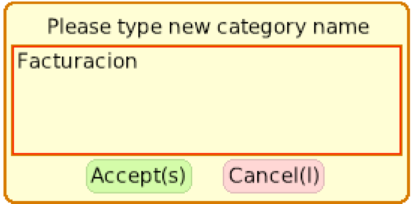
El Botón Amarillo se utiliza para pedir el Halo de los objetos gráficos. En Linux se utiliza el botón derecho, en Windows el botón del medio o ALT-clic y en Apple Opt-clic.

Botón Azul:

El Botón Azul es el que se usa para obtener el menú contextual. En Linux (y todos los sabores de Unix) obtenemos el menú contextual haciendo clic con el botón del medio del ratón, en Windows se obtiene el menú contextual con un clic del botón derecho y en Apple se obtiene el menú contextual con Cmd-clic.

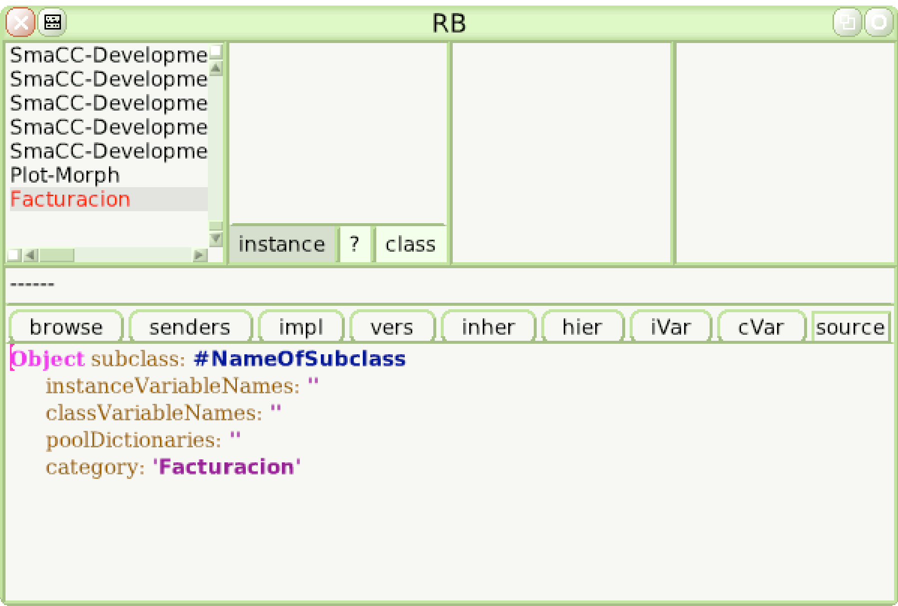


y luego ingresamos el nombre de la nueva categoría, en nuestro caso tecleamos `'Facturacion'`



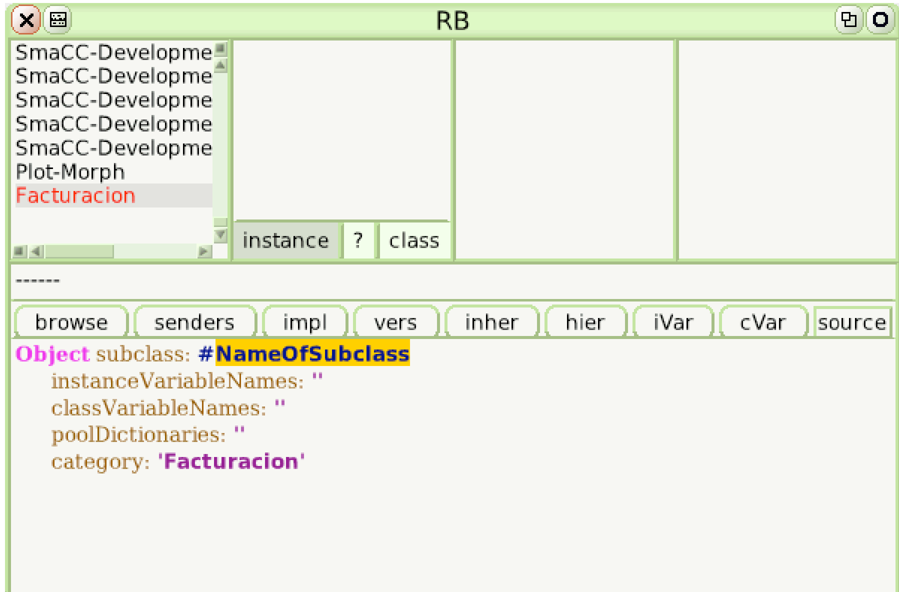
Presionamos `Intro` o hacemos clic en el botón `Accept(s)`

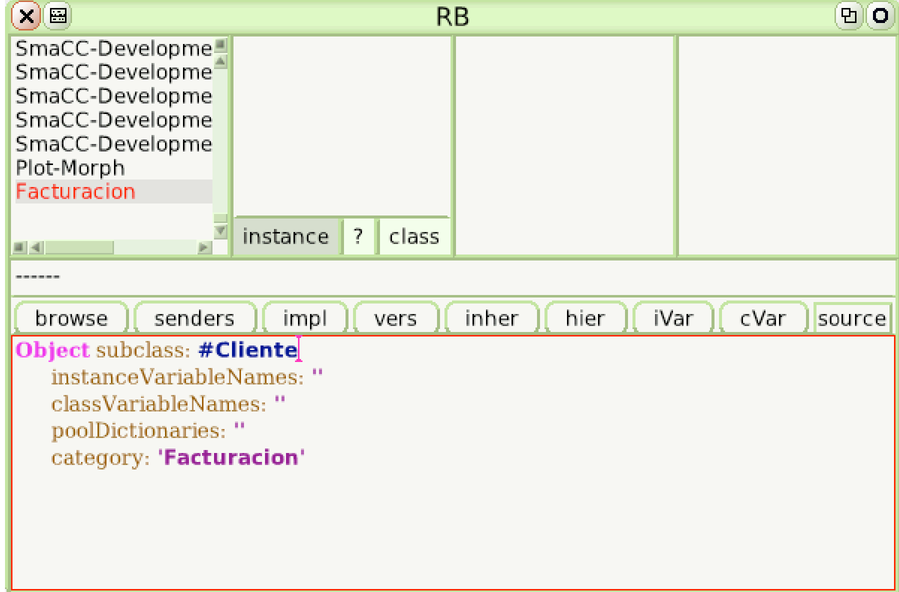
> **Foco de teclado**
>
> En Squeak el foco de teclado lo tiene el control que estén siendo apuntados por el puntero del ratón.
>
> Antes de teclear algo hay que asegurarse que estemos apuntando al control con el ratón.



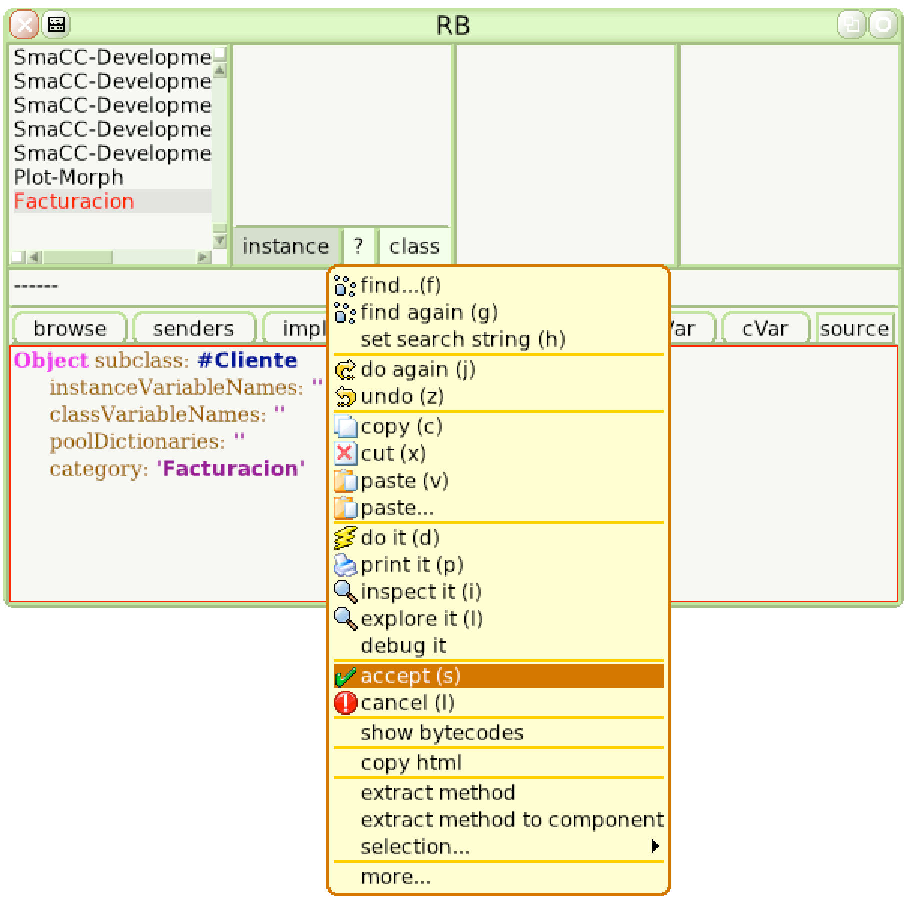
Programar con Smalltalk es enviar mensajes a objetos que viven en un ambiente. Programar, con Smalltalk, no es más que modificar un ambiente de objetos y adaptarlo a nuestra necesidad.

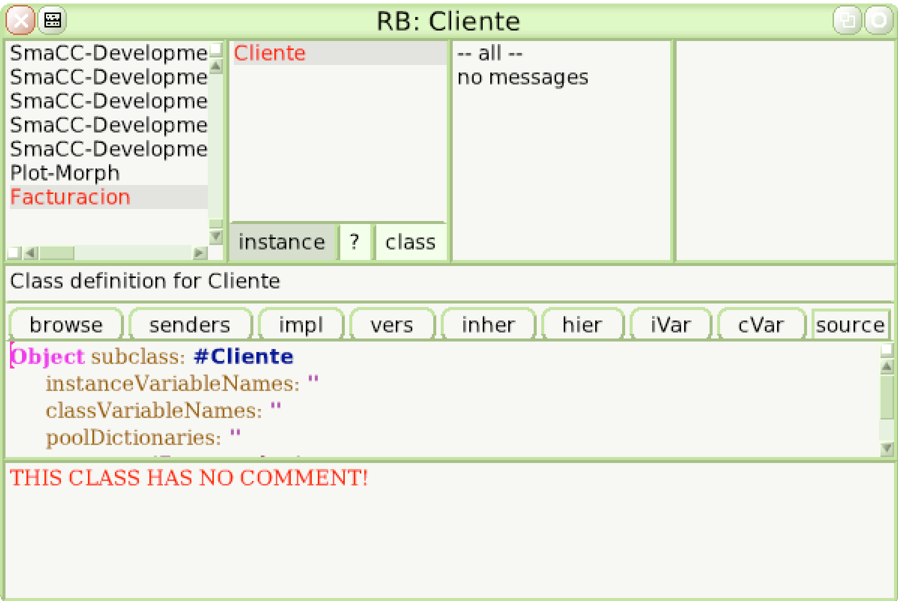
Al crear una nueva categoría, esta se selecciona automáticamente y vemos en el panel inferior una plantilla (un template) para crear una clase nueva. Para crear una nueva clase tenemos que enviarle un mensaje a la superclase, para eso nos podemos valer de la plantilla que el Browser de Clases nos ofrece. Simplemente reemplazamos donde dice `NameOfSubclass` por el nombre de la clase que queremos crear.





Ahora debemos aceptar nuestro cambio usando la opción `'accept (s)'` el menú contextual del panel inferior.





Ahora vamos a crear una instancia de nuestra recién creada clase `Cliente`. Para evaluar algo de código, podemos utilizar otra herramienta que un ambiente Smalltalk nos brinda: El Workspace.

Para abrir un Workspace usaremos nuevamente el Menú del Mundo, y el submenú `open...` y dentro la opción '`workspace (w)`' o la opción '`Shout Workspace`'.

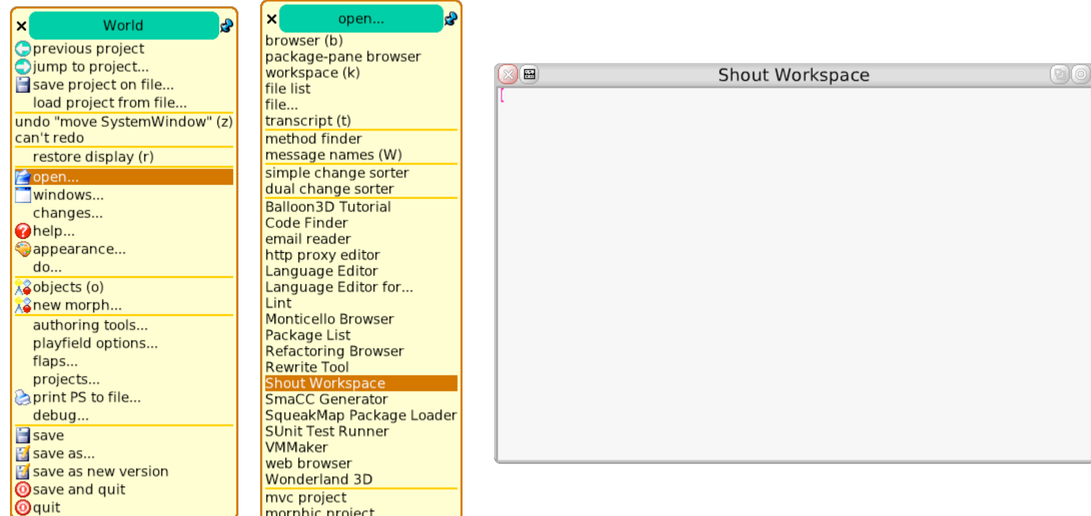
Workspace

El Workspace (espacio de trabajo) es una ventana que nos permite ordenar el código que vamos evaluando de forma interactiva en nuestro ambiente.

El Workspace es una herramienta conveniente, pero es importante resaltar que en Smalltalk se puede evaluar código en cualquier panel de texto y no sólo en los Workspaces.

Tipos de Workspace

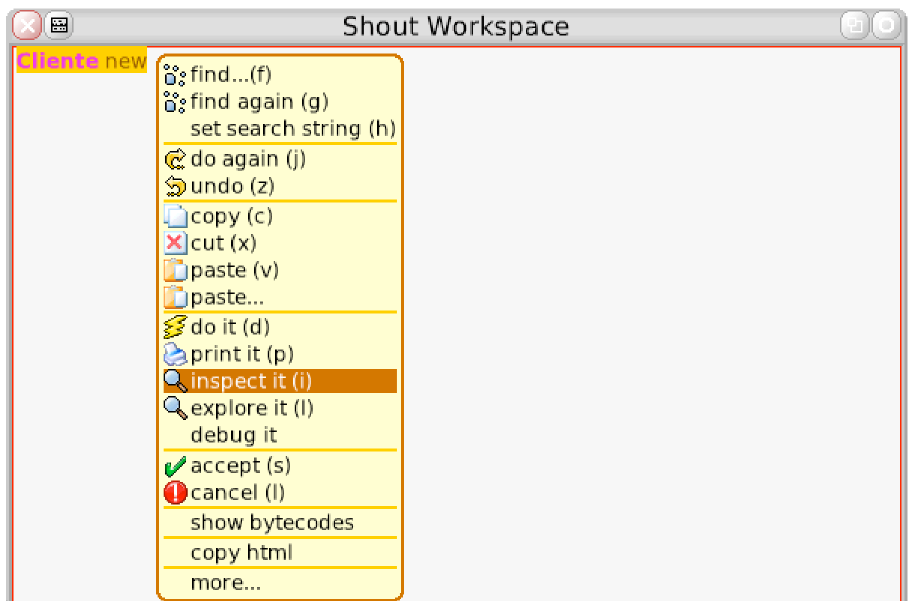
Un 'Shout Workspace' es un workspace que hace syntax-highlight conforme lo utilizamos, para este libro usaremos esta herramienta ya que es un poco más vistosa que el Workspace tradicional. De todas formas, desde el punto de vista de funcionalidad no existe ninguna diferencia entre ambas alternativas.



Ahora tipeamos, en el recién abierto Workspace, lo siguiente:

```
Cliente new
```

Lo seleccionamos con el ratón, activamos el menú contextual e invocamos la opción '`inspect it (i)`'.



Evaluando Código

Al evaluar código Smalltalk de forma interactiva, tenemos 4 opciones sobre como tratar la

respuesta.

do it: Evaluar el código e ignorar la respuesta.

print it: Evaluar el código e imprimir, por pantalla, la representación como `String` de la respuesta.

inspect it: Evaluar el código e inspeccionar el resultado. Esta es la opción utilizada en el ejemplo.

explore it: Evaluar el código y explorar el resultado. El `Explorer` es otro formato de `Inspector`.

Sentencias de Ejemplo para evaluar, imprimir, inspeccionar o explorar

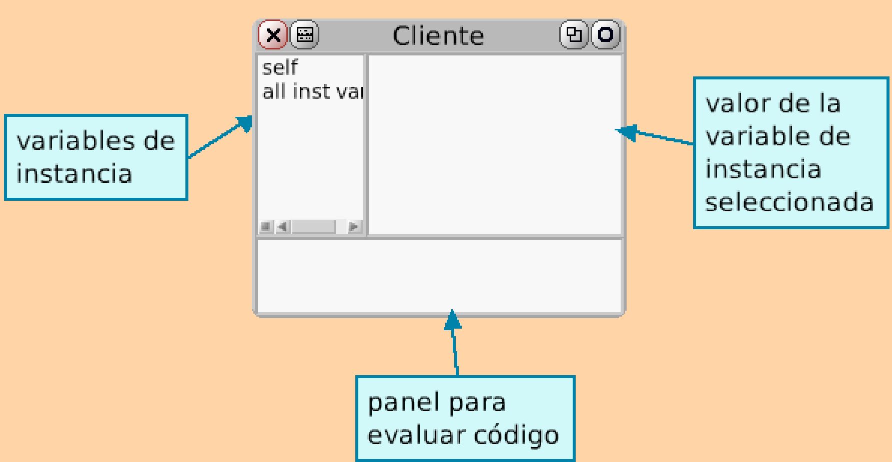
```
"Algunas cositas con String"
'un string'.
'un string' , ' concatenado a otro'.
'un string de cierto tamaño' size.
'*string*' match: 'un string'.

"Algunos números"
34.
1 / 3.
(1 / 3) asFloat.
3.141592653589793.
1000 factorial.

"Algo del Mundo"
World extent.
World color.

"Expresiones Booleanas"
1 > 3.
true.
false.

"Otras Expresiones"
nil.
Date today.
Time now.
Time millisecondsToRun: [1000 factorial].

"Algunas cositas con colecciones"
Smalltalk allClasses.
Smalltalk allClasses collect:[:each | each name].
Smalltalk classNames.
Smalltalk classNames size.
Smalltalk classNames select:[:each | '*String*' match: each].
```

Esta acción nos brindará otra de las herramientas de Smalltalk: El Inspector.

Inspector

El Inspector es una herramienta que nos permite ver como está compuesto un objeto y, también, nos permite enviarle mensajes.

Podemos ver que variables de instancia tiene y que valores tienen estas variables de instancia. En Smalltalk absolutamente todos los objetos pueden ser inspeccionados.



En el panel de las variables de instancia hay 2 variables 'especiales': `self` y `all inst var`. Al seleccionar `self` se ve en el panel de la derecha la representación como String del objeto inspeccionado. Si se selecciona `all inst var` se ven todas las variables de instancia como si fuesen una sola.

En el panel de código se puede introducir cualquier sentencia Smalltalk, la única salvedad es que la variable especial `self` apunta al objeto inspeccionado.

Ahora vamos a modificar un poco la clase `Cliente`, pero lo haremos sin cerrar el Inspector a la instancia. Es decir: vamos a modificar la estructura del objeto mientras este esté vivo. Volvemos al Browser y nos aseguramos que estén seleccionadas la categoría de clase '`Facturacion`' y la clase `Cliente`. Entre las comillas simples que están detrás de la palabra `instanceVariableNames:` tipeamos lo siguiente:

browse | senders | impl | vers | inher | hier | iVar | cVar | source

```
Object subclass: #Cliente
    instanceVariableNames: 'nombre'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Facturacion'
```

Aceptamos los cambios como lo hicimos anteriormente (menú contextual, opción '`accept (s)`').

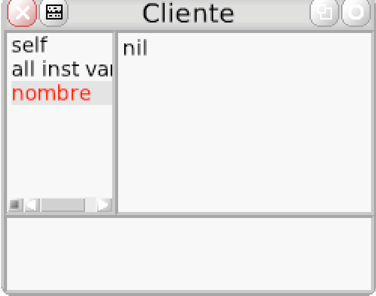
Hot-Keys

Algunas de las opciones de los Menús que fuimos utilizando durante el ejemplo tienen una letra entre paréntesis. Esa letra es el hot-key y podemos invocar esas opciones presionando ALT y la letra.

Ejemplo:

`browser (b)`	»	`ALT-b`
`workspace (k)`	»	`ALT-k`
`accept (s)`	»	ALT-s
`inspect it (i)`	»	ALT-i
`explore it (I)`	»	ALT-MAYÚSCULA-i

Si todo salió bien, ya deberíamos ver como el Inspector a la instancia de Cliente que dejamos abierto ahora nos muestra una nueva variable de instancia.

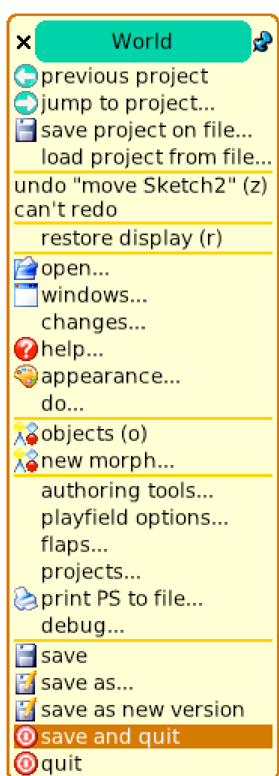


Ahora vamos a ponerle un nombre a nuestro cliente. Para eso seleccionamos la variable de instancia en el panel de la izquierda y luego ingresamos en el panel de la derecha lo siguiente:

```
'pedro'
```

Aceptamos los cambios (menú contextual y opción 'accept (s)' o ALT-s).

Para terminar este ejemplo, saldremos de nuestro ambiente grabando todos los cambios. Abrimos el Menú del Mundo y escogemos la opción 'save and quit'.



Cuando entremos nuevamente a nuestro Smalltalk, tendremos todo el ambiente en el mismo estado que lo dejamos.

El ejemplo que acabamos de hacer deja en evidencia una diferencia fundamental del Smalltalk con los lenguajes de programación "tradicionales" donde un objeto nunca sobrevive a un cambio del

programa. En Smalltalk es posible, y es habitual, modificar un sistema mientras esta funcionando.

Este ejemplo, además, nos mostró algunas de las herramientas principales de Smalltalk (Browser, Workspace e Inspector) y vimos un poco de como se usan.

Ya estamos preparados para pasar a un ejemplo más complejo.

Parser de XML basado en una Pila

Para procesar archivos XML, cosa que haremos en el próximo ejemplo, usaremos una herramienta que simplifica mucho la escritura de parsers de XML. El parser que utilizaremos funciona de la siguiente manera: Cada vez que el parser encuentra un nuevo tag de XML, le envía un mensaje al objeto que está más arriba en su pila; y el resultado del envío del mensaje se apila. Cuando ese mismo tag termina, el objeto se desapila de la pila. El primer objeto apilado es uno puesto a mano y es quien procesará los tags de más alto nivel. El mensaje que se le envía al objeto que está arriba de la pila tiene como argumento un diccionario que contiene los atributos del tag.

Veamos un ejemplo lo suficientemente simple como para analizarlo por completo. Imaginemos un archivo XML como el siguiente:

```
<agenda>
   <persona apellido="Gates" nombre="Bob">
      <direccion ciudad="Los Angeles" numero="1239" provincia="CA" calle="Pine Rd."/>
   </persona>
   <persona apellido="Smith" nombre="Joe">
      <informacion-contacto>
         <email direccion="joes@iro.aibiem.com"/>
         <telefono numero="888-7657765"/>
      </informacion-contacto>
      <direccion ciudad="New York" numero="12789" provincia="NY" calle="W. 15th Ave."/>
   </persona>
</agenda>
```

Este archivo contiene los datos de una agenda; la agenda tiene dentro personas y las personas tienen direcciones y informaciones de contacto; a su vez las informaciones de contacto tienen teléfonos y direcciones de email.

Instruimos al parser para que envíe los siguientes mensajes según el tag que se procese:

Tag de XML	Mensaje a Enviar
<agenda>	#crearAgenda:
<persona>	#crearPersona:
<direccion>	#crearDireccion:
<informacion-contacto>	#crearInformacionContacto:
<email>	#crearEMail:
<telefono>	#crearTelefono:

Pondremos en la pila, antes de comenzar el parseo, un objeto `Agendas`. Ese objeto es capaz de crear y almacenar agendas, es decir: es capaz de responder al mensaje `#crearAgenda:`

Antes de comenzar, la pila del parser contiene lo siguiente:

una Agendas

Smalltalk con Estilo

En Smalltalk existe un idiom que consiste en nombrar a los objetos con 'a' o 'an' (uno o una en Inglés) y el nombre de la clase.

De esa forma un instancia de la clase `String` se la llama aString o 'a String', una instancia de `Date` se llama aDate o 'a Date', una instancia de `Integer` es anInteger o 'an Integer', etc.

El lugar ideal para leer todas las convenciones e idioms que se usan en Smalltalk es el libro “Smalltalk with Style” (ver bibliografía).

```
Agendas>>crearAgenda: aDictionary
    "Crea una nueva agenda y la guarda en el receptor"
    | agenda |
    agenda := Agenda new.
    self addAgenda: agenda.
    ^ agenda.
```

NombreDeClase>>nombreDeMétodo

Cuando es necesario volcar el código fuente de un método en papel, se utiliza la convención de escribirlo con el nombre de la clase, seguido de >> y, a continuación, el nombre del método.

Esto es una convención para el papel y no es parte de la sintaxis Smalltalk.

Como el resultado de la evaluación del mensaje `#crearAgenda:` es una `Agenda`, esta se apila y es ahora la responsable de contestar al mensaje `#crearPersona:`

La pila, en este punto, contiene lo siguiente:

una Agenda
una Agendas

```
Agenda>>crearPersona: aDictionary
   "Crea una nueva persona y la guarda en el receptor"
   | persona |
   persona := Persona
                     apellido: (aDictionary at: 'apellido')
                     nombre: (aDictionary at: 'nombre').
   self addPersona: persona.
   ^ persona.
```

Este método de la clase `Agenda` es muy similar al anterior método `#crearAgenda:` excepto porque este método considera los atributos presentes en el tag `<persona>`, que vienen en el argumento aDictionary, que para crear una `Persona` con nombre y apellido.

Ahora se apiló, en la pila, una `Persona`, así que ahora este objeto es el responsable de contestar al mensaje `#crearDireccion:`

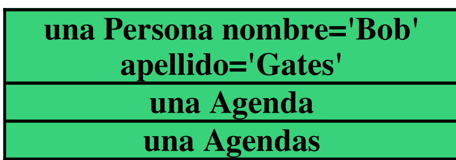
una Persona nombre='Bob' apellido='Gates'
una Agenda
una Agendas

```
Persona>>crearDireccion: aDictionary
   "Crea una nueva direccion y la guarda en el receptor"
   | direccion |
   direccion := Direccion
                       calle: (aDictionary at: 'calle')
                       numero: (aDictionary at: 'numero')
                       ciudad: (aDictionary at: 'ciudad')
                       provincia: (aDictionary at: 'provincia').
   self addDireccion: direccion.
   ^ direccion.
```

La Direccion devuelta por el método anterior se apila, dejando la pila de la siguiente forma:

una Direccion calle='Pine Rd.' numero='1239' ciudad='Los Angeles' provincia='CA'
una Persona nombre='Bob' apellido='Gates'
una Agenda
una Agendas

Ahora el parser se enfrenta, por primera vez en este ejemplo, al hecho que un tag se cierra. En este punto el tag `<direccion>` se cierra, entonces el parser desapila la dirección.



Ahora se cierra el tag `<persona>`, así que se desapila la persona.

una Agenda
una Agendas

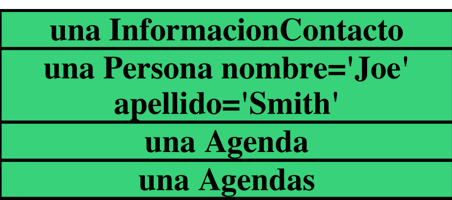
Ahora se envía otro mensaje `#crearPersona:` a la Agenda que está en la pila, creando una nueva Persona y apilándola.

una Persona nombre='Joe' apellido='Smith'
una Agenda
una Agendas

Ahora se envía un mensaje `#crearInformacionContacto:` a la persona como reacción al tag `<informacion-contacto>`.

```
Persona>>crearInformacionContacto: aDictionary
   "Crea una nueva informacion de contacto y la guarda en el receptor"
   ^ informacionContacto := InformacionContacto new.
```

La pila queda ahora de la siguiente forma:



A continuación, el objeto que está arriba de la pila (una InformacionContacto) recibe el mensaje `#crearEMail:`

```
InformacionContacto>>crearEMail: aDictionary
   "Crear un nuevo email y lo guarda en el receptor"
   | eMail |
   eMail := EMail direccion: (aDictionary at: 'direccion').
   self addEMail: eMail.
   ^ eMail.
```

El resultado del método anterior (un EMail) se apila, pero inmediatamente después se desapila ya que el tag `<email>` se cierra y nuevamente queda arriba de la pila la Información de Contacto.

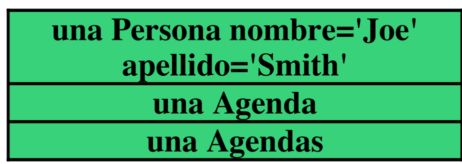
El tag `<telefono>` se procesa de forma similar al tag `<email>`

```
InformacionContacto>>crearTelefono: aDictionary
   "Crear un nuevo teléfono y lo guarda en el receptor"
   | telefono |
   telefono := Telefono numero: (aDictionary at: 'numero').
   self addTelefono: telefono.
   ^ telefono.
```

A continuación se cierra el tag `<informacion-contacto>` y se desapila la Información de Contacto.

Ahora se procesa el tag `<direccion>` de igual forma que se procesó para la persona anterior, apilando el resultado y desapilándolo en cuanto se cierra el tag.

El estado de la pila, en este punto, es el siguiente:



Ya estamos terminando de procesar el XML, sólo resta procesar el cierre del tag `<persona>` y el cierre del tag `<agenda>`. Es decir que se desapila la persona y luego la agenda, quedando la pila en el mismo estado que estaba antes de comenzar el parseo.

una Agendas

Habiendo quedado explicado como funciona el parser de XML basado en una pila, podemos comenzar con el importador de Wikipedia.

Importador de Wikipedia

Ahora nos ocuparemos de un ejemplo un poco más complejo y, a su vez, más completo. Haremos un importador de los datos de la Wikipedia.

Wikipedia

Wikipedia es un proyecto para escribir comunitariamente enciclopedias libres en todos los idiomas. Fue fundada por Jimmy Wales y Larry Sanger basándose en el concepto wiki que permite crear colectivamente documentos web, sin que la revisión del contenido sea necesaria antes de su aceptación para ser publicado en la red. La versión en inglés comenzó el 15 de enero de 2001. Tres años y medio después, en septiembre de 2004, unos 10.000 editores activos trabajaban en 1.000.000 de artículos en más de 50 idiomas.

Para más información: http://es.wikipedia.org/wiki/Wikipedia

Archivos de la Wikipedia:

- http://download.wikimedia.org/ y
- http://meta.wikimedia.org/wiki/Importing_a_Wikipedia_database_dump_into_MediaWiki

El proyecto Wikipedia brinda toda la información de sus sitios disponibles para bajar desde Internet. Se pueden bajar todos los artículos de las diferentes enciclopedias de diferentes idiomas en un archivo XML.

Vamos a tratar de llevar a cabo el ejemplo programando de la forma más incremental posible, programando todo lo que podamos en el Depurador.

Veamos un XML de ejemplo sacado de la Wikipedia en Latín, pero simplificado y acortado para poder analizarlo en toda su extensión:

```
<mediawiki>
  <siteinfo>
    <sitename>Vicipaedia</sitename>
    <base>http://la.wikipedia.org/wiki/Pagina_prima</base>
    <generator>MediaWiki 1.6devel</generator>
    <case>first-letter</case>
    <namespaces>
      <namespace key="-1">Specialis</namespace>
      <namespace key="0" />
      <namespace key="1">Disputatio</namespace>
```

```
    </namespaces>
  </siteinfo>
  <page>
    <title>Astronomia</title>
    <id>1</id>
    <revision>
      <id>46556</id>
      <timestamp>2005-10-31T22:28:43Z</timestamp>
      <contributor>
        <ip>80.136.214.100</ip>
      </contributor>
      <text>'''Astronomia'''</text>
    </revision>
  </page>
  <page>
    <title>Auxilium</title>
    <id>2</id>
    <revision>
      <id>44264</id>
      <timestamp>2005-10-11T16:43:35Z</timestamp>
      <contributor>
        <ip>217.72.33.184</ip>
      </contributor>
      <comment>better latin from italy [[:it:Utente:OrbiliusMagister]]</comment>
      <text>''Hic addendae sunt notiones ad usum lectoris</text>
    </revision>
  </page>
  <page>
    <title>Usor:Archibald Fitzchesterfield</title>
    <id>3</id>
    <revision>
      <id>25286</id>
      <timestamp>2002-11-26T18:33:53Z</timestamp>
      <contributor>
        <ip>host155-164.pool21345.interbusiness.it</ip>
      </contributor>
      <comment>*</comment>
      <text>Studiosus sum in Universitate Torontoniense</text>
    </revision>
  </page>
</mediawiki>
```

Vemos que un `<mediawiki>` tiene dentro `<siteinfo>` y `<page>`, que los `<page>` tienen dentro `<revision>`, y que las `<revision>` tienen el texto de las páginas.

También vemos que algunos tags se usan sólo para meter datos en forma de String (`<sitename>`, `<base>`, `<generator>`, `<case>`, `<namespace>`, `<title>`, `<id>`, `<timestamp>`, `<ip>`, `<text>`, `<comment>`, etc) para estos casos el parser nos ofrece otra forma de procesar el tag: definimos un text-tag y el objeto de arriba de la pila recibirá un mensaje con un String como argumento conteniendo el texto encerrado por el tag, y se apilará el mismo objeto.

Con este ejemplo pretendo mostrar las facilidades para la programación incremental que nos brinda un ambiente como Smalltalk. Pos esa misma razón creo que ya hemos analizado demasiado el XML antes de comenzar. ¡Empecemos ya mismo!

Lo primero que tenemos que hacer es instanciar el parser y ponerle la mínima información como para empezar a programar, creamos una clase `Mediawiki` y dentro pondremos unos métodos de clase para instanciar Mediawikis desde archivos XML.

Métodos de Clase vs. Métodos de Instancia

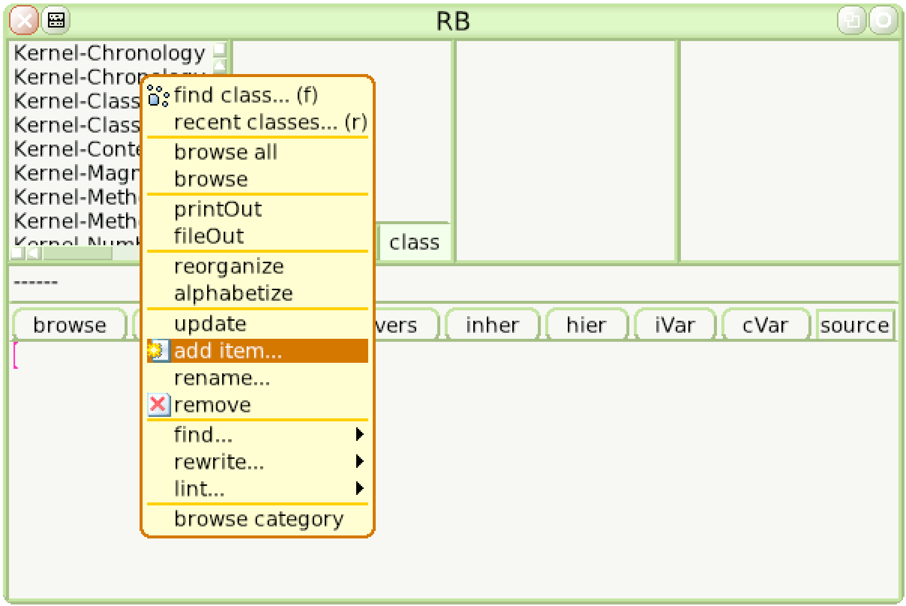
Las clases en Smalltalk son, obviamente, también objetos. La clase modela las instancias, y la clase de la clase (la metaclase) modela la clase.

Las clases reciben mensajes (como cualquier objeto) y los métodos de clase son la forma de responder a esos mensajes.

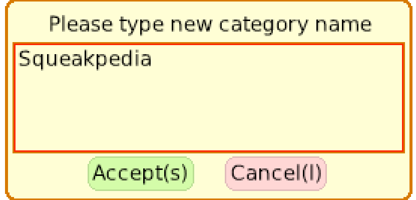
Por ejemplo: En Smalltalk los constructores no tienen una semántica especial (como si la tienen en Java, C++, etc). Son simplemente métodos de clase que tienen la responsabilidad de crear instancias.

También es frecuente encontrarse, en la clase, con métodos que operen sobre todas las instancias de la clase. Ejemplos: `#allInstances`, `#allSubInstances`, `#inspectAllInstances`, etc.

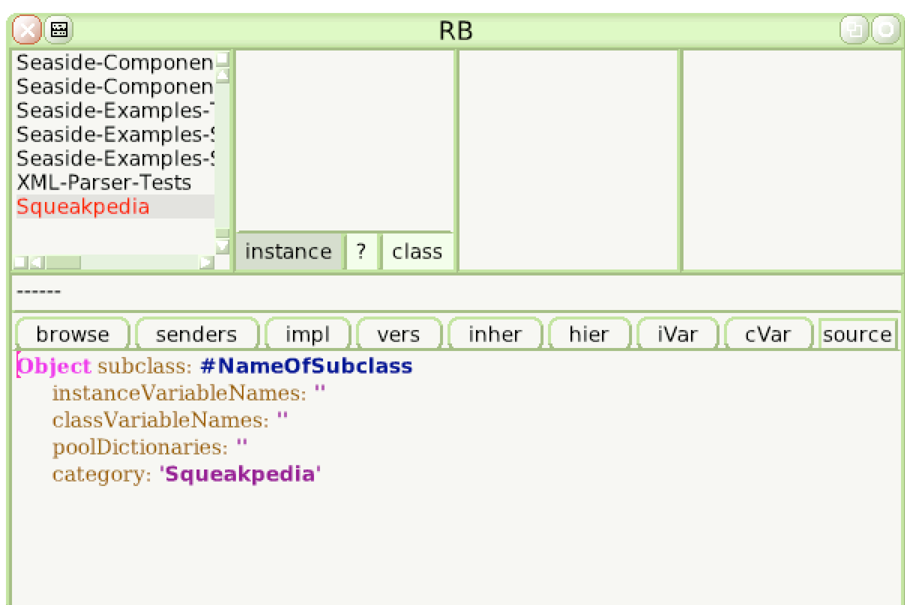
Para eso nos hacemos de un Browser de Clases (Menú del Mundo >> `'open...'` >> `'browser (b)'` o ALT-b). Ahora creamos una categoría de clases para poner nuestra clases dentro. Para lograr eso pedimos el menú contextual (haciendo clic con el botón azul del ratón) del panel de categorías de clases y seleccionamos la opción `'add item...'`.



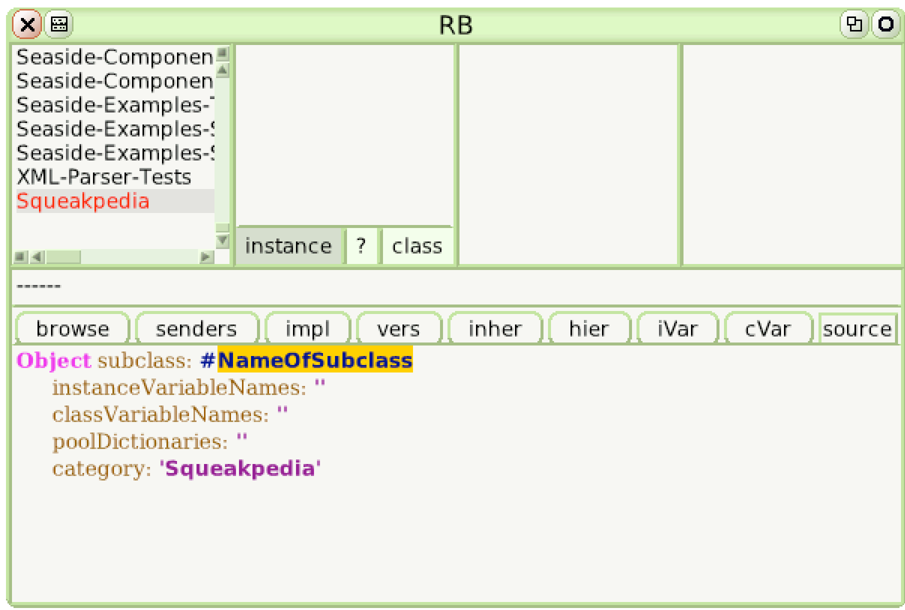
Tecleamos el nombre de nuestra Categoría de Clases



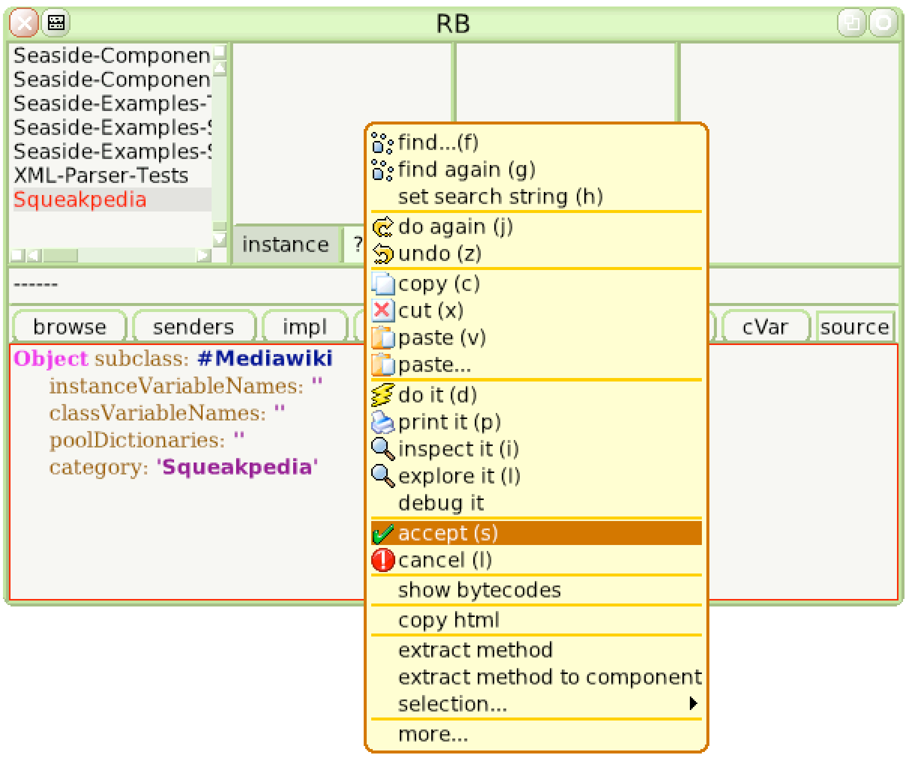
y aceptamos

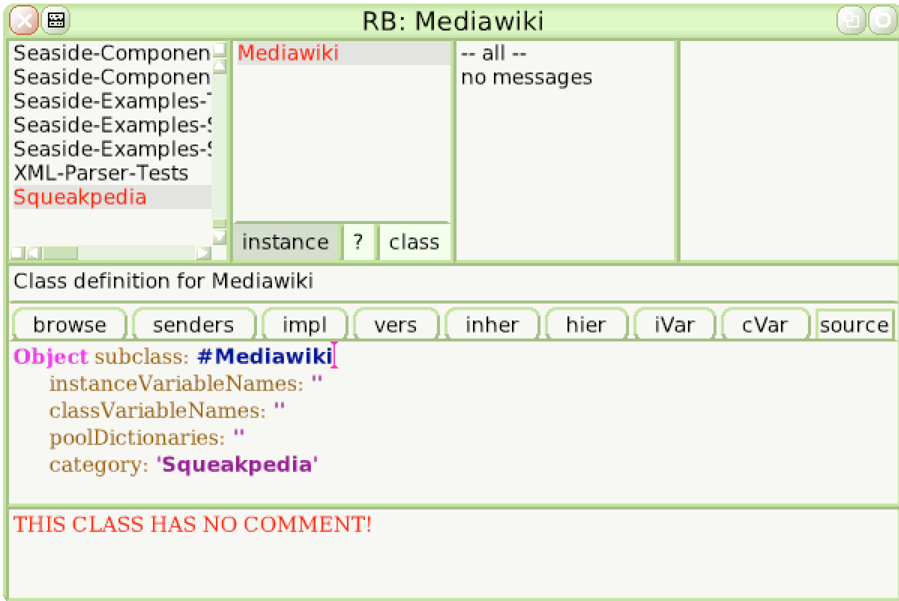


Ahora estamos listos para crear la primer clase de nuestro ejemplo. Para eso reemplazamos donde dice `NameOfSubclass` por el nombre de la clase que queremos crear, en este caso Mediawiki.



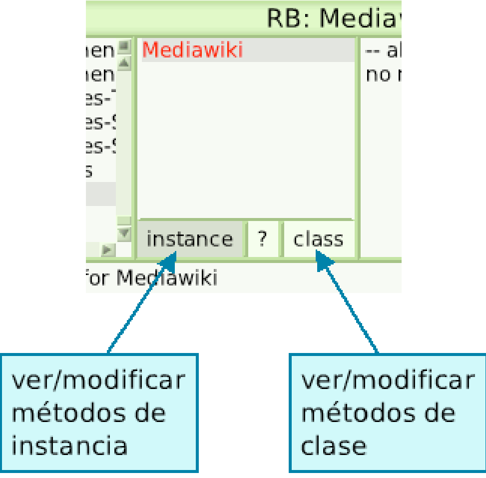
y aceptamos:



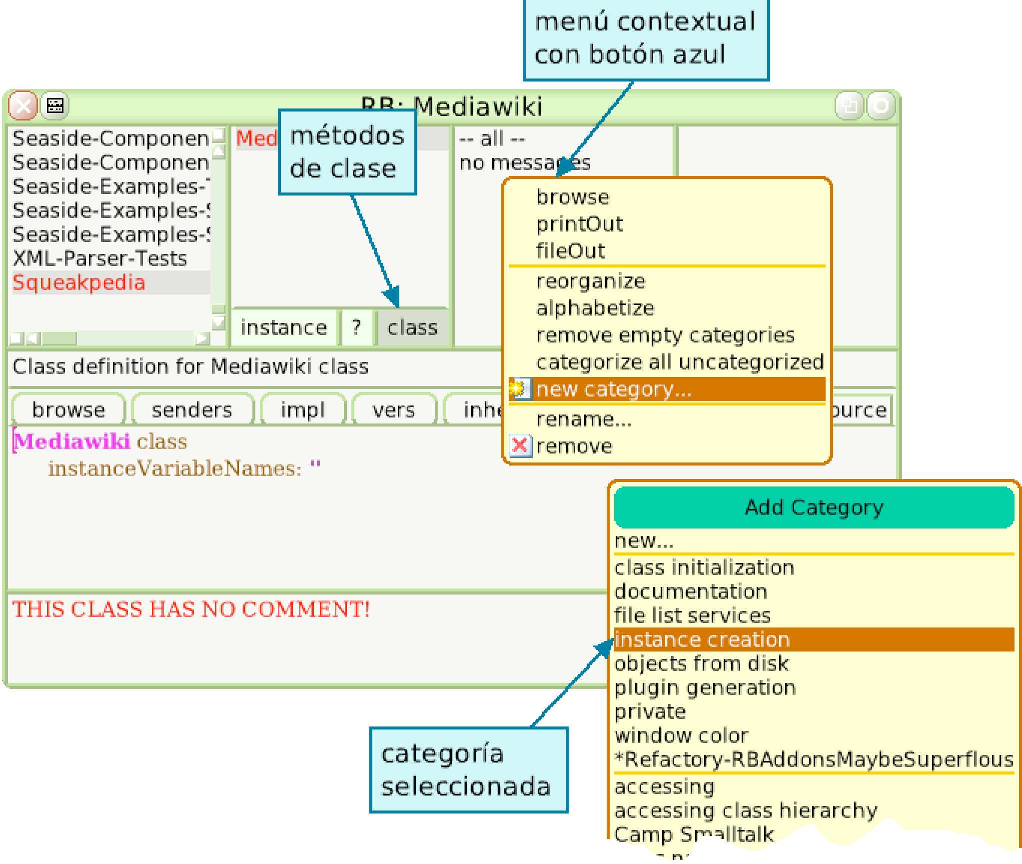


Ahora que ya tenemos nuestra clase creada vamos a crear unos métodos de clase, a modo de constructores, para instanciar Mediawikis desde archivos XML.

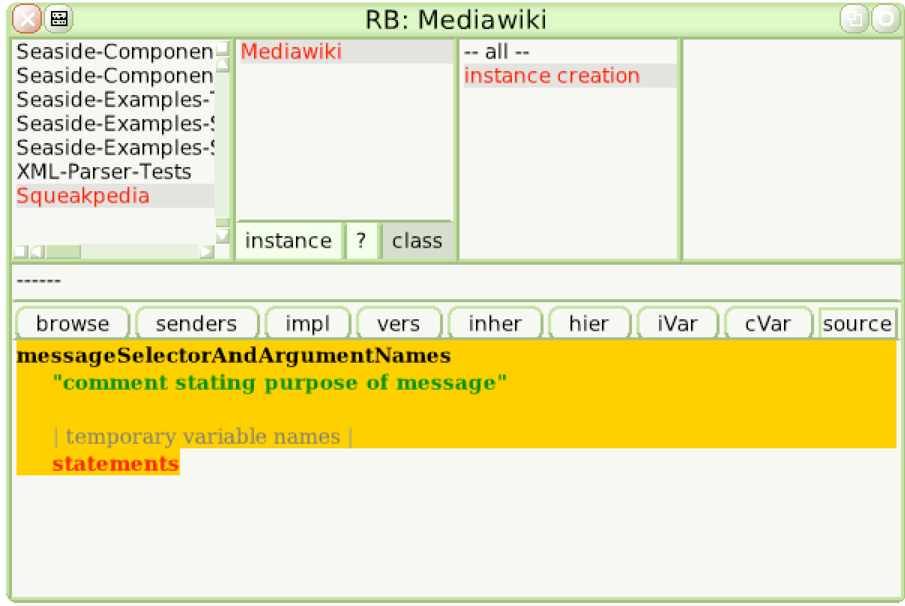
Para crear métodos de clase en lugar de métodos de instancia, debemos seleccionar el botón `class` que está debajo del panel de clases. Cuando lo hagamos, los otros 2 paneles mostrarán categorías de métodos y métodos pero de clase en lugar de instancia. Si queremos volver a ver o modificar categorías de métodos y métodos de instancia presionamos el botón `instance`.

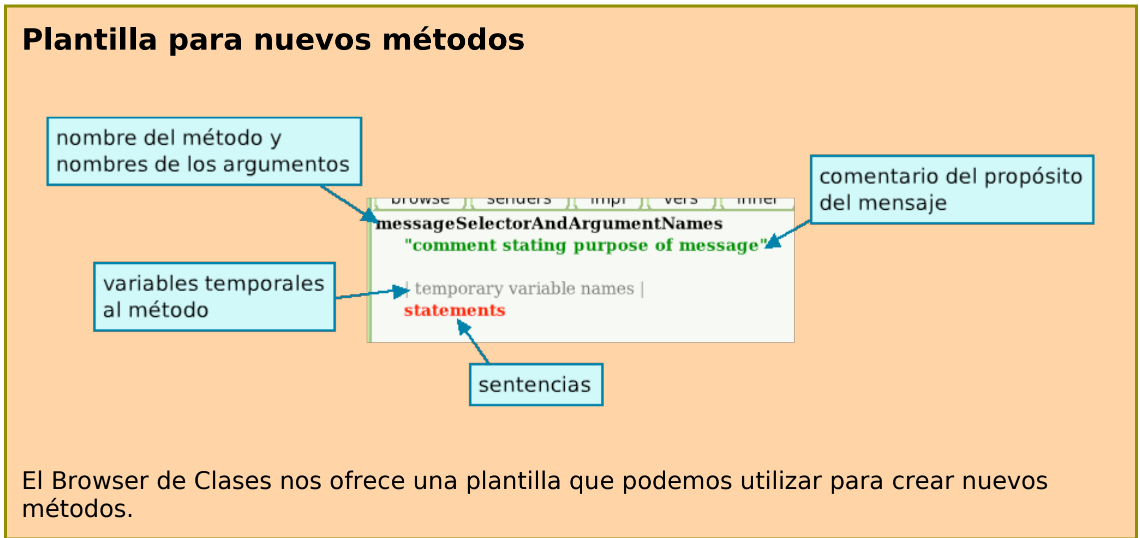


Seleccionamos el botón `class`, y creamos una categoría de métodos (de clase en este caso) llamada `'instance creation'` usando la opción `'new category...'` del menú contextual del panel de categorías de métodos y luego eligiendo la categoría desde las opciones ofrecidas.



Si todo salió bien, la recién creada categoría de métodos (de clase) queda seleccionada y el browser nos ofrece una plantilla en el panel inferior para crear un método dentro de esa categoría:

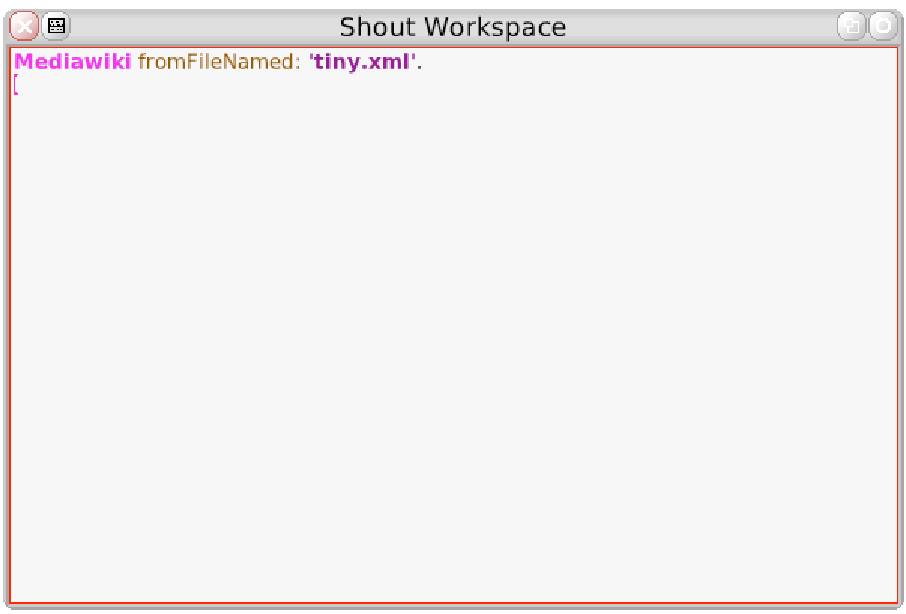




Ahora ingresamos lo siguiente en el panel inferior, y aceptamos:

```
fromFileNamed: aString
	"Crea una nueva instance del receptor desde el archivo llamado aString"

	| archivo instancia parser |

	"abre el archivo"
	archivo := FileStream readOnlyFileNamed: aString.

	"crea la instancia del receptor"
	instancia := self new.

	"instancia el parser"
	parser := XMLStackParser on: archivo.

	"la instancia recién creada es el primer objeto de la pila"
	parser push: instancia.

	"instruimos al parser sobre como procesar los tags"
	parser onTag: 'mediawiki' send: #onMediawiki:.

	"le pedimos al parser que haga su trabajo"
	parser parseDocument.

	"cerramos el archivo"
	archivo close.

	"devolvemos la instancia al remitente del mensaje"
	^ instancia
```

Es hora de ejecutar, por primera vez, nuestro importador de Wikipedia. Abrimos un Workspace (Menú del Mundo >> `'open...'` >> `'Shout Workspace'`) y tecleamos lo siguiente:



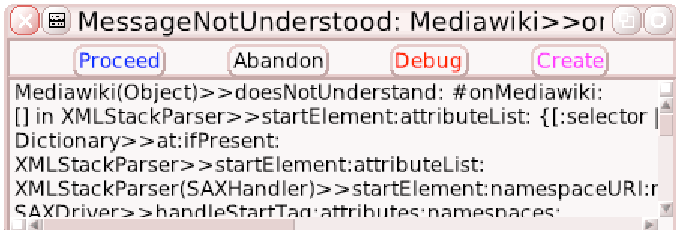
Archivos de ejemplo

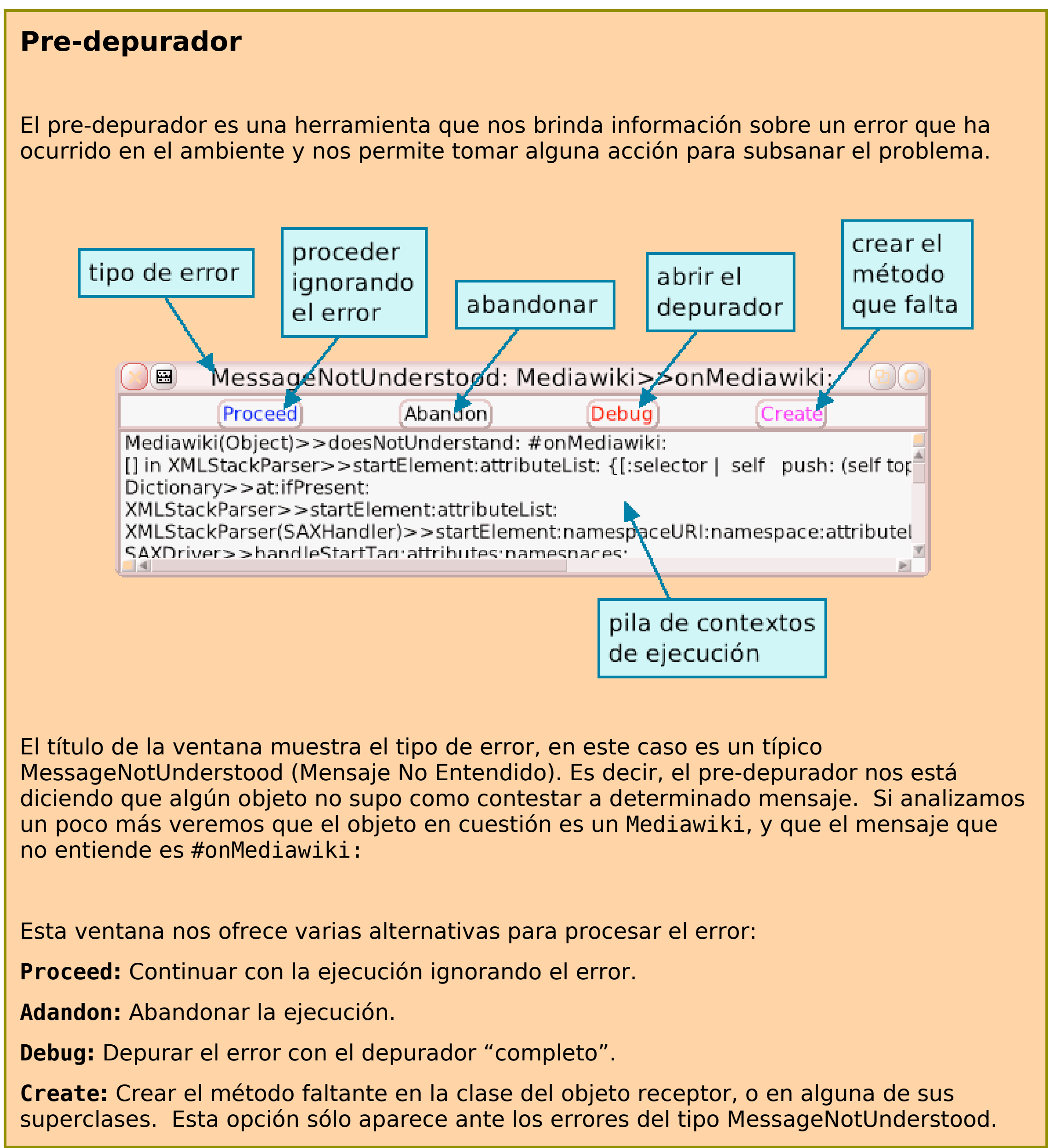
El archivo tiny.xml y otros archivos usados en este libro se pueden descargar desde el sitio web del libro en http://smalltalk.consultar.com

El archivo tiny.xml contiene exactamente el texto expuesto en la página 22.

Seleccionamos la sentencia recién tecleada y la evaluamos con `'do it (d)'` (o ALT-d).

¡Voila! ¡Aparece por primera vez un error! Bienvenido al maravilloso y dinámico mundo de Smalltalk.





Pre-depurador

El pre-depurador es una herramienta que nos brinda información sobre un error que ha ocurrido en el ambiente y nos permite tomar alguna acción para subsanar el problema.

El título de la ventana muestra el tipo de error, en este caso es un típico MessageNotUnderstood (Mensaje No Entendido). Es decir, el pre-depurador nos está diciendo que algún objeto no supo como contestar a determinado mensaje. Si analizamos un poco más veremos que el objeto en cuestión es un `Mediawiki`, y que el mensaje que no entiende es `#onMediawiki:`

Esta ventana nos ofrece varias alternativas para procesar el error:

Proceed: Continuar con la ejecución ignorando el error.

Adandon: Abandonar la ejecución.

Debug: Depurar el error con el depurador "completo".

Create: Crear el método faltante en la clase del objeto receptor, o en alguna de sus superclases. Esta opción sólo aparece ante los errores del tipo MessageNotUnderstood.

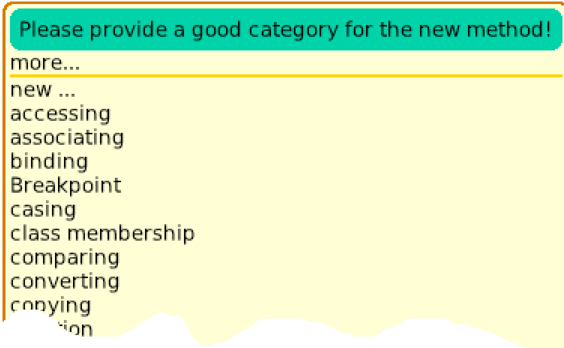
Los errores, de cualquier tipo, siempre son oportunidades de reflexión sobre nuestro sistema y nos permiten re-validar nuestras concepciones sobre el dominio que estamos modelando.

Analizando la información que nos da el pre-depurador, es fácil darse cuenta que lo que ocurre es que le dijimos al parser que enviara el mensaje `#onMediawiki:` cuando encuentre un tag `<mediawiki>`, el parser se lo envía al objeto que está arriba en la pila, es decir que se lo envía a nuestro objeto Mediawiki.

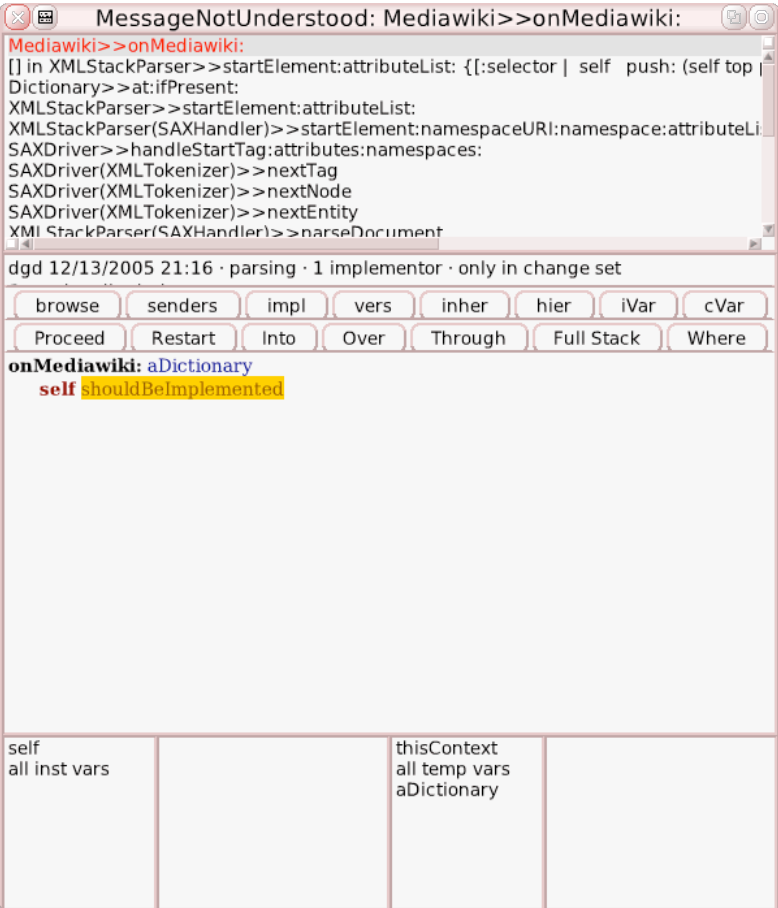
Nos aprovechamos de una de las posibles acciones que nos ofrece el pre-depurador, la opción de crear el método. Para eso presionamos el botón `Create`.

En este momento el pre-depurador nos ofrece la posibilidad de crear el método en la clase del objeto receptor, o en cualquiera de sus superclases. En este caso elegimos la clase `Mediawiki`.

A continuación el entorno nos pregunta en cual categoría de métodos queremos crear nuestro método.



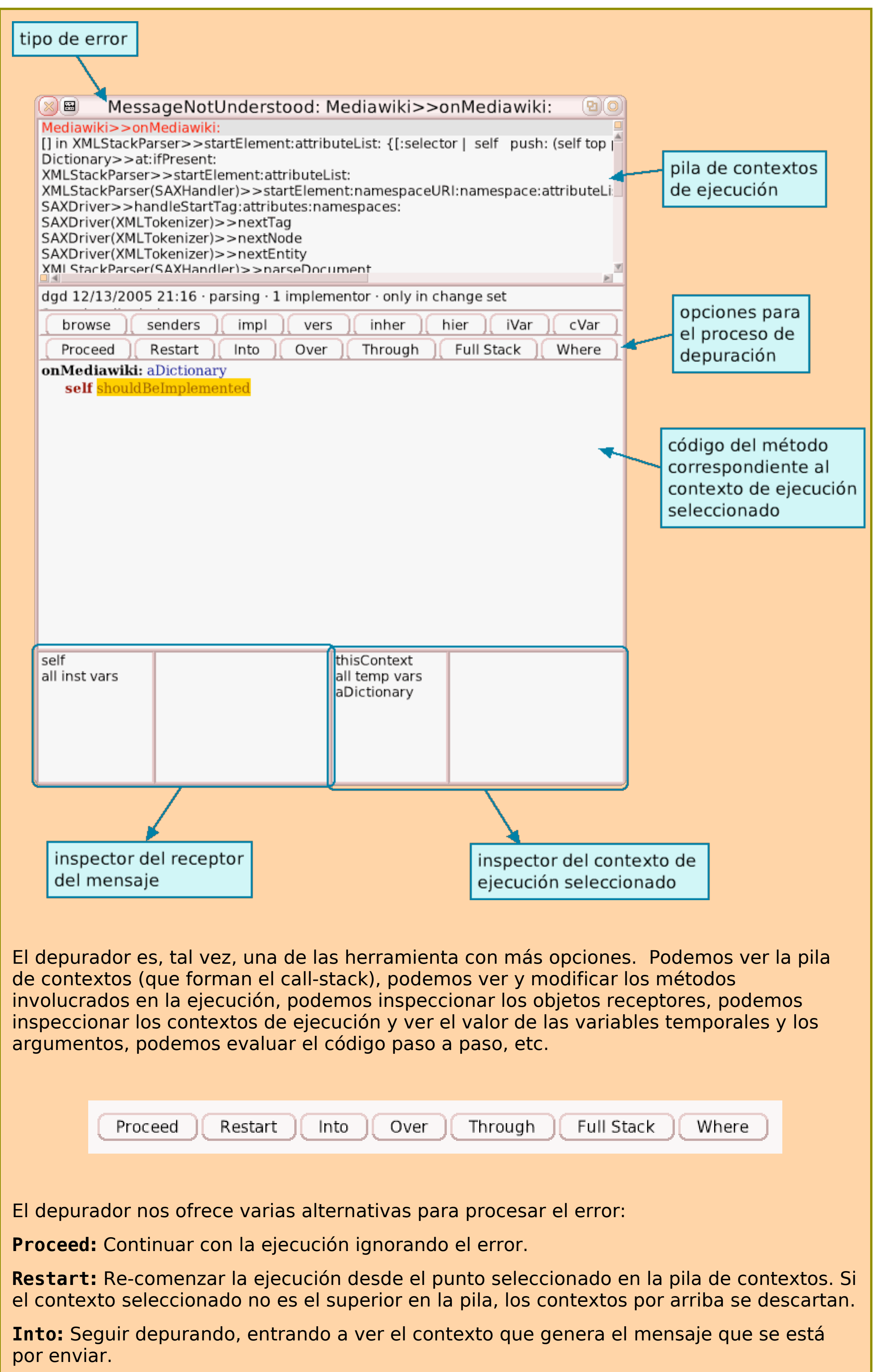
Ninguna de las categorías ofrecidas nos sirve, así que seleccionamos la opción `'new...'` y ingresamos `'parsing'`.



Les presento al depurador, una herramienta donde pasarán gran parte del tiempo que utilicen programando con Smalltalk.

Depurador

El depurador es una herramienta que permite examinar como ocurre la ejecución de código Smalltalk. También permite modificar los métodos on-the-fly y seguir la ejecución desde el punto anterior al método modificado.



El depurador es, tal vez, una de las herramienta con más opciones. Podemos ver la pila de contextos (que forman el call-stack), podemos ver y modificar los métodos involucrados en la ejecución, podemos inspeccionar los objetos receptores, podemos inspeccionar los contextos de ejecución y ver el valor de las variables temporales y los argumentos, podemos evaluar el código paso a paso, etc.

El depurador nos ofrece varias alternativas para procesar el error:

`Proceed:` Continuar con la ejecución ignorando el error.

`Restart:` Re-comenzar la ejecución desde el punto seleccionado en la pila de contextos. Si el contexto seleccionado no es el superior en la pila, los contextos por arriba se descartan.

`Into:` Seguir depurando, entrando a ver el contexto que genera el mensaje que se está por enviar.

Over: Seguir depurando, pero se ignora los detalles del envío del mensaje que se está por enviar.

Through: Seguir la ejecución y parar en la primera sentencia dentro del próximo bloque.

Full Stack: Ver la pila de contextos completa y no la versión reducida que se muestra por defecto.

Where: Volver a resaltar el mensaje que se está enviando.

Depurador 100% en Smalltalk

El Depurador de Smalltalk está escrito (como casi todo el Smalltalk) en Smalltalk. Todo es un objeto (¿Cuantas veces lo repetimos ya?) y las clases, los métodos, y los contextos de ejecución también son también objetos, así que es posible escribir un depurador 100% en Smalltalk.

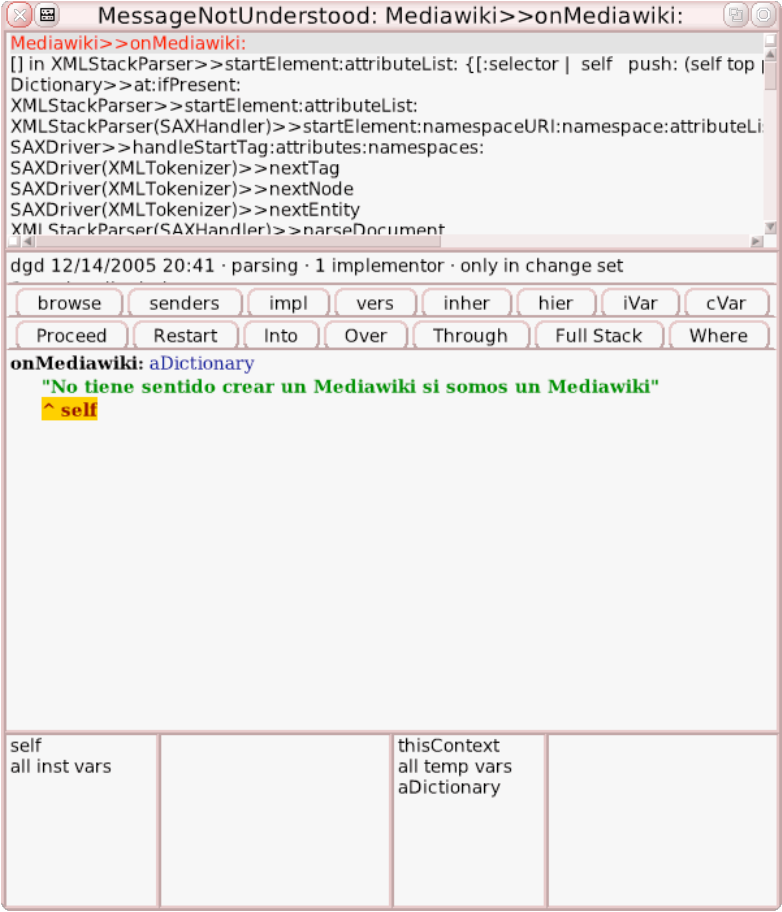
Como los contextos de ejecución son objetos como cualquier otro, de la misma forma que el depurador es un objeto como cualquier otro, podemos hacer con ellos lo mismo que con cualquier otro objeto. Por ejemplo podemos grabar la imagen con un depurador abierto y, cuando volvamos al ambiente, podemos seguir depurando el código exactamente en el punto donde lo dejamos aunque pasen meses desde que grabamos la imagen y la volvemos a abrir. También se pueden modificar para agregarle funcionalidad, etc.

El depurador se abre en ese punto, porque la implementación por defecto que se creó del método es una versión con un error que indica que tenemos que completarlo.

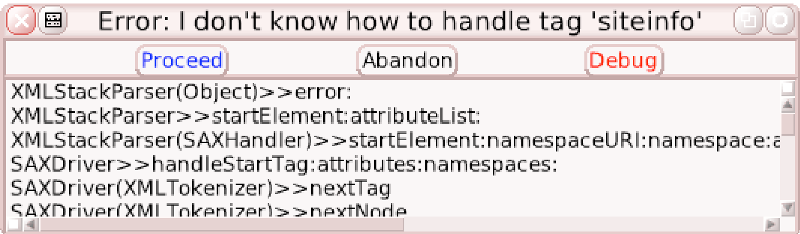
La implementación que haremos de este método es con “truco”. Un objeto de clase `Mediawiki` recibió el mensaje `#onMediawiki:` que, en teoría, debería crear un Mediawiki. Como el tag `<mediawiki>` está sólo una vez en el archivo (y por ende recibiremos sólo un mensaje `#onMediawiki:`), simplemente ignoramos el tag y devolvemos el mismo objeto para apilar.

```
onMediawiki: aDictionary
   "No tiene sentido crear un Mediawiki si somos un Mediawiki"
   ^ self
```

Tecleamos ese método en el panel que muestra el código del método del depurador y aceptamos los cambios (menú contextual y opción `'accept (s)'` o ALT-s). El depurador, ahora, luce así:



Presionamos el botón `Proceed` y seguimos con el desarrollo.



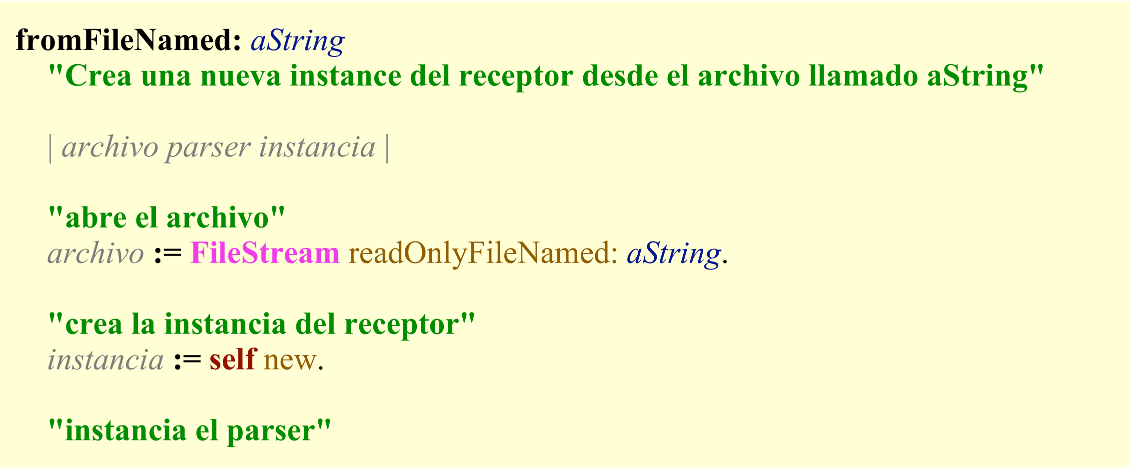
Ahora el parser nos dice que no sabe que hacer con el tag `<siteinfo>`. Eso ocurre porque no le dijimos, al parser, que hacer con los tags que no sean `<mediawiki>`.

Pedimos el depurador presionando el botón `Debug` y seleccionamos, en la pila de contextos, el que corresponde con nuestro método `Mediawiki class>>fromFileNamed:` (que es el método donde instruimos al parser como procesar cada tag).

En la parte donde instruimos al parser sobre como procesar los tags, agregamos lo siguiente:

```
parser onTag: 'siteinfo' send: #onSiteinfo:.
```

Para ganar un poco de tiempo, podemos ingresar las instrucciones para varios tags quedando el método completo así:

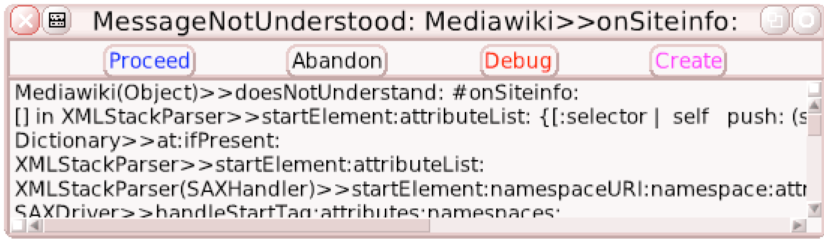


```
fromFileNamed: aString
   "Crea una nueva instance del receptor desde el archivo llamado aString"

   | archivo parser instancia |

   "abre el archivo"
   archivo := FileStream readOnlyFileNamed: aString.

   "crea la instancia del receptor"
   instancia := self new.

   "instancia el parser"
```

```
parser := XMLStackParser on: archivo.

"la instancia recién creada es el primer objeto de la pila"
parser push: instancia.

"instruimos al parser sobre como procesar los tags"
parser onTag: 'mediawiki' send: #onMediawiki:.
parser onTag: 'siteinfo' send: #onSiteinfo:.
parser onTag: 'namespaces' send: #onNamespaces:.
parser onTag: 'namespace' send: #onNamespace:.
parser onTag: 'page' send: #onPage:.
parser onTag: 'revision' send: #onRevision:.
parser onTag: 'contributor' send: #onContributor:.

"le pedimos al parser que haga su trabajo"
parser parseDocument.

"cerramos el archivo"
archivo close.

"devolvemos la instancia al remitente del mensaje"
^ instancia
```

Aceptamos los cambios, y el depurador ahora queda así:

Es interesante ver que pasó con la pila de contextos cuando aceptamos los cambios: El depurador se deshizo de los contextos que estaban por arriba del método modificado, así que estamos listos para presionar el botón `Proceed` re-comenzando desde el método cambiado.



Ahora el ambiente nos dice que nuestro Mediawiki no sabe como contestar el mensaje `#onSiteinfo:`, así que creamos el método siguiendo el mismo procedimiento que utilizamos para crear el método `#onMediawiki:`. Presionamos el botón `Create`, seleccionamos la clase `Mediawiki`, seleccionamos la categoría de métodos `'parsing'`, y obtenemos el depurador de esta forma:

Tecleamos lo siguiente:

```
onSiteinfo: aDictionary
   "Crea un SiteInfo y lo guarda en el receptor"
   siteInfo := SiteInfo new.
   ^ siteInfo
```

Antes de aceptar el syntax-highlight, con el color rojo, nos indica que no sabe que es `siteInfo` ni tampoco sabe que es `SiteInfo`. La primera será una variable de instancia, y la segunda será una clase.

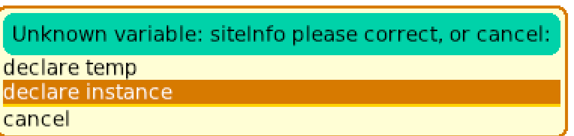
Convención de nombres

Los nombres de variables de instancia, los argumentos y las variables temporales, por convención, comienzan con una letra minúscula.

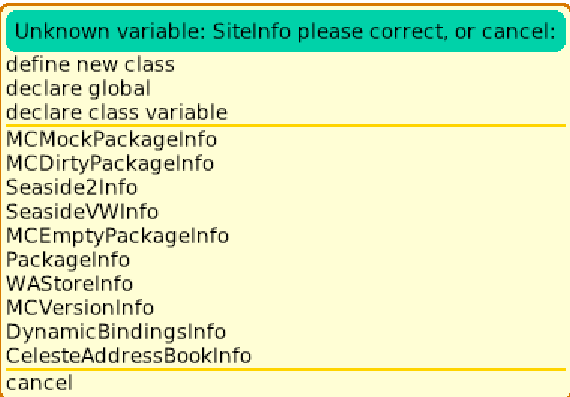
Las variables de clase, las variables globales y los nombres de clases comienzan con una letra mayúscula.

El lugar ideal para leer todas las convenciones e idioms que se usan en Smalltalk es el libro “Smalltalk with Style” (ver bibliografía).

Aceptamos y vemos como reacciona el ambiente:



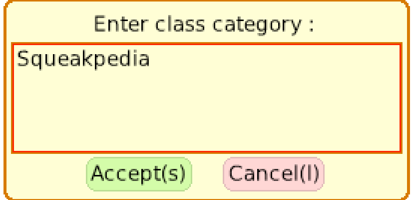
El ambiente nos dice que no sabe de una variable llamada `siteInfo`, así que nos ofrece 2 alternativa: crear una variable temporal o crear una variable de instancia, en este caso elegimos crear una variable de instancia.



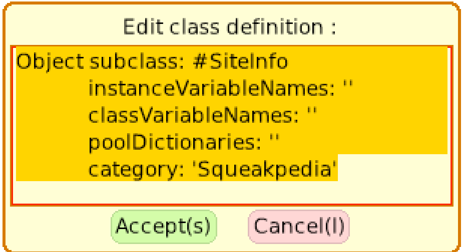
Ahora el ambiente nos dice que no sabe que es `SiteInfo` y nos ofrece varias alternativas. En el primer grupo de alternativas (las 3 que están antes de la primera línea) nos ofrece crear una clase, una variable global o una variable de clase. En el segundo grupo de opciones el ambiente cree que cometimos un error de tipeo y nos ofrece alternativas parecidas a lo que ingresamos nosotros, pero válidas. Y finalmente, nos ofrece cancelar la operación.

Nosotros queremos crear una nueva clase, así que seleccionamos la opción `'define new class'`.

Al hacerlo el ambiente nos pregunta en que categoría de clases queremos poner la nueva clase, nosotros ingresamos `'Squeakpedia'`.



Al aceptar el ambiente nos muestra el mensaje que va a enviar para crear la clase, dándonos la oportunidad de cambiar algo:



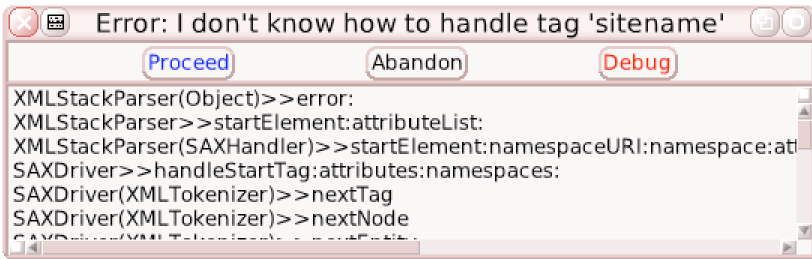
Nosotros aceptamos la plantilla sin introducir cambios.

Ahora el depurador nos muestra el método recién aceptado, y el syntax-highlight nos confirma que ahora `siteInfo` y `SiteInfo` son cosas conocidas. También vemos en el Inspector del receptor (los 2 paneles abajo, a la izquierda) que tenemos una nueva variable de instancia. Y si tenemos un Browser de Clases abierto con la categoría de clases `'Squeakpedia'` seleccionada, veremos que también tenemos una nueva clase.

Es importante remarcar que tanto la creación de métodos, de variables de instancia y la creación de una nueva clase ocurrió mientras estábamos depurando la aplicación, es decir: modificamos el ambiente mientras lo estamos usando.

En este momento tengo ganas de decir “prueben esto en su lenguaje de preferencia”, pero no sería políticamente correcto decirlo así que sigamos con el ejemplo.

Presionamos el botón `Proceed` para seguir ejecutando la aplicación.



Ahora el parser nos dice que no sabe como manejar el tag `<sitename>`. Si analizamos el XML de ejemplo, vemos que ese tag se usar para contener un texto. El parser nos permite manejar esos casos de forma especial. Instruimos al parser diciéndole que `<sitename>` es un textTag. Pedimos el depurador (presionando el botón `Debug`) y elegimos el método `Mediawiki class>>fromFileNamed:` (que es el método donde instruimos al parser como procesar los tags), y agregamos lo siguiente:

```
"instruimos al parser como procesar los text-tags"
parser onTextTag: 'sitename' send: #siteName:.
```

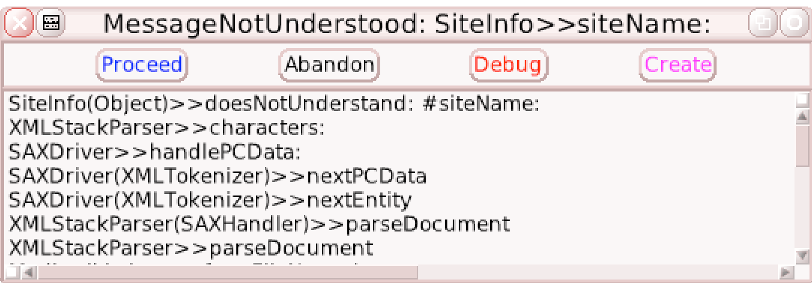
Y, ya que estamos, podemos completar algunos text-tags más dejando el método de la siguiente forma:

```
fromFileNamed: aString
   "Crea una nueva instance del receptor desde el archivo llamado aString"

   | archivo parser instancia |

   "abre el archivo"
   archivo := FileStream readOnlyFileNamed: aString.

   "crea la instancia del receptor"
   instancia := self new.

   "instancia el parser"
   parser := XMLStackParser on: archivo.

   "la instancia recién creada es el primer objeto de la pila"
   parser push: instancia.

   "instruimos al parser sobre como procesar los tags"
   parser onTag: 'mediawiki' send: #onMediawiki:.
   parser onTag: 'siteinfo' send: #onSiteinfo:.
```

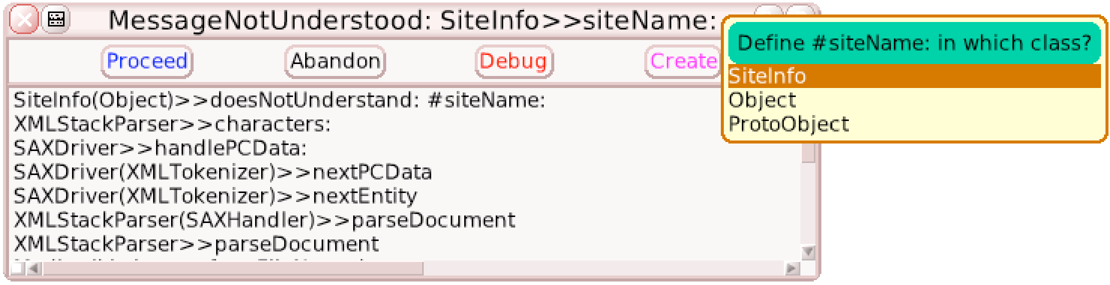
```
parser onTag: 'namespaces' send: #onNamespaces:.
parser onTag: 'namespace' send: #onNamespace:.
parser onTag: 'page' send: #onPage:.
parser onTag: 'revision' send: #onRevision:.
parser onTag: 'contributor' send: #onContributor:.

"instruimos al parser como procesar los text-tags"
parser onTextTag: 'sitename' send: #siteName:.
parser onTextTag: 'base' send: #base:.
parser onTextTag: 'generator' send: #generator:.
parser onTextTag: 'case' send: #case:.
parser onTextTag: 'namespace' send: #name:.
parser onTextTag: 'title' send: #title:.
parser onTextTag: 'id' send: #id:.
parser onTextTag: 'timestamp' send: #timestamp:.
parser onTextTag: 'ip' send: #ip:.
parser onTextTag: 'text' send: #text:.
parser onTextTag: 'comment' send: #comment:.

"le pedimos al parser que haga su trabajo"
parser parseDocument.

"cerramos el archivo"
archivo close.

"devolvemos la instancia al remitente del mensaje"
^ instancia
```

Aceptamos los cambios, y la ventana del depurador ahora queda así:

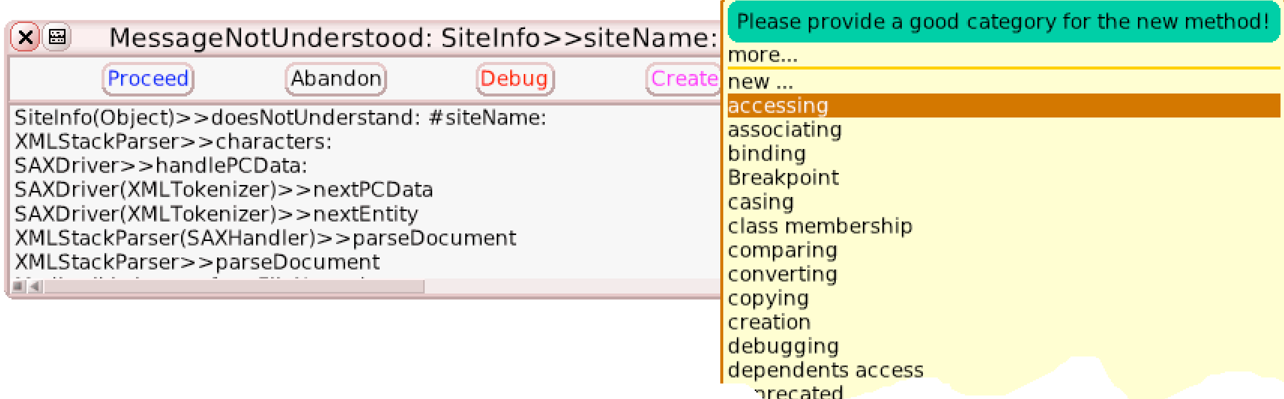
Nuevamente estamos listos para continuar presionando el botón `Proceed`.



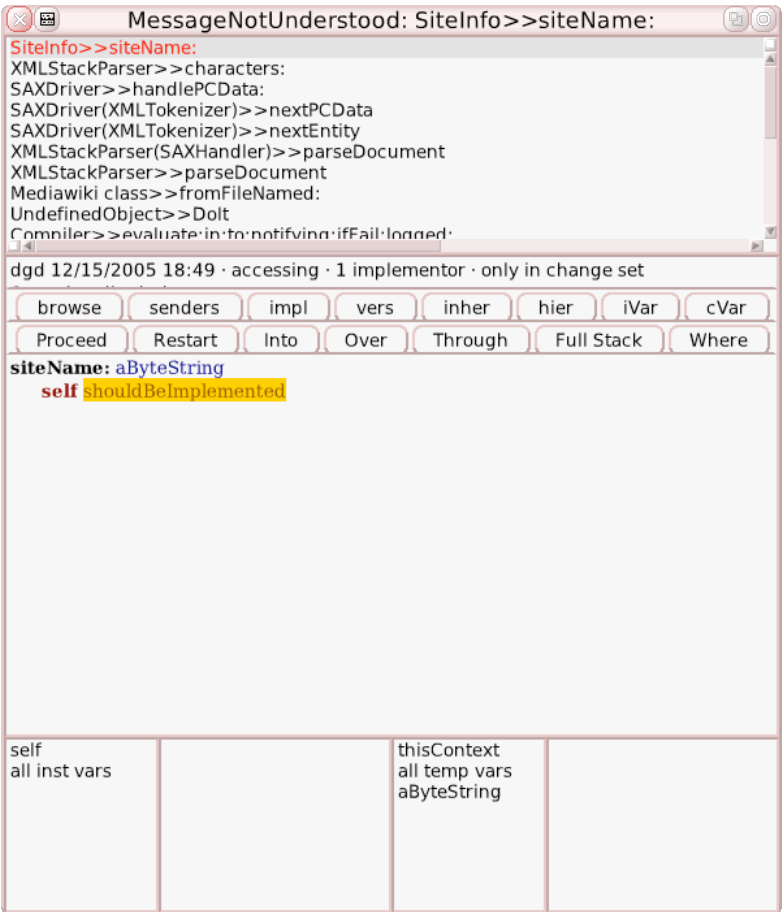
Elegimos crear un método (presionando el botón `Create`) en la clase `SiteInfo`



en la categoría de métodos llamada `'accessing'`



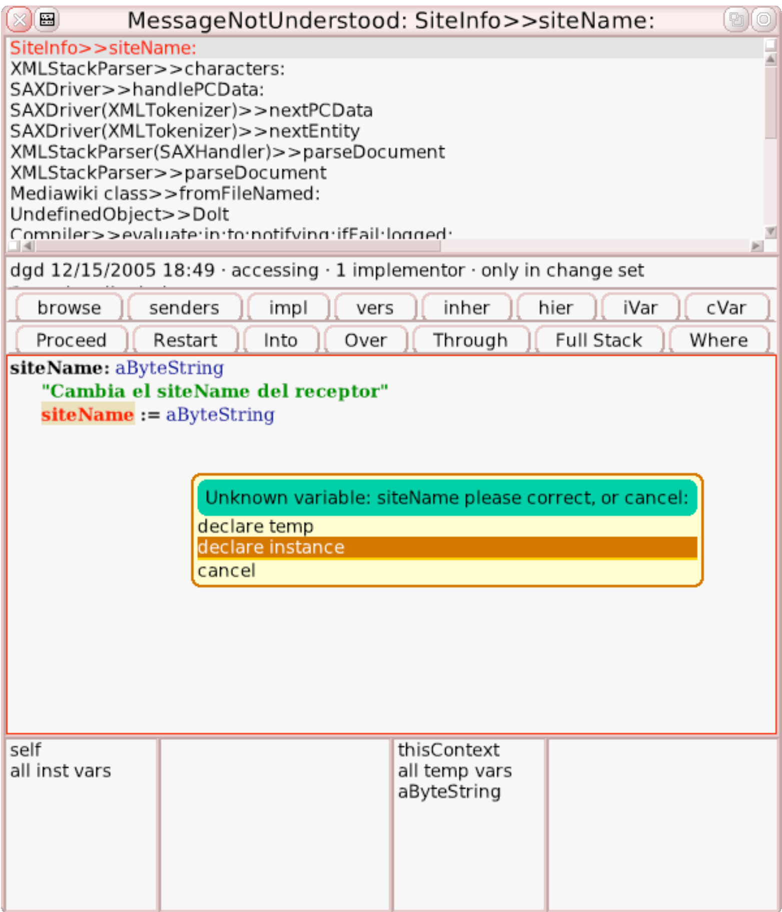
Ahora tenemos que implementar el método en el depurador, tal y como lo hicimos anteriormente



Escribimos lo siguiente:

```
siteName: aByteString
    "Cambia el siteName del receptor"
    siteName := aByteString
```

Nuevamente el syntax-highlight nos indica, con el color rojo, que `siteName` es algo desconocido. Aceptamos los cambios y, ante la pregunta, escogemos crear una variable de instancia:



Y ya estamos listos, nuevamente, para continuar (botón `Proceed`).

Ahora implementamos el método `#base:` de la misma forma que recién implementamos el método `#siteName:`, creando también una variable de instancia.

```
base: aByteString
    "Cambia el base del receptor"
    base := aByteString
```

Y continuamos (con `Proceed`).

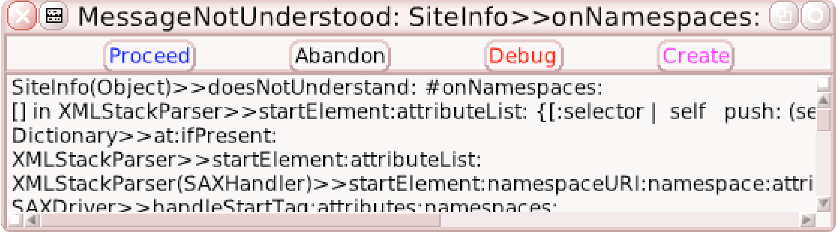
Repetimos el proceso para el método `#generator:`

```
generator: aByteString
    "Cambia el generator del receptor"
    generator := aByteString
```

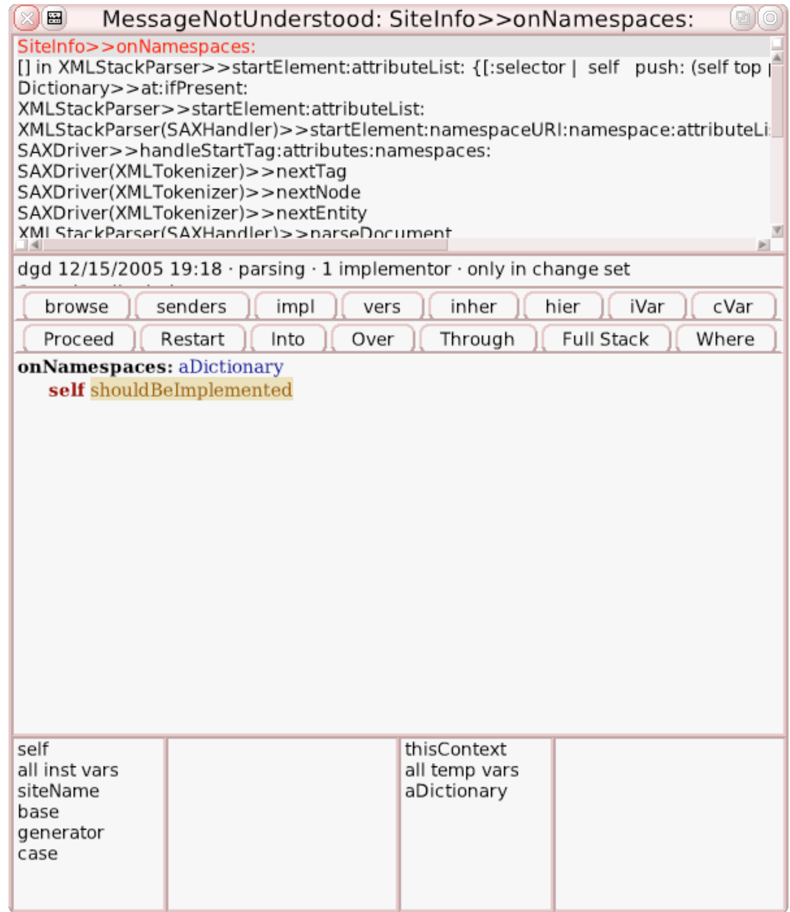
Y para el método `#case:`

```
case: aByteString
   "Cambia el case del receptor"
   case := aByteString
```

Y nos detenemos con más atención para el método `#onNamespaces:`



Creamos el método en la clase `SiteInfo`, en una nueva categoría llamada `'parsing'`



Si analizamos un poco el XML de ejemplo, veremos que el tag `<namespaces>` (en plural) contiene varios tags `<namespace>` (en singular), así que necesitamos una colección para poder guardar los varios namespace que contiene un Mediawiki.

Dicho lo dicho, implementamos el método de la siguiente manera (creando la variable de instancia `namespaces` cuando nos pregunte)

```
onNamespaces: aDictionary
   "Crea una colección vacía donde guardar los namespaces venideros"
   namespaces  := Set new.
```

Colecciones

Smalltalk cuenta con un framework de colecciones muy poderoso. No es de extrañar ya que lleva más de 20 años de depuración.

Contamos con numerosos tipos de colecciones (Bag, Set, OrderedCollection, Dictionary, Array, etc) y, a su vez, con un rico y extenso protocolo de mensajes.

A continuación enumero algunos de los mensajes que se les puede enviar a todas las colecciones.

`#add:` Agrega un objeto (dado como argumento) a la colección.

`#addAll:` Agrega todos los elementos de la colección dada como argumento a la colección receptora del mensaje.

`#remove:` Remueve un determinado objeto de la colección. Genera un error si el elemento no es parte de la colección.

`#remove:ifAbsent:` Remueve un determinado elemento de la colección. Evalúa el bloque dado si el elemento no es parte de la colección.

`#removeAll:` Remueve todos los elementos del receptor que están contenidos en la colección dada en el argumento.

`#do:` Evalúa el bloque dado como argumento por cada elemento contenido en la colección. Este es el principal mensaje de las colecciones y prácticamente todo el resto de mensajes están implementados usando el **`#do:`** de alguna forma. Es interesante ver la implementación de, por ejemplo, los mensajes **`#select:`**, **`#collect:`**, **`#anyOne`**, **`#detect:ifNone:`**, **`#do:separatedBy:`**, etc.

`#do:separatedBy:` Evalúa el primer bloque dado como argumento por cada elemento contenido en la colección y evalúa el segundo bloque entre elementos.

`#select:` Evalúa el bloque dado por cada elemento del receptor como argumento. Colecciona los elementos en los cuales el bloque evalúa a `true` (verdadero) en una

colección del tipo del receptor. Responde esa colección nueva como resultado.

#reject: Similar a **#select:** pero colecciona los elementos para los cuales el bloque evalúa a false.

#collect: Evalúa el bloque dado por cada elemento del receptor como argumento. Colecciona los resultados en una colección del tipo del receptor. Responde esa colección nueva como resultado.

#size Responde el tamaño de la colección.

#anyOne Responde algún elemento de la colección.

#atRandom Responde uno de los elementos del receptor de forma aleatoria.

#detect: Evalúa el bloque dado con cada elemento del receptor. Devuelve el primer elemento donde el bloque evalúa a `true`. Genera un error si ningún elemento es encontrado.

#detect:ifNone: Evalúa el bloque dado con cada elemento del receptor. Devuelve el primer elemento donde el bloque evalúa a `true`. Evalúa el otro bloque dado si ningún elemento es encontrado y devuelve el resultado de esa evaluación.

#isEmpty Responde si el receptor está vacío y no contiene elementos.

#includes: Responde si el objeto dado es un elemento del receptor.

#includesAllOf: Responde si todos los elementos de la colección dada están incluidos en la colección receptora.

#includesAnyOf: Responde si algunos de los elementos de la colección dada está incluido en la colección receptora.

#allSatisfy: Responde `true` si el bloque dado se evalúa a `true` por todos los elementos de la colección receptora.

#anySatisfy: Responde true si el bloque dado se evalúa a true por algunos de los elementos de la colección receptora.

#noneSatisfy: Responde `true` si el bloque dado se evalúa a `false` por todos los elementos de la colección receptora.

#occurrencesOf: Responde cuantas veces está incluido el objeto dado en la colección receptora.

Colecciones – Set

El Set es un tipo de colección que no mantiene ningún orden sobre sus elementos y que no permite que un objeto esté contenido más de una vez.

Un Set representa el concepto matemático de conjunto, donde no tiene sentido decir que un elemento está más de una vez ni tiene sentido hablar de orden.

Ejemplos:

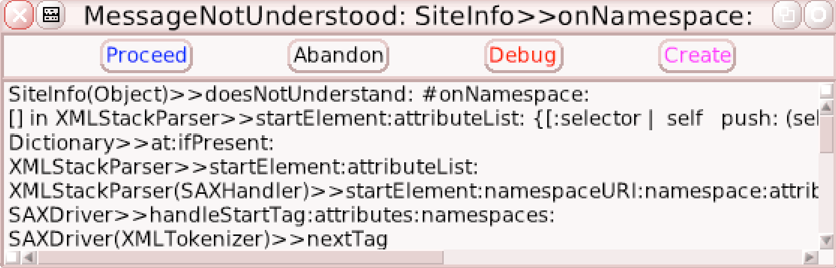
```
| set |
set := Set new.
set add: 'un string'.
```

```
set add: 'otro string'.
set add: 'un string'.
set add: 'otro string'.
set add: 'un string'.
set explore.


"convertir colecciones de diferente tipo a Set para remover los
duplicados"
#(5 4 1 2 2 2 1 2 1 2 3 4 3 2 3 4 5) asSet.
'un string que tiene muchos caracteres' asSet.


"Los 2 sets contienen sólo un elemento"
(Set with: 1) = (Set with:1 with:1).

"El orden no importa"
(Set with: 1 with: 2) = (Set with:2 with:1).
```

Valor de retorno por defecto

Los métodos que no tengan un valor de retorno explícito – usando el carácter ^ - terminan devolviendo al receptor.

Como el método anterior devuelve al receptor (si no se especifica una respuesta, el método termina retornando `self`) el objeto `SiteInfo` será también responsable de responder al mensaje `#onNameSpace:`



Creamos (presionando el botón `Create`) el método en la clase `SiteInfo`, en la categoría `'parsing'`

```
onNamespace: aDictionary
   "Crea un nuevo namespace y lo guarda en la colección de namespaces del
receptor"
   | namespace |
   namespace := Namespace new.
   namespaces add: namespace.
```

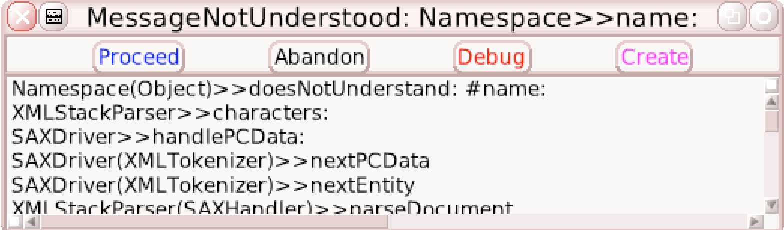
```
^ namespace
```

Colecciones – mensaje #add:

Agrega un objeto (dado por argumento) a la colección.

Otra vez el rojo del syntax-highlight nos avisa que no sabe que es `Namespace`. Para resolverlo creamos una nueva clase, en la categoría de clases `'Squeakpedia'` y continuamos la ejecución presionando el botón `Proceed`.

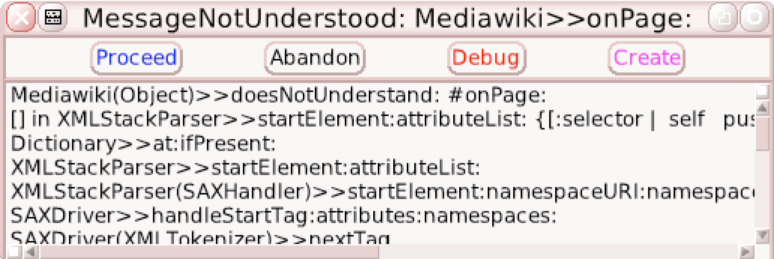
Ahora el ambiente nos avisa de un nuevo error:



Y lo resolvemos creando el método en la clase `Namespace`, en la categoría de métodos `'accessing'`, de la siguiente manera:

```
name: aByteString
   "Cambia el name del receptor"
   name := aByteString.
```

Creamos la variable de instancia `name` al aceptar. Continuamos la ejecución con el botón `Proceed`.



Es el momento de procesar la creación de la primera página del Mediawiki.

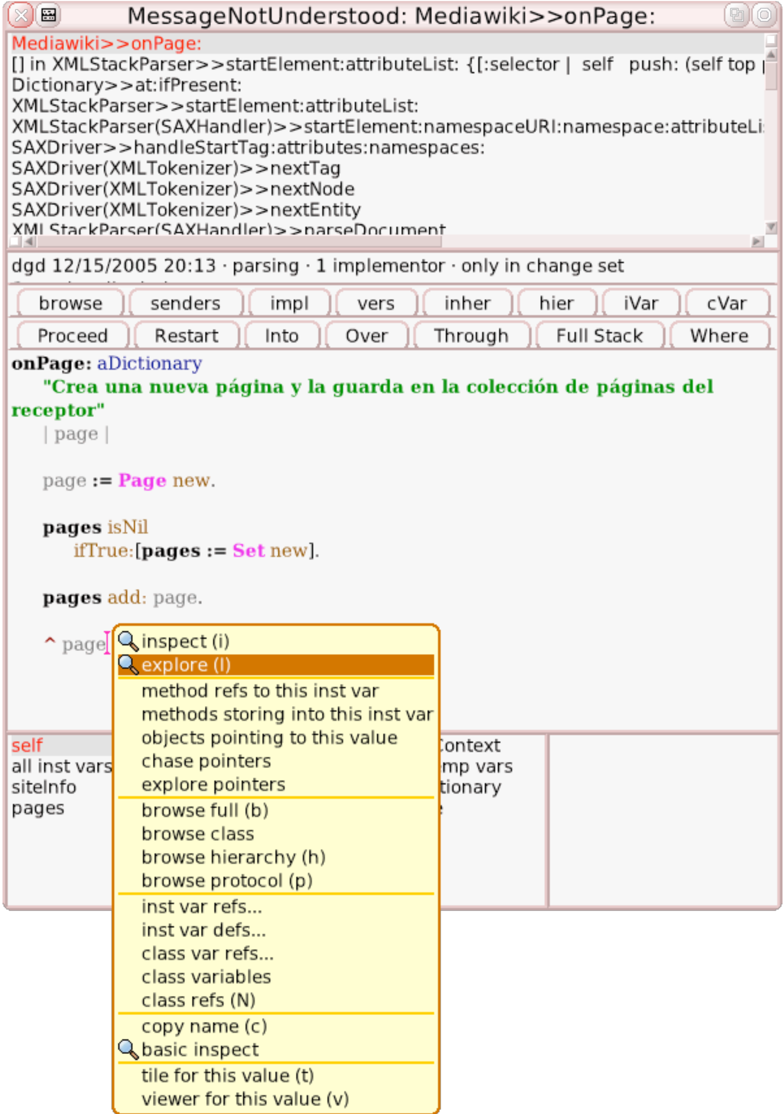
Creamos el método en la clase `Mediawiki`, en la categoría `'parsing'`, siguiendo el procedimiento que ya usamos varias veces, y lo implementamos así:

```
onPage: aDictionary
   "Crea una nueva página y la guarda en la colección de páginas del receptor"
   | page |
   page := Page new.

   pages isNil
      ifTrue:[pages := Set new].

   pages add: page.

   ^ page
```

Tenemos que crear la clase `Page` (en la categoría de clases `'Squeakpedia'`) y la variable de instancia `pages` al aceptar el método.

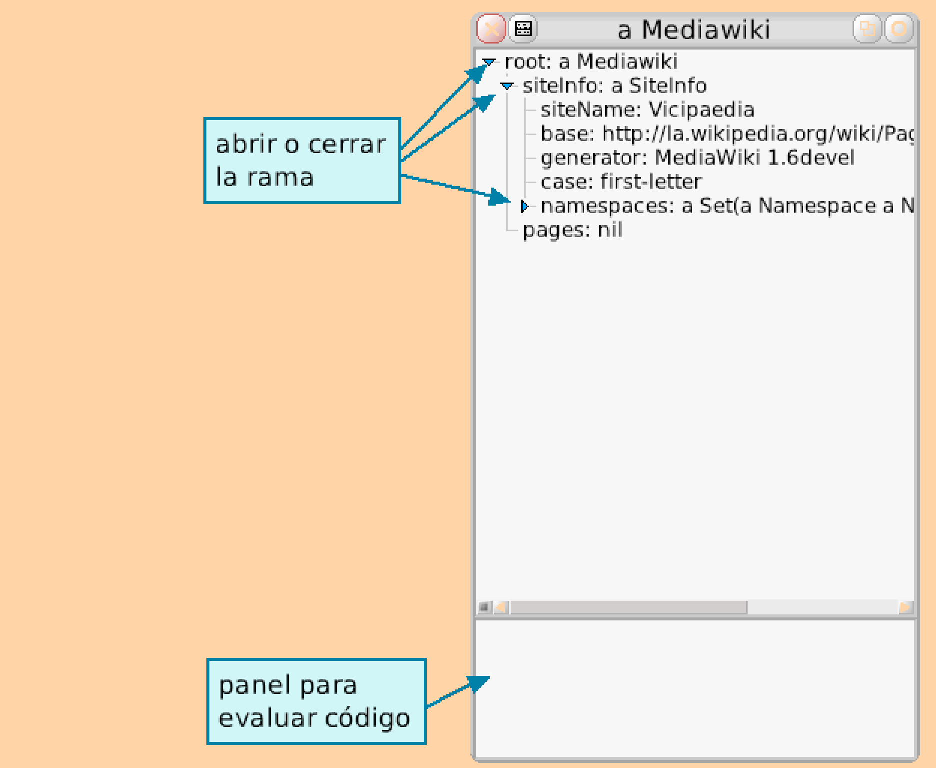
Debido a una inconsistencia en el formato del XML de la Wikipedia, tenemos que crear la colección de páginas usando lazy-initialization ya que los tags `<page>` no están contenidos en un tag `<pages>` y no podemos crear la colección siguiendo el mismo procedimiento que usamos para crear la colección de namespaces de la clase `SiteInfo`.

Antes de continuar, vamos a inspeccionar un poco el estado de nuestros objetos. Para hacerlo seleccionamos la opción `self` de los paneles que forman el Inspector del receptor, y desde el menú contextual (botón azul) seleccionamos la opción `'explore (I)'`:



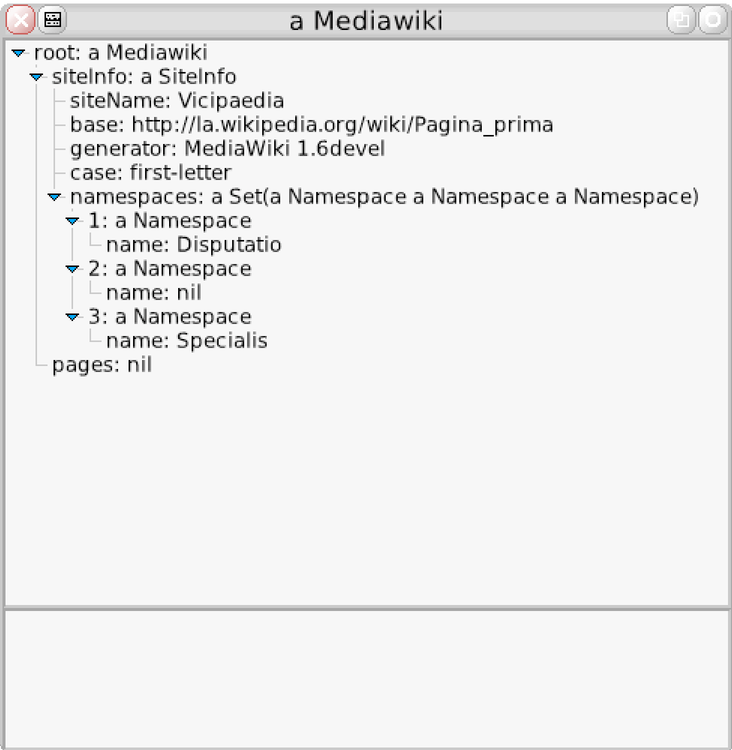
Explorador

El Explorador es una herramienta que cumple las mismas funciones que el Inspector, pero con un formato de árbol que nos permite ir navegando la estructura que forman nuestros objetos.



Las pequeños triángulos azules nos permiten abrir o cerrar las ramas para analizar la estructura a nuestro antojo.

En el panel de código se puede evaluar cualquier sentencia Smalltalk, la única consideración es que la variable especial `self` apuntará al objeto seleccionado en el panel de arriba y, si no hubiese ningún objeto seleccionado, apuntará al objeto que está siendo explorado (la raíz del árbol).



Abriendo y cerrando las distintas ramas del árbol del explorador podemos validar si nuestro importador está funcionando correctamente. Vemos que hay 3 namespaces en la colección, vemos que la variable de instancia `pages` está en `nil`, y que otros datos del `SiteInfo` reflejan bien los datos del XML.

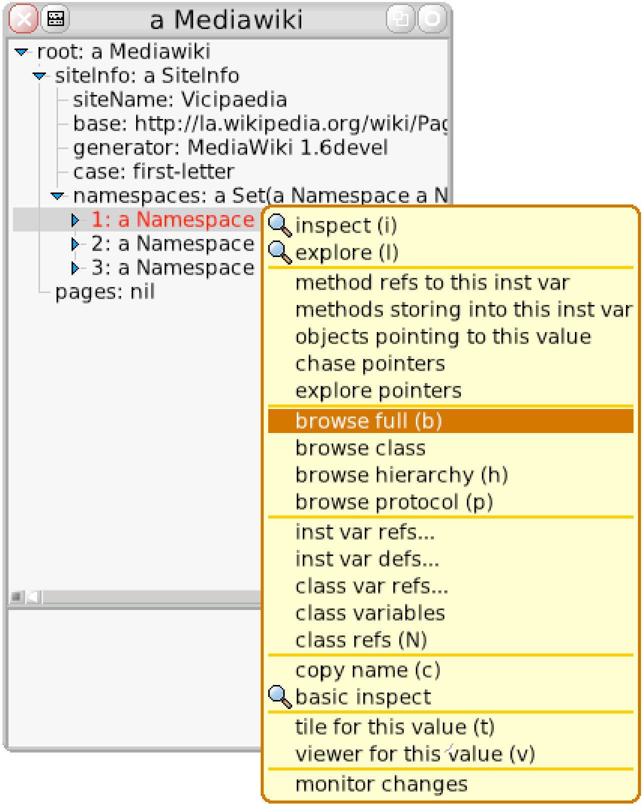
Sin embargo, si comparamos los datos que tienen nuestros objetos con el XML de ejemplo, vemos que los objetos `Namespace` no tienen el dato 'key' que viene como atributo del tag.

```
...
    <namespaces>
      <namespace key="-1">Specialis</namespace>
      <namespace key="0" />
      <namespace key="1">Disputatio</namespace>
    </namespaces>
...
```

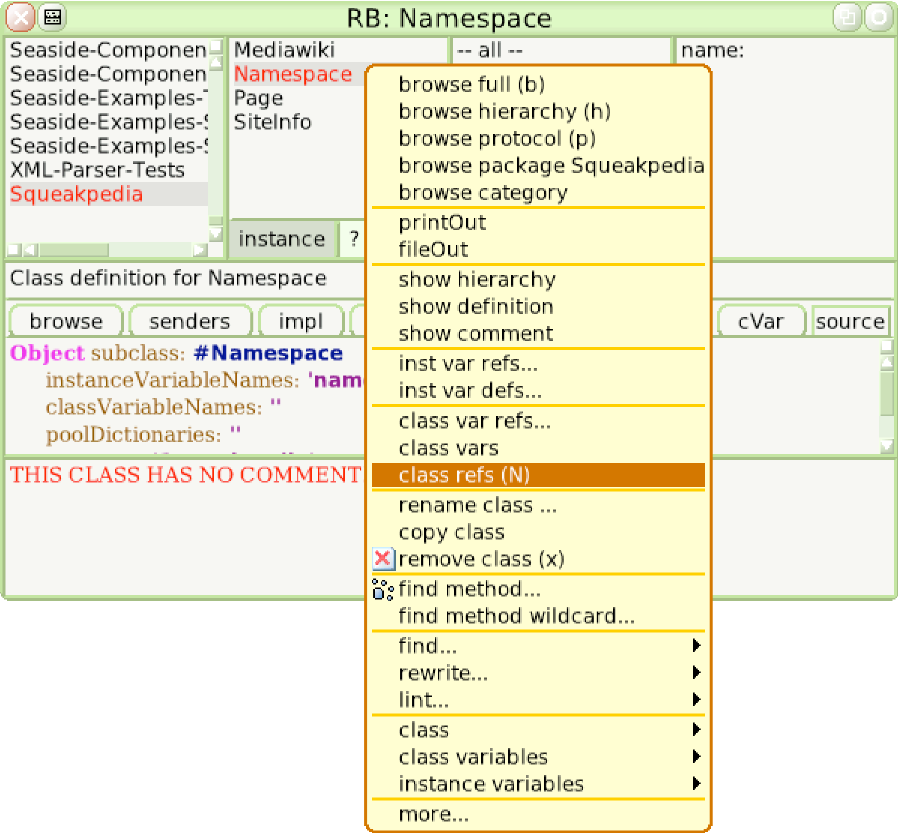
Pareciera que nos olvidamos de considerar ese dato en el método donde creamos los objetos `Namespace`.

Como somos un poco vagos, y el Smalltalk es ideal para gente vaga como nosotros, vamos a utilizar las herramientas del ambiente para localizar el lugar donde se instancian esos objetos. Pare eso seguimos el siguiente procedimiento:

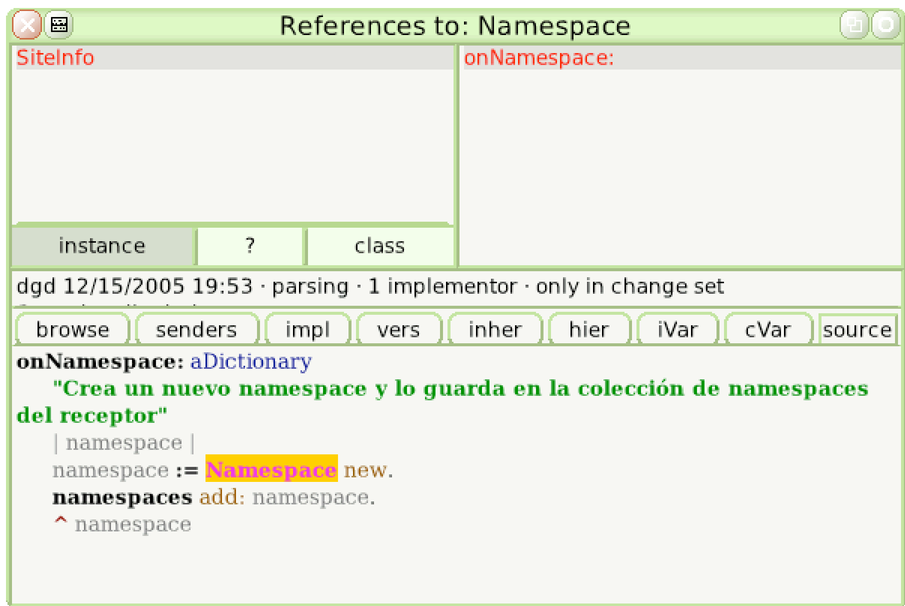
Buscamos, en el Explorador, uno de los objetos `Namespace` y lo seleccionamos. Pedimos el menú contextual sobre ese objeto (con botón azul) y seleccionamos la opción `'browse full (b)'` (podríamos haber presionado ALT-b).



Esto nos abre el Browser de Clases apuntando a la clase del objeto seleccionado. Pedimos el menú contextual sobre la clase `SiteInfo` y seleccionamos la opción `'class refs (N)'` (podríamos haber presionado ALT-MAYÚSCULA-n).



Esa opción nos abre un tipo de Browser que muestra los métodos donde se hace referencia a la clase `Namespace`. Seleccionamos la clase en el panel superior izquierdo, y luego el método en el panel superior derecho y vemos el código del método que instancia los `Namespace`.

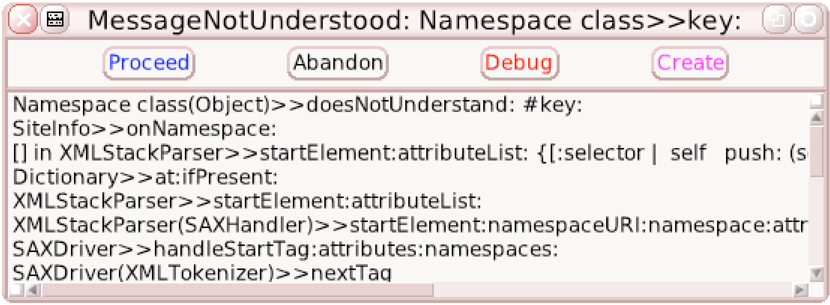


¡Encontramos el error! El dato del atributo 'key' del tag `<namespace>` viene dentro del

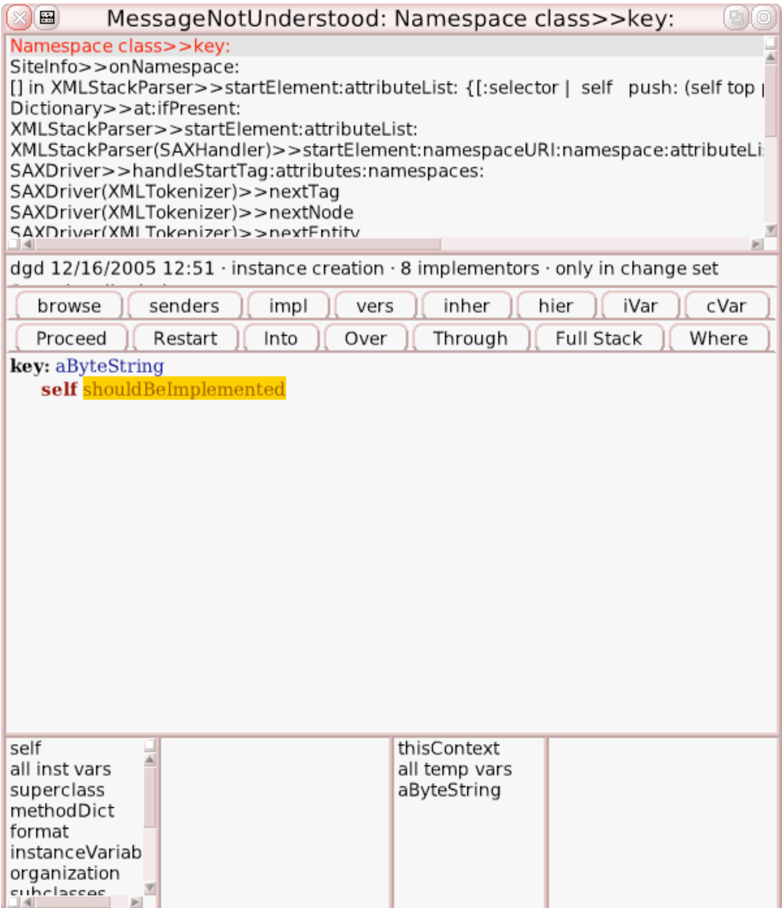
diccionario que recibimos por argumento, pero nosotros lo ignoramos. Corregimos el método de la siguiente forma:

```
onNamespace: aDictionary
   "Crea un nuevo namespace y lo guarda en la colección de namespaces del receptor"
   | namespace |
   namespace := Namespace key: (aDictionary at: 'key').
   namespaces add: namespace.
   ^ namespace
```

Podemos cerrar algunas ventanas que ya no usaremos como el Browser de Referencias, el Browser de Clases y el Explorador. Y, en el todavía abierto Depurador, buscamos el contexto referente al método `Mediawiki class>>fromFileNamed:` y presionamos la opción `Restart` para volver a comenzar la ejecución y, acto seguido, presionamos la opción `Proceed`.



¿Qué ocurrió? Ocurrió que la forma de instanciar los objetos `Namespace` cambió y nosotros no actualizamos la clase. Seleccionamos la opción `Create` y decimos que el método se cree en la clase `Namespace class` (notar el class, ahora estamos creando un método de clase y no de instancia) y en la categoría `'instance creation'`:

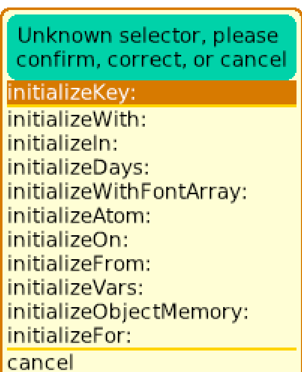


Implementamos el método de la siguiente manera:

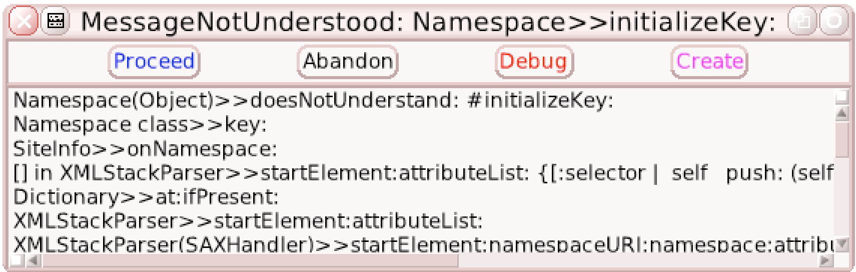
```
key: aByteString
   "Devuelva una nueva instancia del receptor con el key dado"
   ^ self new initializeKey: aByteString
```

Nuevamente el syntax-highlight nos indica que algo no se conoce, en este caso nos dice que no hay en todo el Smalltalk ningún método que se llame `#initializeKey:` y que, probablemente, nos hallamos equivocado de mensaje. En nuestro caso no nos equivocamos, sino que todavía no hemos creado el método correspondiente.

Aceptamos el método y, ante la propuesta de alternativas de mensaje, seleccionamos la primera que es la que realmente nos interesa ya que no es un error el mensaje que nosotros ingresamos.



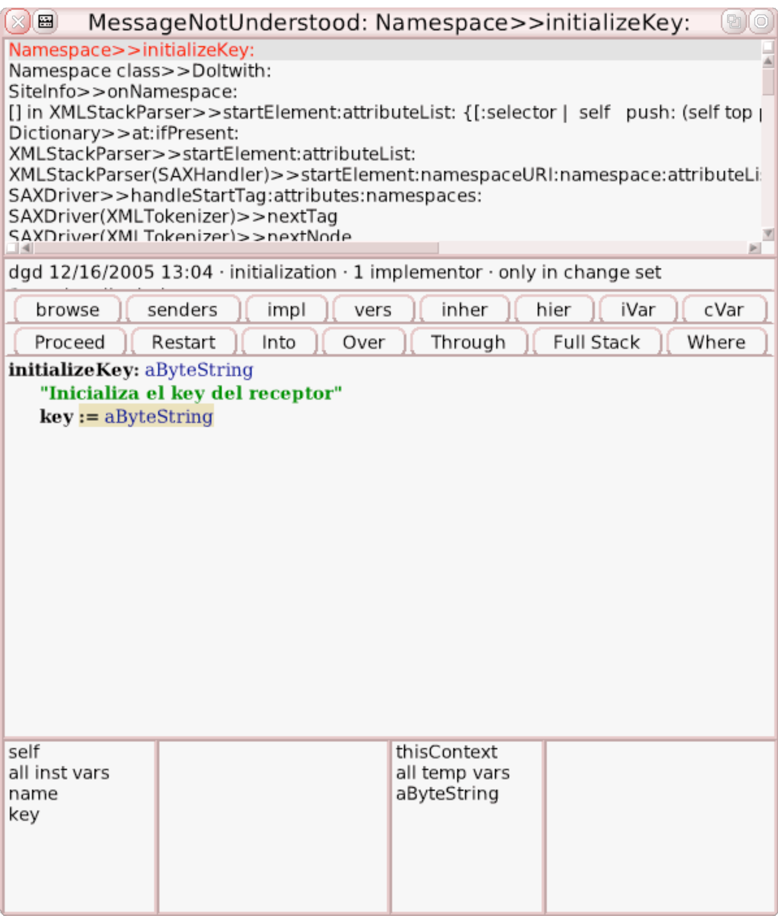
Presionamos el botón `Proceed` para continuar y vemos que pasa:



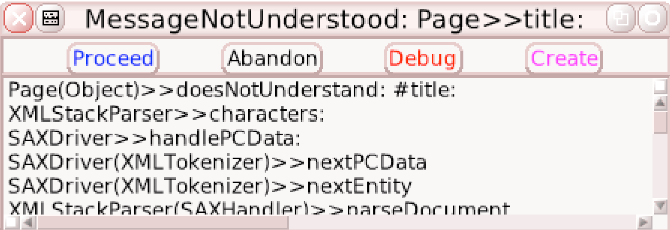
Ahora nos dice que un objeto de clase `Namespace` no entiende el mensaje `#initializeKey:`, lo creamos en esa misma clase, dentro de una nueva categoría de métodos llamada `'initialization'` y lo implementamos así:

```
initializeKey: aByteString
   "Inicializa el key del receptor"
   key := aByteString
```

Otra vez el syntax-highlight nos dice que no conoce nada con el nombre key, así que aceptamos y creamos una variable de instancia.



Y seguimos con la ejecución presionado el botón `Proceed`.



Creamos el método en la clase `Page`, dentro de la categoría `'accessing'`, de la siguiente forma:

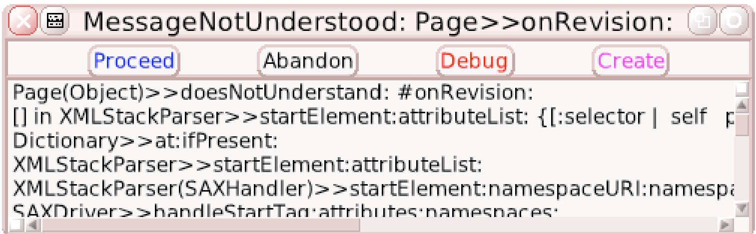
```
title: aByteString
   "Cambia el title del receptor"
   title := aByteString
```

Y creamos la variable de instancia `title` al aceptar el método.

Hacemos lo mismo para el método `#id:` creando, también, la variable de instancia al aceptar

```
id: aByteString
   "Cambia el id del receptor"
   id := aByteString
```

Ahora nos detenemos un poco para implementar el método `#onRevision:`



Creamos el método, en la clase `Page`, en una nueva categoría llamada `'parsing'`

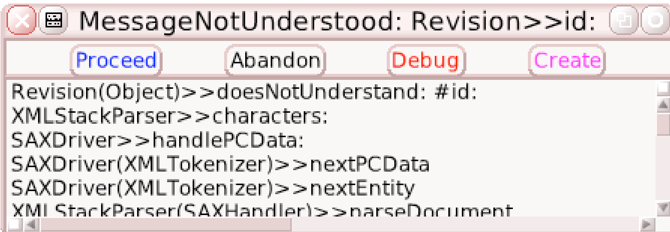
```
onRevision: aDictionary
   "Crea una nueva revision guardándola en la coleccion de revisiones del
receptor"

   | revision |

   revisions isNil
      ifTrue:[revisions := OrderedCollection new].

   revision := Revision new.

   revisions add: revision.

   ^ revision
```

Al aceptar este método, creamos la variable de instancia `revisions` y la clase `Revision` dentro de la categoría de clases `'Squeakpedia'`, luego continuamos la ejecución con el botón `Proceed`.

Colecciones – OrderedCollection

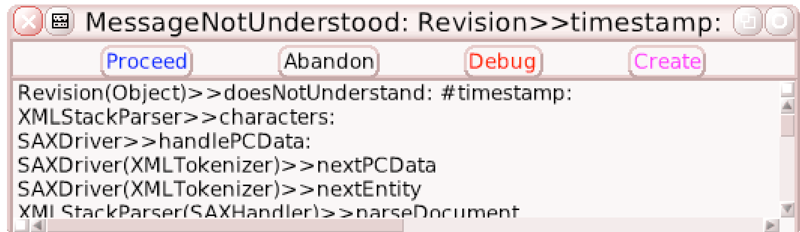
Las colecciones `OrderedCollection` mantienen a los elementos en el mismo orden en que fueron insertados.

Al ser una colección que tiene sentido de orden, expone un protocolo que incluye mensajes para acceder a los elementos por su posición. Ejemplos: `#first`, `#second`, `#third`, `#last`, `#allButFirst`, `#allButLast`, `#addLast:`, `#addFirst:`, `#add:after:`, `#addBefore:`, etc.



Ahora nos toca completar la implementación de la clase `Revision`, primero creamos el método `#id:` , en la categoría `'accessing'`, con su correspondiente variable de instancia.

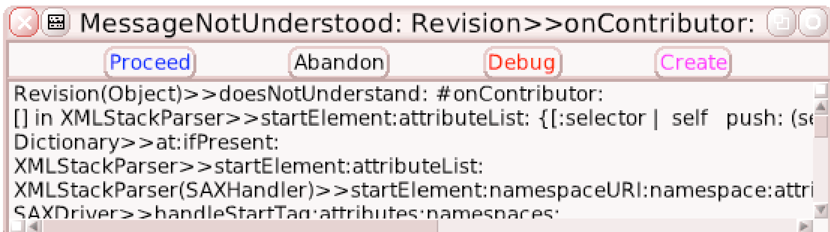
```
id: aByteString
    "Cambia el id del receptor"
    id := aByteString
```



Luego implementamos el método `#timestamp:`, en la clase `Revision`, dentro de la categoría `'accessing'` y creamos, al aceptar el método, la variable de instancia:

```
timestamp: aByteString
    "Cambia el timestamp del receptor"
    timestamp := aByteString
```

Y continuamos

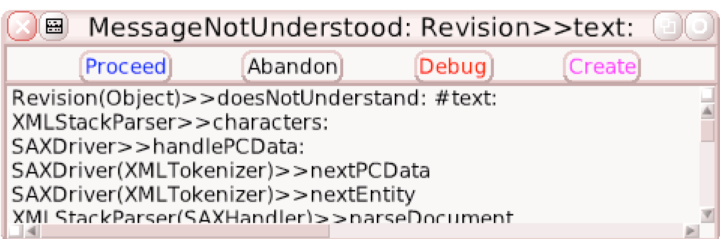


Implementamos el método `#onContributor:`, en la categoría `'parsing'`, y creamos la variable de instancia `contributor` y la clase `Contributor` en la categoría `'Squeakpedia'`.

```
onContributor: aDictionary
    "crea un contributor y lo guarda en el receptor"
    contributor := Contributor new.
    ^ contributor
```

Ahora nos toca implementar el método #ip: en la clase Contributor, y creamos la variable de instancia correspondiente.

```
ip: aByteString
    "Cambia el ip del receptor"
    ip := aByteString
```



Y llegamos al, tal vez, método más importante del importador.

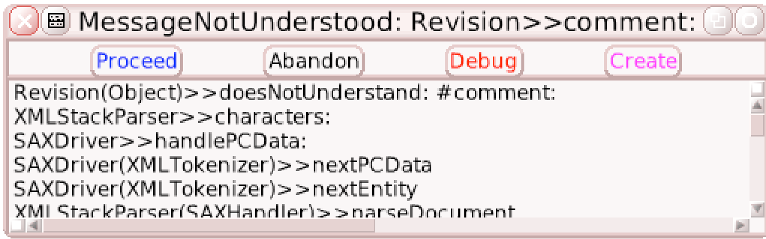
Creamos el método en la clase `Revision`, en la categoría `'accessing'`, de la siguiente forma:

```
text: aByteString
    "Cambiar el text del receptor"
    text := aByteString asByteString zipped.
```

Y creamos, al aceptar el método, la variable de instancia correspondiente.

Debido a que el mayor porcentaje de los datos serán precisamente los textos, guardamos los datos comprimidos para ocupar menos espacio. Por esa causa es que le enviamos el mensaje `#zipped` al String después de asegurarnos que el argumento sea un `ByteString`.

Continuamos:

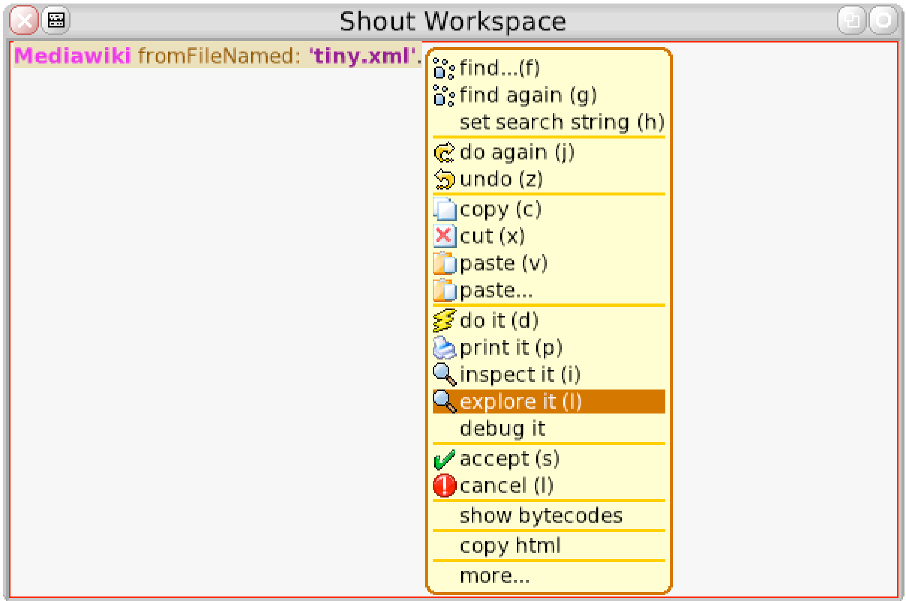


Creamos el método `#comment:`, en la clase `Revision`, en la categoría `'accessing'`:

```
comment: aByteString
   "Cambia el comment del receptor"
   comment := aByteString
```

Y creamos la variable de instancia al aceptar.

¡TERMINAMOS! ¿No lo creen? Vamos a explorar el resultado de la exploración para validar que los datos del XML de ejemplo estén en nuestros objetos.



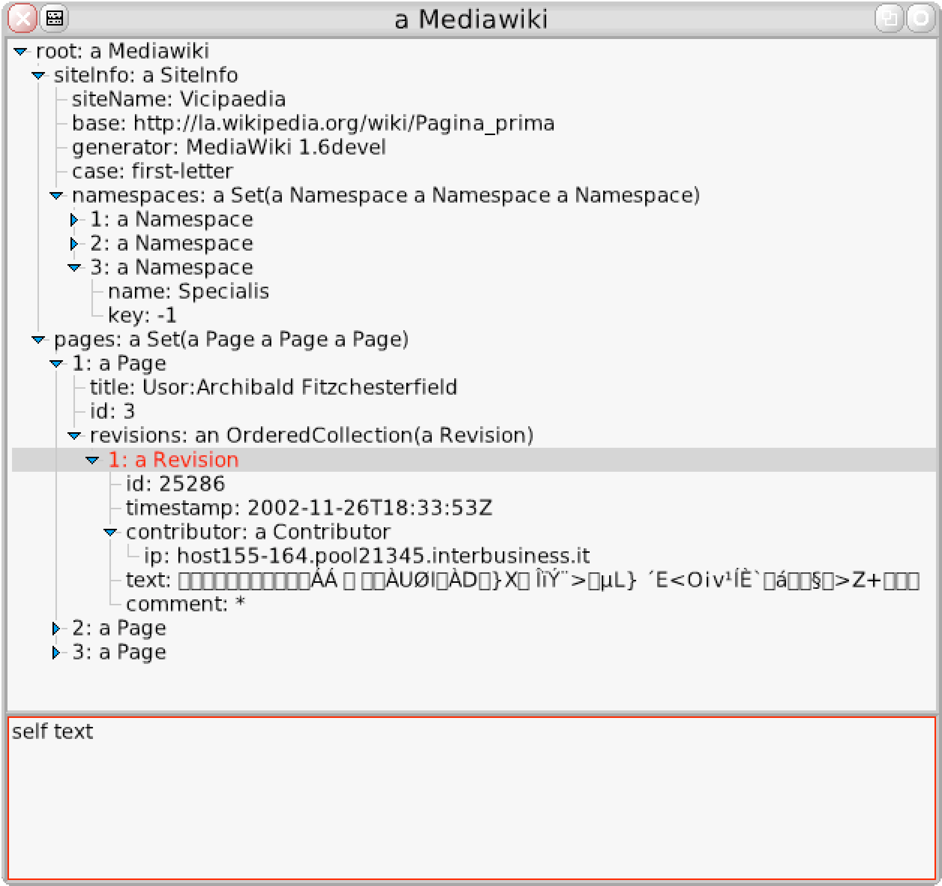
Y navegamos un poco a través de los objetos generados:

```
a Mediawiki
root: a Mediawiki
  siteInfo: a SiteInfo
    siteName: Vicipaedia
    base: http://la.wikipedia.org/wiki/Pagina_prima
    generator: MediaWiki 1.6devel
    case: first-letter
    namespaces: a Set(a Namespace a Namespace a Namespace)
      1: a Namespace
      2: a Namespace
      3: a Namespace
        name: Specialis
        key: -1
  pages: a Set(a Page a Page a Page)
    1: a Page
      title: Usor:Archibald Fitzchesterfield
      id: 3
      revisions: an OrderedCollection(a Revision)
        1: a Revision
          id: 25286
          timestamp: 2002-11-26T18:33:53Z
          contributor: a Contributor
            ip: host155-164.pool21345.interbusiness.it
          text: [illegible]
          comment: *
    2: a Page
    3: a Page
```

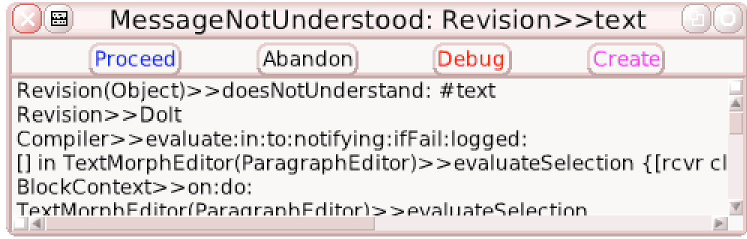
Pareciera que está todo bien, lo único que no es fácil de validar así a simple vista es el texto que guardamos comprimido.

Para validar los textos, vamos a implementar un método que descomprima el texto.

Seleccionamos, en el explorador, algún objeto de clase `Revision` y escribimos en el panel inferior lo siguiente:



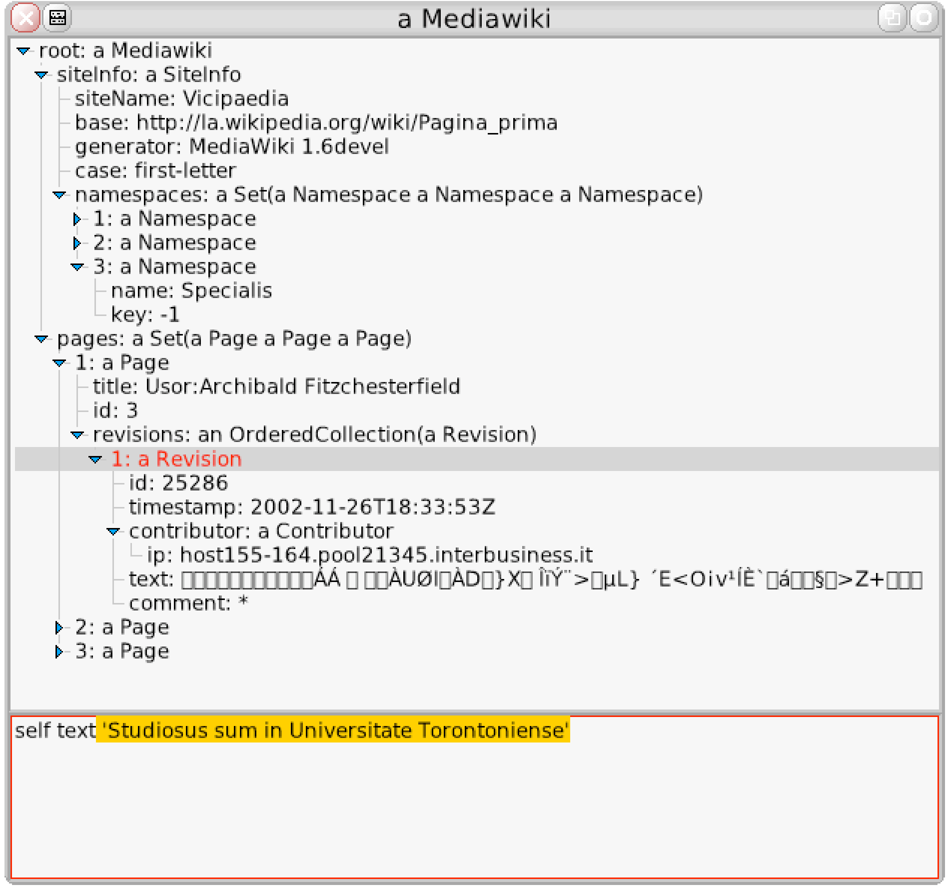
Y lo evaluamos con print it `'print it (p)'` (o ALT-p).



Lo implementamos, en la categoría `'accessing'`, de la siguiente forma:

```
text
    "Responde el text del receptor"
    ^ text unzipped
```

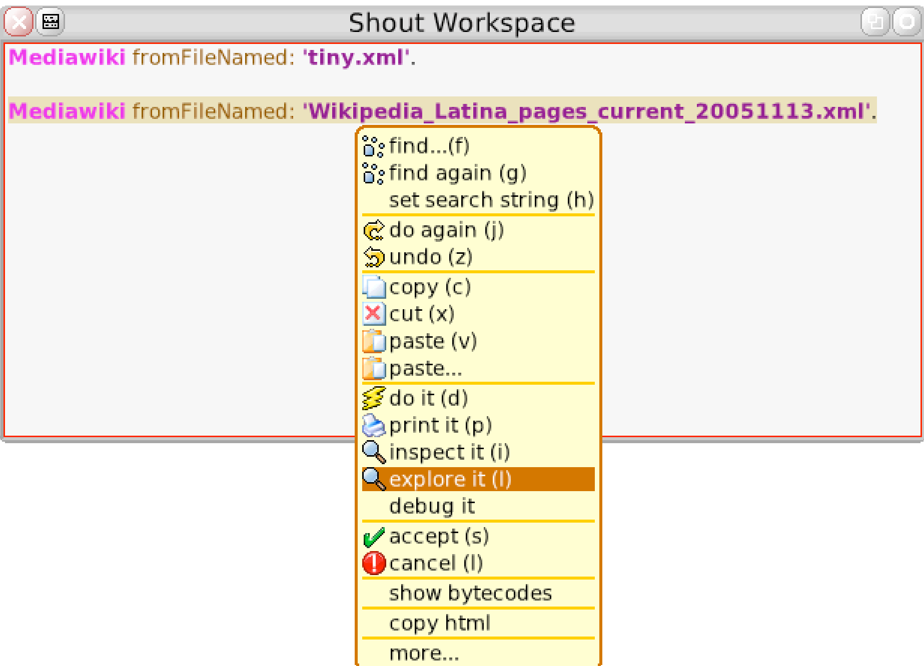
Y presionamos el botón `Proceed` para continuar:



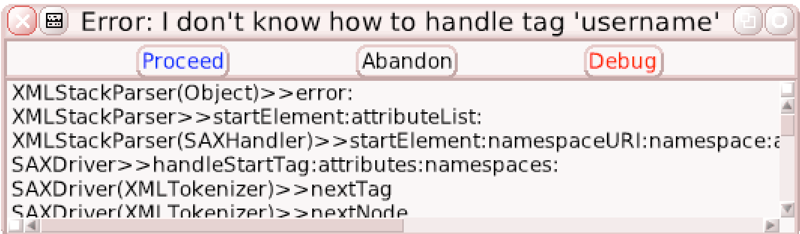
Y, pareciera que, el texto está bien.

Ahora vamos a validar el importador procesando algún archivo real y más grande, comencemos importando la Wikipedia en Latín. Es posible que, procesando un archivo más grande, encontremos que falta implementar algún otro tag que no teníamos en el ejemplo anterior.

Tecleamos lo siguiente, en el `Workspace`, y seleccionamos la linea correspondiente al archivo de la Wikipedia en Latín y lo ejecutamos explorando el resultado.



Y obtenemos lo siguiente:



Parece que tendremos que implementar algunos métodos más para procesar este ejemplo. Eso no debería ser problema, ya que a esta altura manejamos el Depurador mejor que a un hornos microondas.

Ya hemos trabajado sobre un error como este, lo que está pasando es que el parser nos dice que no sabe como reaccionar al tag `<username>`.

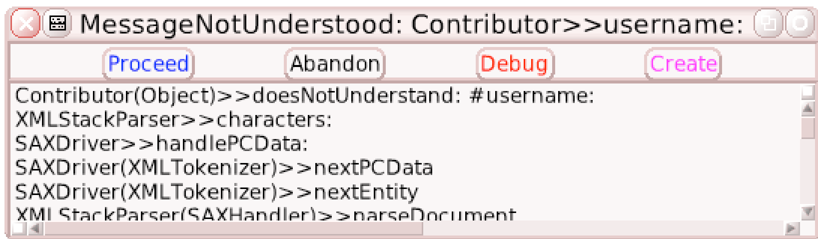
```
...
  <page>
    <title>Arithmetica</title>
    <id>5</id>
    <revision>
      <id>45467</id>
      <timestamp>2005-10-21T01:42:12Z</timestamp>
```

```
      <contributor>
        <username>FlaBot</username>
        <id>509</id>
      </contributor>
      <minor />
      <comment>Bot: Fixing wiki syntax</comment>
      <text xml:space="preserve">== De vocabulo Arithmeticae disciplinae ==...</text>
    </revision>
  </page>
...
```

Buscamos en el XML el lugar donde está el tag, y vemos que es otro dato que pueden tener los `Contributor`. Instruir al parser, en el método `Mediawiki class>>fromFileNamed:`, sobre un nuevo text-tag de la siguiente forma:

```
parser onTextTag: 'username' send: #username:.
```

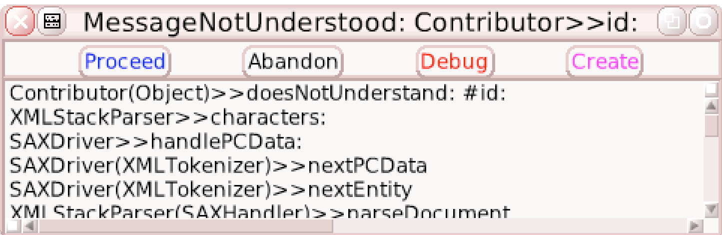
Recomenzamos la ejecución



Y ahora debemos implementar el método `#username:` como ya sabemos (clase `Contributor`, categoría de métodos `'accessing'`, variable de instancia, etc.)

```
username: aByteString
   "Cambia el username del receptor"
   username := aByteString
```

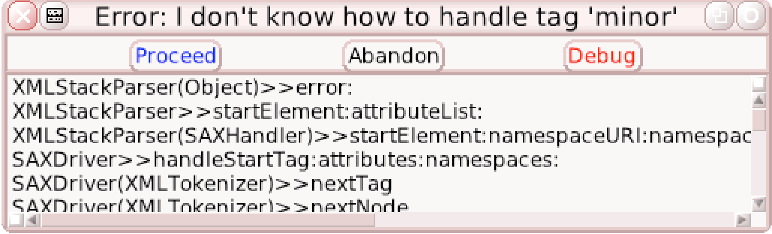
Continuamos:



Mismo procedimiento para el método `#id:`

```
id: aByteString
   "Cambia el id del receptor"
   id := aByteString
```

Y continuamos



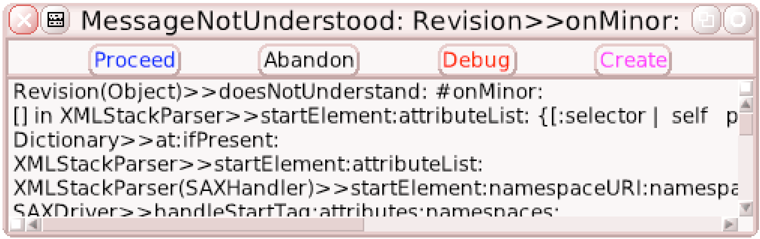
Apareció otro tag que desconoce el parser.

```
...
  <page>
    <title>Arithmetica</title>
    <id>5</id>
    <revision>
      <id>45467</id>
      <timestamp>2005-10-21T01:42:12Z</timestamp>
      <contributor>
        <username>FlaBot</username>
        <id>509</id>
      </contributor>
      <minor />
      <comment>Bot: Fixing wiki syntax</comment>
      <text xml:space="preserve">== De vocabulo Arithmeticae disciplinae ==</text>
    </revision>
  </page>
...
```

Algunas revisiones tienen una especie de marca que indica que son revisiones-menores. Instruimos al parser con lo siguiente:

```
parser onTag: 'minor' send: #onMinor:.
```

Y recomenzamos el proceso.



Ahora implementamos el método `#onMinor:`, en la categoría `'parsing'`, de la siguiente manera:

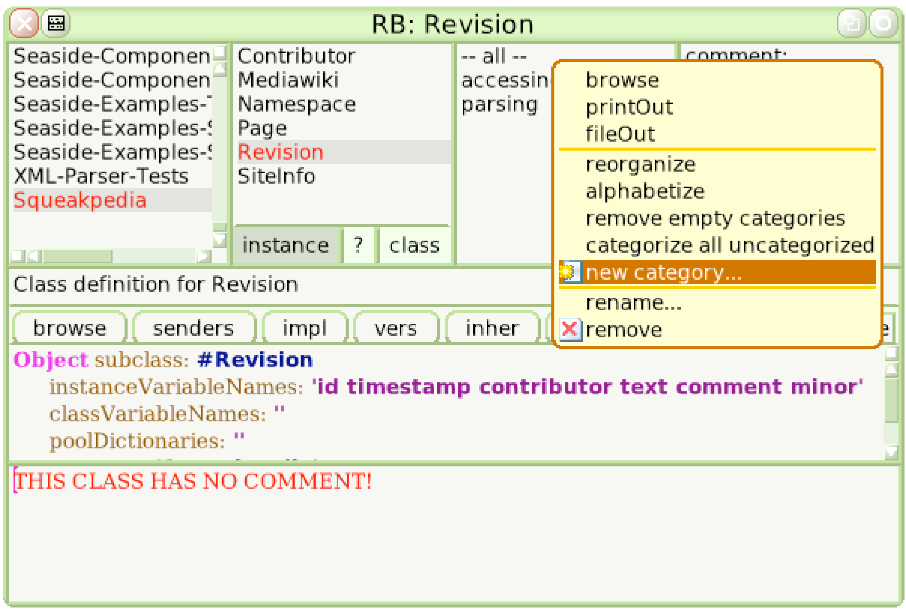
```
onMinor: aDictionary
   "Marca al receptor como minor"
   minor := true.
```

Y creamos la variable de instancia al aceptar.

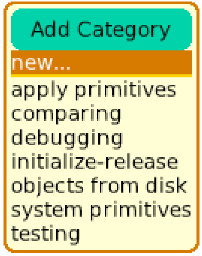
Debido a la forma en que el XML nos avisa que alguna revisión es menor (está el tag sólo para las revisiones menores, y no hay nada que indique que una revisión no es menor), debemos tomar una consideración extra. ¿Qué pasará con la variable `minor` en las revisiones no-menores? Con el código tal y cual lo tenemos en este punto, tendremos un `true` para las revisiones menores y un `nil` para las otras. Para dejar consistente esto, podemos inicializar la variable en `false`.

Para hacerlo, podemos utilizar el Browser de Clases. ¡Tanto usar el Depurador que casi nos olvidamos del Browser!.

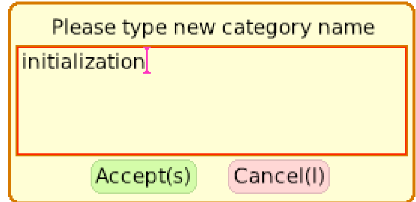
Abrimos el browser (Menú del Mundo >> `'open...'` >> `'browser (b)'` o ALT-b), seleccionamos la categoría de clases `'Squeakpedia'`, seleccionamos la clase `Revision` y en el panel correspondiente a las categorías de métodos invocamos el menú contextual (botón azul) y escogemos la opción `'new category...'`



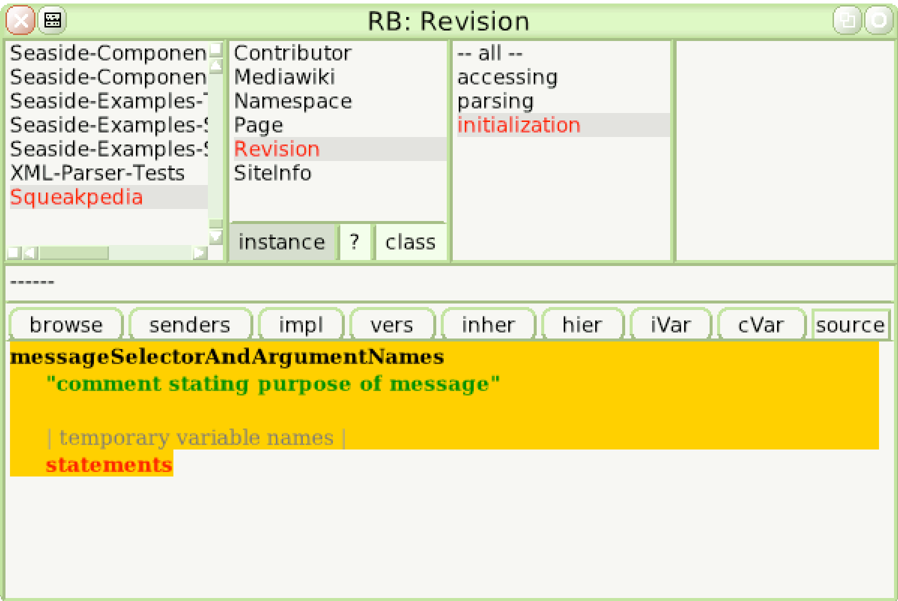
Elegimos la opción `'new..'`



Y tecleamos: initialization.



Al aceptar, el Browser nos muestra una plantilla de método en el panel inferior:



Reemplazamos la plantilla con lo siguiente:

```
initialize
    "Inicializa el receptor"
    minor := false.
```

Y aceptamos.

Inicialización de objetos

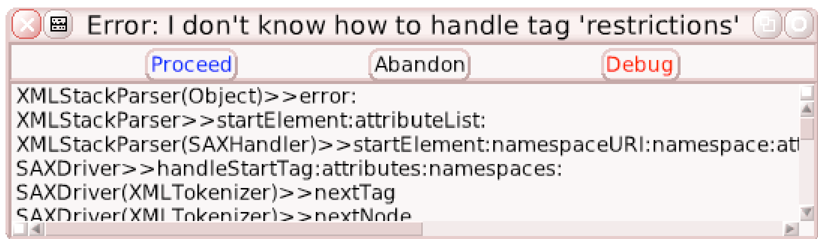
En las versiones actuales de Squeak las instancias reciben el mensaje `#initialize` inmediatamente después de ser creadas.

En otros dialectos de Smalltalk esto puede no ser así.

Si estuviese usando otro dialecto de Smalltalk, o una versión vieja de Squeak, se debería enviar el mensaje `#initialize` desde el método de clase `#new` de forma similar a lo que hicimos con el mensaje de clase `#key:` y el método de instancia `#initializeKey:` de la clase `Namespace`.

Ya podemos recomenzar la ejecución.

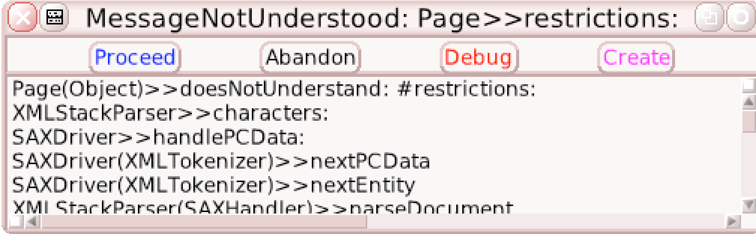
Ahora el proceso se toma un buen rato (y esa es buena señal ya que quiere decir que está procesando el XML sin problemas) y nos dice:



Buscamos en el XML el tag, y vemos que algunas páginas tienen restricciones.

```
...
  <page>
    <title>Pagina prima</title>
    <id>328</id>
    <restrictions>sysop</restrictions>
    <revision>
      <id>46417</id>
      <timestamp>2005-10-29T06:52:57Z</timestamp>
      <contributor>
        <username>MycÄ s</username>
        <id>79</id>
      </contributor>
      <comment>svg</comment>
      <text xml:space="preserve">&lt;!-- Please note that most </text>
    </revision>
  </page>
...
```

Instruimos al parser para que procese los tags `<restrictions>` como un text-tag enviando el mensaje `#restrictions:` y recomenzamos.



Creamos el método y la variable de instancia, y seguimos.

Al cabo de un buen rato (aproximadamente 1 minuto y 40 segundos en mi máquina) el proceso termina correctamente.

Antes de abandonar el ejemplo, vamos a mirar un poco donde se está consumiendo el tiempo.

Para eso utilizamos la clase `MessageTally` de la siguiente forma:

```
MessageTally spyOn:[Mediawiki fromFileNamed: 'Wikipedia_Latina_pages_current_20051113.xml']
```

Y obtenemos lo siguiente:

MessageTally

La herramienta MessageTally recolecta información sobre la cantidad de tiempo que se consume en los métodos y nos las muestra de una forma clara para poder optimizar nuestro código en lo que respecta a la velocidad de ejecución.

Si prestamos atención a los resultados del `MessageTally`, vemos que el 63.3% del tiempo se está consumiendo al comprimir los textos.

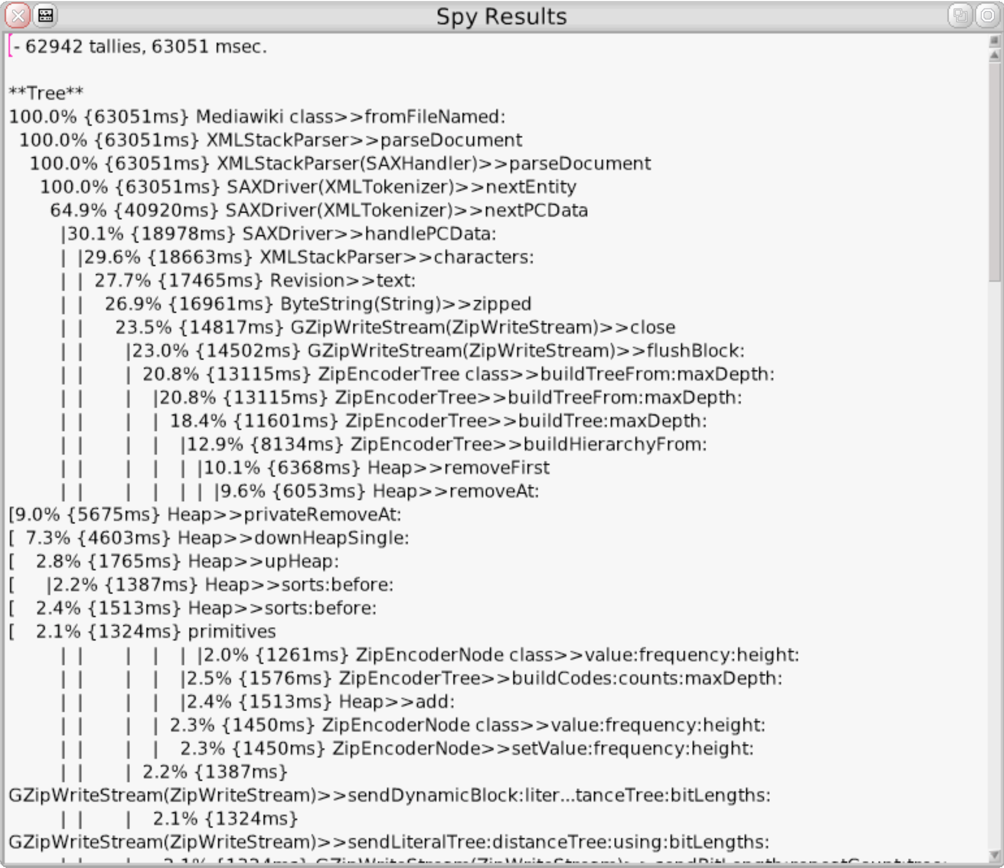
```
 |64.4% {74572ms} XMLStackParser>>...
 | | 64.0% {74109ms} Revision>>text:
 | |   63.3% {73298ms} ByteString(String)>>zipped
 | |     46.8% {54192ms} GZipWriteStream(ZipWriteStream)
 | |       |46.4% {53729ms} GZipWriteStream(ZipWriteStrea
          | 42.1% {48750ms} ZipEncoderTree class>>bui
                  {48750ms} ZipEncoderTree>>bu
```

Como es bien sabido que no tiene sentido comprimir textos chicos, vamos a modificar un poco el método `#text:` para comprimir solamente los textos que midan más de 2 Kbytes.

```
text: aByteString
   "Cambiar el text del receptor"

   | string |

   string := aByteString asByteString.

   text := (string size > 2048)
             ifTrue:[string zipped]
             ifFalse:[string].
```

No es necesario que modifiquemos el método `#text` porque el método `#unzziped` es lo suficientemente inteligente como para darse cuenta que el string no está comprimido.

Volvemos a ejecutar el importador, usando el `MessageTally`, para ver el impacto de nuestro cambio.



Hemos bajado el tiempo de 115 segundos a 63 segundos, nada mal para un cambio de 3 o 4 líneas de código.

Otra opción, para hacer esta optimización, hubiese sido usar el `TimeProfileBrowser`. Para usarlo hay que evaluar lo siguiente:

```
TimeProfileBrowser spyOn:[Mediawiki fromFileNamed: 'Wikipedia_Latina_pages_current_20051113.xml'].
```

TimeProfilerBrowser

El `TimeProfileBrowser` muestra los mismos datos que el `MessageTally`, pero los presenta en un Browser que permite modificar directamente los métodos.

Con esto damos por concluido el ejemplo del importador de Wikipedia. Sólo levantamos los datos al Squeak y no hicimos nada con ellos, pero recuerden que el objetivo del libro es enseñar a utilizar un ambiente de Smalltalk y no hacer una aplicación. Elegimos hacer una aplicación no trivial para tener la oportunidad de mostrar diferentes opciones del ambiente, pero no porque quisiéramos hacer la aplicación en si. De cualquier forma, quedan todos invitados a completar esta aplicación dotándola de nuevas funcionalidades.

Algunas ideas:

– Generar, con los datos importados, un documento open-office con la Wikipedia en formato libro. La idea es llegar, con la información libre, incluso a la gente que no tenga ordenadores.

- Hacer una interfaz gráfica y poner todos los datos, con la aplicación, en un CD. De esa forma llegaríamos a la gente que si tiene ordenador, pero que no tiene acceso a Internet.
- Investigar con técnicas de text-mining e indexar la información de mejores maneras que las ofrecidas hoy en día por la Wikipedia.

Motor de Workflow

El desarrollo del ejemplo anterior estuvo regido por el parseo de un archivo. El Smalltalk nos ofrece excelentes alternativas para interactuar con información que está fuera de la imagen (archivos, bases de datos tanto relacionales como de objetos, sockets de tcp, etc).

Sin embargo, cuando hacemos un sistema completamente contenido en la imagen, las ventajas se multiplican. Cuando manipulamos objetos de nuestra imagen nos olvidamos de, por ejemplo, abrir y cerrar conexiones, abrir y cerrar archivos, de situaciones de errores por modificación de la información por otros programas, etc. y sólo aplicamos las reglas de un ambiente de objetos: Objetos y mensajes.

Ahora desarrollaremos, también paso a paso, un pequeño motor de Workflow.

Workflow

El Flujo de trabajo (Workflow en inglés) es el estudio de los aspectos operacionales de una actividad de trabajo: cómo se estructuran las tareas, cómo se realizan, cuál es su orden correlativo, cómo se sincronizan, cómo fluye la información que soporta las tareas y cómo se le hace seguimiento al cumplimiento de las tareas. Generalmente los problemas de flujo de trabajo se modelan con redes de Petri.

Si bien el concepto de flujo de trabajo no es específico a la tecnología de la información, un parte esencial del software para trabajo colaborativo (groupware) en justamente el flujo de trabajo.

Para más información ver:

- http://es.wikipedia.org/wiki/Workflow
- http://en.wikipedia.org/wiki/Workflow

Vamos a utilizar una metodología muy similar al “Test Driven Development”.

Test Driven Development

El objetivo de todo programador debería ser el de generar *Código Limpio que Funciona*.

Existen numerosas razones para escribir *Código Limpio que Funciona (CLQF)*, algunas de ellas son:

- Generar CLQF es una forma predecible de desarrollar. Se puede saber cuando se termina el desarrollo y no nos preocupamos por un largo ciclo de depuración.

- El CLQF nos da la oportunidad de aprender todas las lecciones que el código tiene que decirnos.
- El CLQF brinda mejores oportunidades a los usuarios de nuestro software.
- El CLQF permite que nosotros confiemos en nuestros compañeros y que ellos confien en nosotros.
- Cuando escribimos CLQF, nos sentimos mejor con nosotros mismos.

Hay muchas fuerzas que nos dificultan escribir CLQF, entonces ¿cómo podemos generar CLQF?.

Esta metodología (Test Driven Development - TDD) consiste en hacer que los tests automáticos sean los que rigen el desarrollo. Para eso seguimos la siguiente reglas:

- Escribir nuevo código sólo si tenemos un test automático que falla.
- Eliminar la duplicación.

Aplicar esas 2 reglas imprimen el siguiente ritmo al proceso de desarrollo:

- **Rojo:** Escribir un test que falle.
- **Verde:** Hacer que el test funcione lo más rápidamente posible. Podemos cometer cuantos pecados queramos en esta etapa ya que el objetivo es salirse del rojo cuanto antes.
- **Refactorizar:** Eliminar toda la duplicación creada en el paso anterior.

Más información:

- En el libro: “Test-Driven Development by Example” (ver bibliografía).
- http://es.wikipedia.org/wiki/Tdd
- http://en.wikipedia.org/wiki/Test_driven_development

Ya que toda la funcionalidad se implementa sólo después de que un determinado test lo requiera, un sistema desarrollado de esta forma contiene decenas, si no centenares, de tests. La administración de los test es, pues, parte del desarrollo y se hace conveniente contar con herramientas que nos ayuden. Para escribir y administrar los tests usaremos el framework llamado SUnit.

SUnit

SUnit es la madre de todos los frameworks de Unit Testing.

Para más información:

- http://sunit.sourceforge.net/

- http://www.xprogramming.com/testfram.htm
- http://es.wikipedia.org/wiki/Prueba_unitaria
- http://en.wikipedia.org/wiki/Unit_test
- http://www.iam.unibe.ch/~ducasse/Programmez/OnTheWeb/Art8-SUnit.pdf
- http://www.iam.unibe.ch/~ducasse/Programmez/OnTheWeb/SUnitEnglish2.pdf

Para ayudarnos en la tarea de refactorización nos valdremos de otra herramienta: El Refactoring Browser.

Refactoring Browser

El Refactoring Browser es una variante del Browser de Clases que nos ofrece opciones para automatizar los operaciones típicas de refactoring.

Para más información ver: http://st-www.cs.uiuc.edu/users/brant/Refactory/

Refactoring

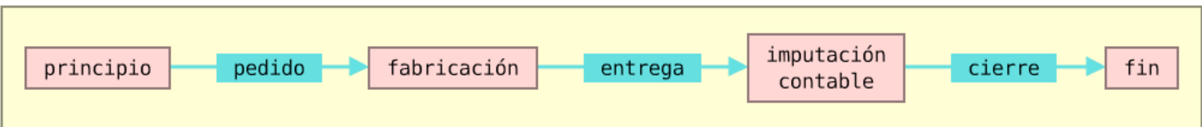
Se llama Refactoring al proceso de cambiar un sistema de software de tal forma que no se cambie el comportamiento visto desde afuera, pero se mejora su estructura interna para hacerlo más fácil (y más barato) de modificar.

Más información en el libro: "Refactoring – Improving the design of existing code" (ver bibliografía).

Comencemos con el ejemplo. Nuestro humilde motor de Workflow estará compuesto, principalmente, por 2 grandes grupos de objetos: La definición de los procesos, y las instancias de los procesos.

Las definiciones son la declaración de como tienen que ser los procesos. Allí decimos que, por ejemplo, los procesos llamados "Venta" comienzan después de recibir una orden de pedido. Que a continuación tenemos que emitir una orden de producción para que la fábrica se ponga en funcionamiento, luego informamos al departamento contable sobre la nueva deuda que el cliente tendrá con nuestra empresa, etc.

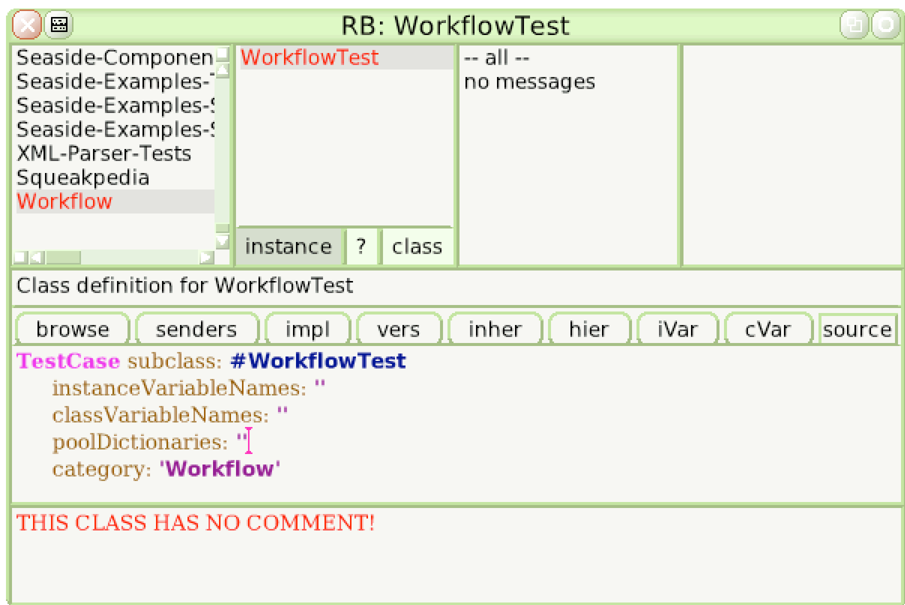
Las definiciones enumeran una serie de estados y las transiciones entre ellos.



Las instancias de procesos son las representaciones, en nuestro sistema, de una operación en curso. Siguiendo el ejemplo anterior podemos decir que una instancia de la definición “Venta” es cuando el cliente Juan Pérez nos pide 100 escritorios. Otra instancia, de la misma definición, es cuando otro cliente nos pide otro producto. Varias instancias corresponden a una única definición ya que la definición sólo indica como es el procedimiento y las instancias son una venta en particular.

Para empezar a escribir un poco de código (¡Ya tengo ganas después de tanta introducción!) comencemos con un test que pruebe la definición más simple que se nos ocurra y obtengamos el rojo.

Usamos el Browser de Clases para crear, primero, una categoría de clases llamada `'Workflow'` y luego creamos una subclase de `TestCase` llamada `WorkflowTest`.



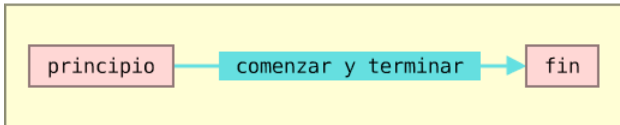
Ahora creamos una categoría de métodos llamada `'running'` y, dentro, creamos un método llamado `#testLaMasSimpleDefinición`.

Los métodos de testing comienzan por #test

Los métodos que comienzan con #test serán considerados métodos de testing por el framework SUnit. Cuando le digamos al framework que ejecute todos los test de una determinada clase, este invocará a todos los métodos cuyos nombres comiencen con

#test. El resto del nombre, por convención, nos describe que es lo que el método prueba.

La más simple definición que se me ocurre es algo que comience y termine inmediatamente, algo como:



Traducimos ese gráfico a código Smalltalk. Decidimos que vamos a definir las transiciones y estas harán referencia a los estados.

```
testLaMasSimpleDefinición
    "Prueba la definición más simple posible"
    | definiciónDeProceso |

    definiciónDeProceso := DefiniciónDeProceso new.
    definiciónDeProceso
            agregarDefiniciónDeTransición: 'comenzar y terminar'
            desde: 'principio'
            hasta: 'fin'.

    self should:[definiciónDeProceso definicionesDeTransiciones size = 1].
    self should:[definiciónDeProceso definicionesDeEstados size = 2].
```

Consejo: Pensar primero en la interfaz pública

Cuando escribimos un test es conveniente evitar pensar como vamos a implementar las clases y sus métodos. Sólo nos concentramos en como nos gustaría usar esos objetos.

Pensar como se usan los objetos (su interfaz pública) antes de pensar como se resolverán los mensajes (los métodos) nos ayudará a crear interfaces más limpias.

Estructura de los test

Prácticamente todos los tests tienen la siguiente forma:

– Instanciación de los objetos.

- Alguna operación sobre los objetos.
- Especificación del resultado esperado.

Para especificar cual es el resultado esperado disponemos de varios métodos (heredados de `TestCase`). Los más importantes son:

#assert: Especifica que el argumento debe ser `true`.

#deny: Especifica que el argumento debe ser `false`.

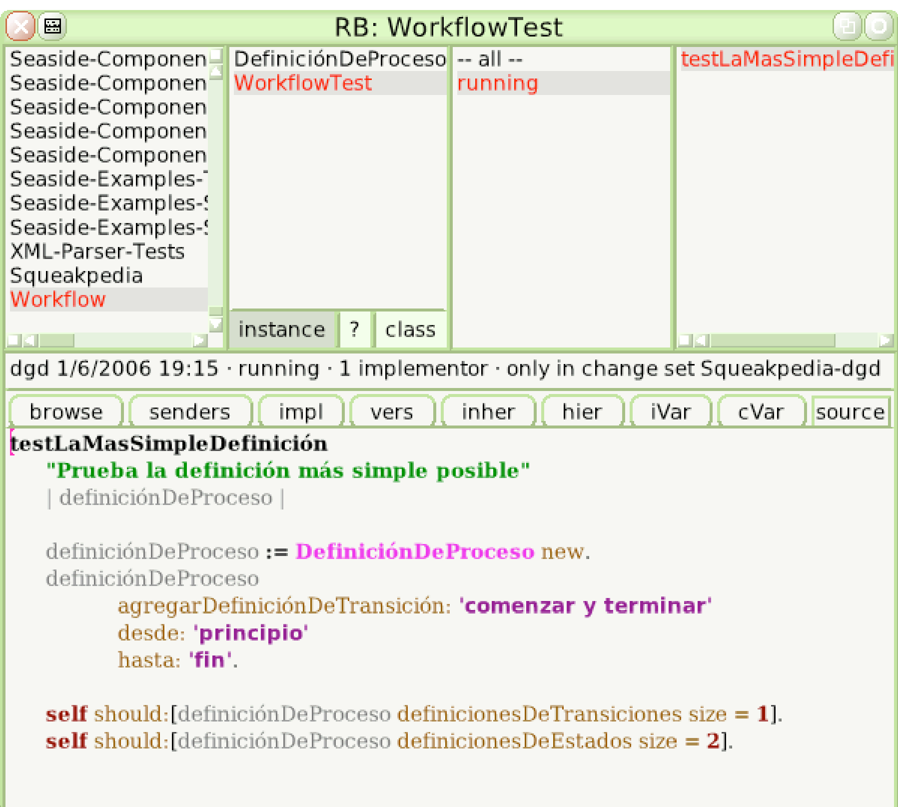
#should: Especifica que el argumento (normalmente un bloque) debe evaluarse a `true`.

#shouldnt: Especifica que el argumento (normalmente un bloque) debe evaluarse a `false`.

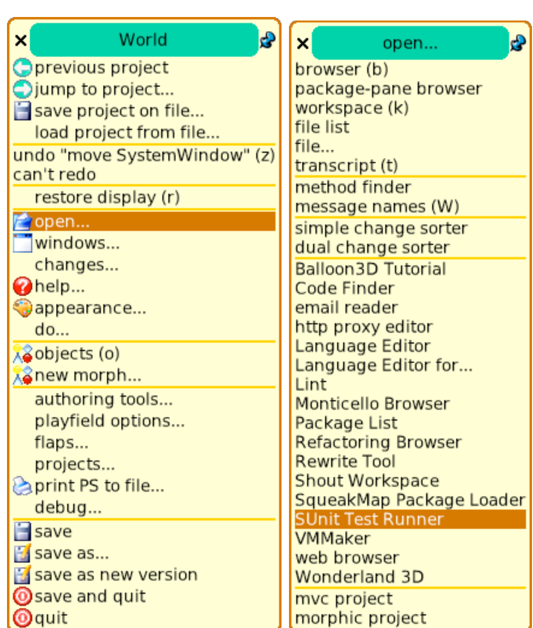
#should:raise: Especifica que la evaluación del primer argumento (normalmente un bloque) debe lanzar una excepción de la clase dada (normalmente alguna subclase de `Exception`) en el segundo argumento.

#shouldnt:raise: Especifica que la evaluación del primer argumento (normalmente un bloque) no debería lanzar una excepción de la clase dada (normalmente alguna subclase de `Exception`) en el segundo argumento.

Al aceptar el método el ambiente nos informa que no sabe que es `DefiniciónDeProceso`. Elegimos crear la clase, herencia de `Object`, en la categoría `'Workflow'`. Luego nos dice que `#agregarDefiniciónDeTransición:desde:hasta:` es un selector desconocido, nosotros confirmamos que está bien escrito. Luego hacemos lo mismo con los selectores `#definicionesDeTransiciones` y `#definicionesDeEstado`. El Browser nos queda así:

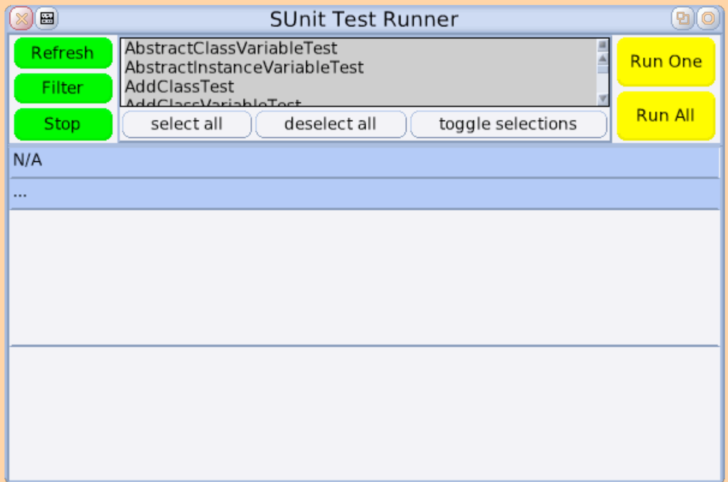


Ahora abrimos la herramienta llamada “SUnit Test Runner” (Menú del Mundo >> `'open...'` >> `'SUnit Test Runner'`) que es la encargada de ejecutar nuestros tests.



SUnit Test Runner

Esta herramienta nos permite ejecutar, automáticamente, todos los tests de las clases seleccionadas. Nos muestra todas las clases que son herencias de `TestCase` y nos permite seleccionar las que queramos.



Los botones tienen las siguiente funciones:

`Refresh:` Vuelve a llenar la lista de clases. Útil cuando creamos nuevas clases de test o borramos alguna ya existente.

`Filter:` Nos permite seleccionar clases usando caracteres comodines.

`Stop:` Detiene la ejecución de los tests.

`Run One:` Ejecuta los test de la clase seleccionada.

`Run All:` Ejecuta los tests de todas las clases seleccionadas.

`select all:` Selecciona todas las clases de test.

`deselect all:` Deselecciona todas las clases de test.

`togle selections:` Invierte la selección. Las clases seleccionadas pasan a estar deseleccionadas y viceversa.

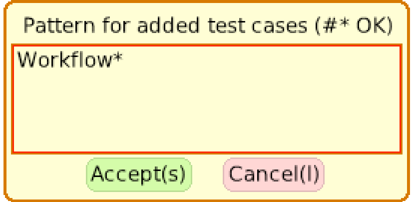
Los dos paneles inferiores muestran la lista de test que fallaron. Hay 2 tipos de fallos en los tests:

– La condición que exigimos en el test no se cumple. A estos tests se les asocia el color amarillo y se muestran en el panel central.

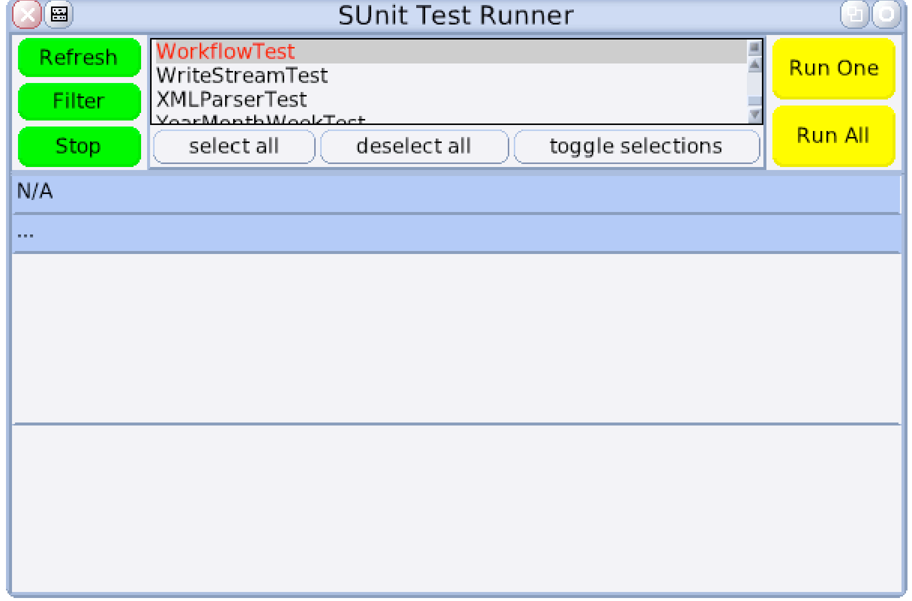
– El test cancela y no termina de ejecutar. Se los asocia con el color rojo y se los muestra en el panel inferior.

Haciendo clic sobre el test con el fallo se abrirá el pre-depurador con el error correspondiente. Podemos utilizar el depurador para corregir el fallo.

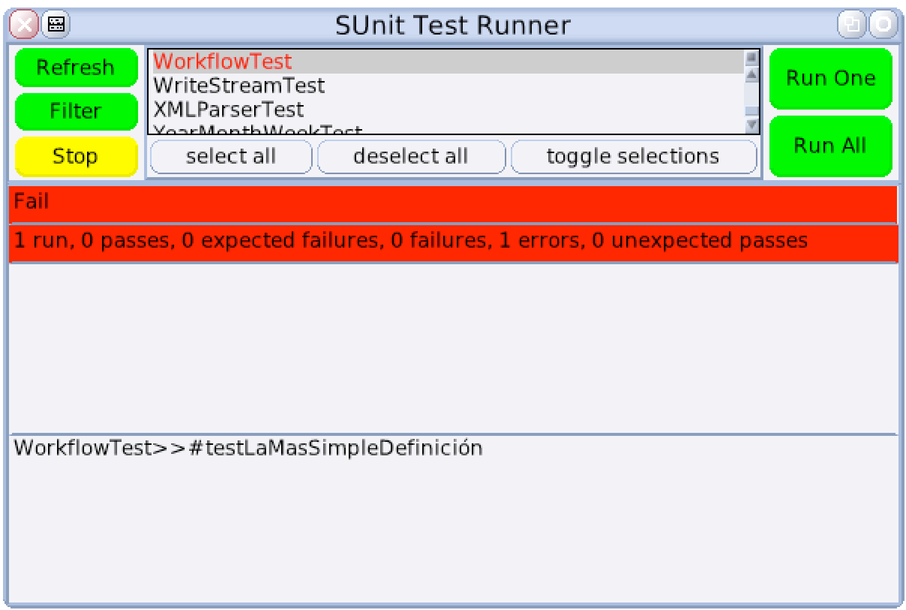
El SUnit Test Runner, al abrirse, tiene seleccionadas todas las clases así que presionamos el botón `deselect all` para deseleccionar todas las clases, luego presionamos el botón `Filter` e ingresamos, en el cuadro de dialogo, lo siguiente: `Workflow*`



Al aceptar nuestra clase de test recién creada queda seleccionada:



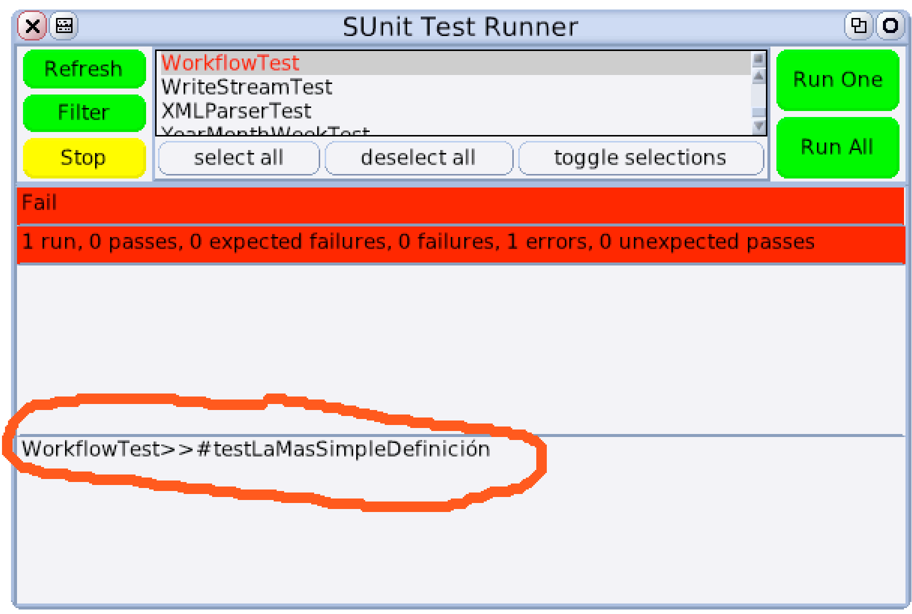
Y estamos listos para ejecutar, por primera vez, nuestro test presionando el botón `Run One`:



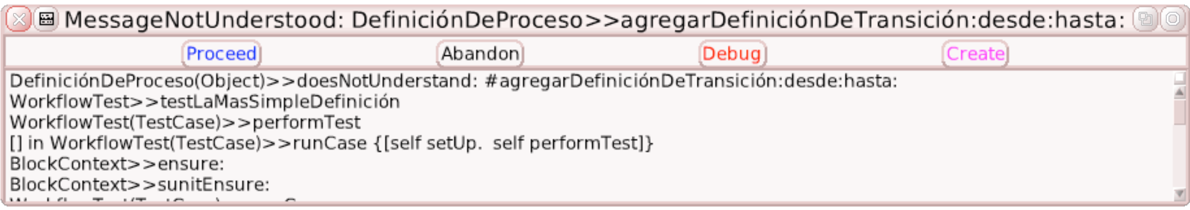
¡Eureka! Ya tenemos el rojo que estábamos buscando (¿Será esto una declaración política?).

Según el TDD, ahora debemos escaparnos del rojo lo más rápido posible. Incluso estamos autorizamos a cometer cuantos pecados necesitemos con tal de lograrlo cuanto antes. (¿Estamos autorizados a pecar para escaparnos del rojo?, mejor me olvido de seguir haciendo un paralelismo con la política antes de que el tema se me escape de las manos).

Ahora hacemos clic sobre el test en rojo (en el panel inferior)



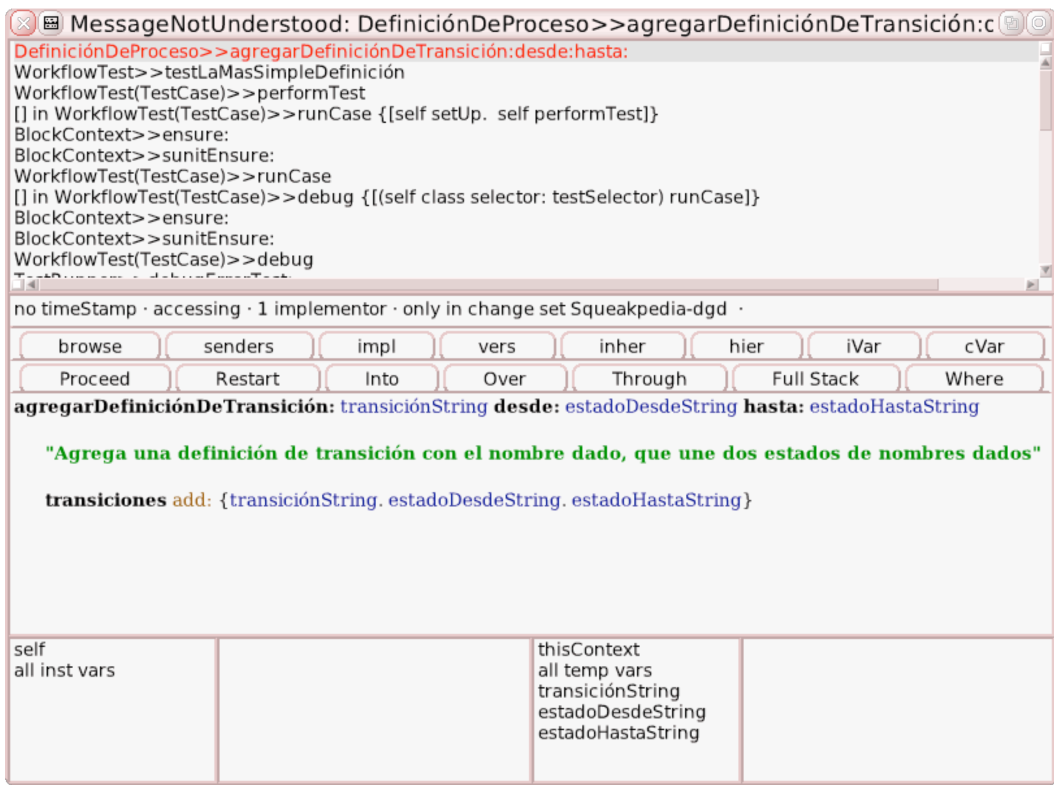
y obtenemos el pre-depurador



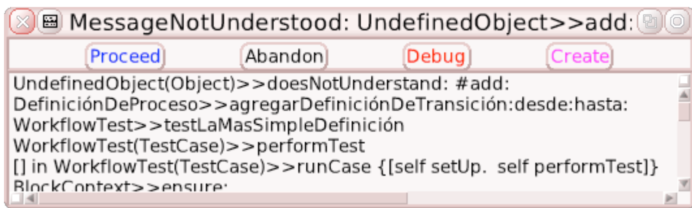
Estamos ante una situación conocida: Un objeto no entiende un determinado mensaje y el pre-depurador nos ofrece crear el método. Lo hacemos en la clase `DefiniciónDeProceso`, en la categoría `'accessing'`.

```
agregarDefiniciónDeTransición: transiciónString desde: estadoDesdeString
hasta: estadoHastaString

   "Agrega una definición de transición con el nombre dado, que une dos
estados de nombres dados"

   transiciones add: {transiciónString. estadoDesdeString. estadoHastaString}
```

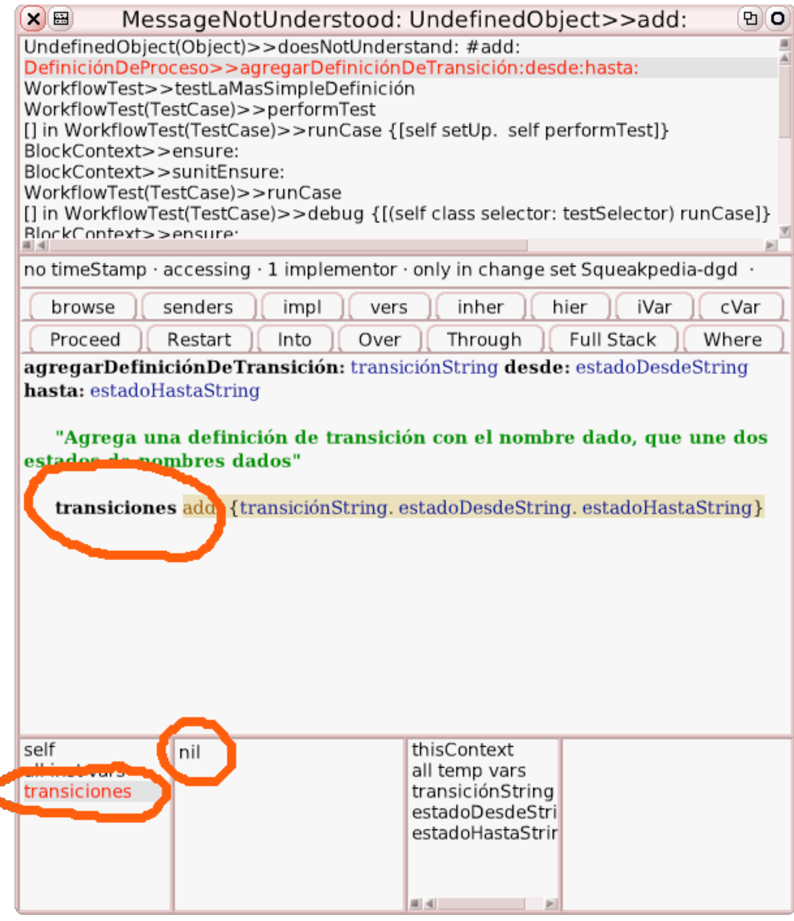
Al aceptar el ambiente nos ofrece la posibilidad de crear la variable de instancia `transiciones`, nosotros aceptamos la propuesta y el depurador nos queda así:



Elegimos continuar presionando el botón `Proceed`. y obtenemos lo siguiente:



Presionamos el botón `Debug` para analizar un poco que es lo que está pasando. Seleccionamos, en el panel superior, el método `DefiniciónDeProceso>>agregarDefiniciónDeTransición:desde:hasta:`. Vemos que el mensaje se está enviando a la recién creada variable de instancia `transiciones`. Seleccionamos la variable de instancia en el panel inferior izquierdo y vemos, en el panel siguiente, que tiene un `nil`.

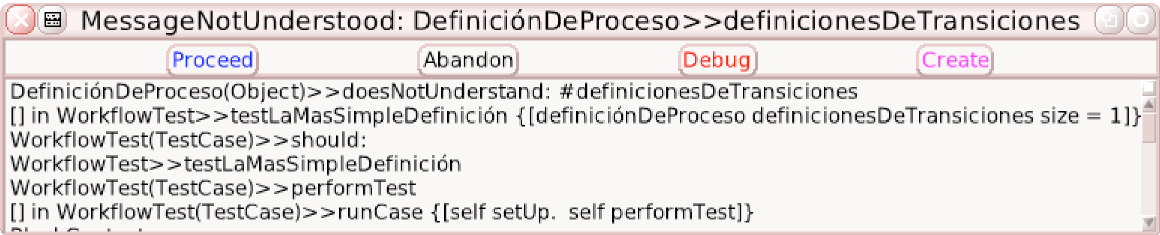


Deberíamos tener, en esa variable, algún tipo de colección como un `Set`. Este es un caso ya conocido: Podemos implementar el método `#initialize` para inicializar la variable de instancia

`transiciones` o podemos usar el idiom lazy-initialization. Preferimos, en este punto, la opción del lazy-initialization ya que estamos apurados para escaparnos del rojo y la podemos implementar directamente en el depurador:

```
agregarDefiniciónDeTransición: transiciónString desde: estadoDesdeString
hasta: estadoHastaString

    "Agrega una definición de transición con el nombre dado, que une dos
estados de nombres dados"

    transiciones isNil
        ifTrue:[transiciones := Set new].

    transiciones add: {transiciónString. estadoDesdeString. estadoHastaString}
```

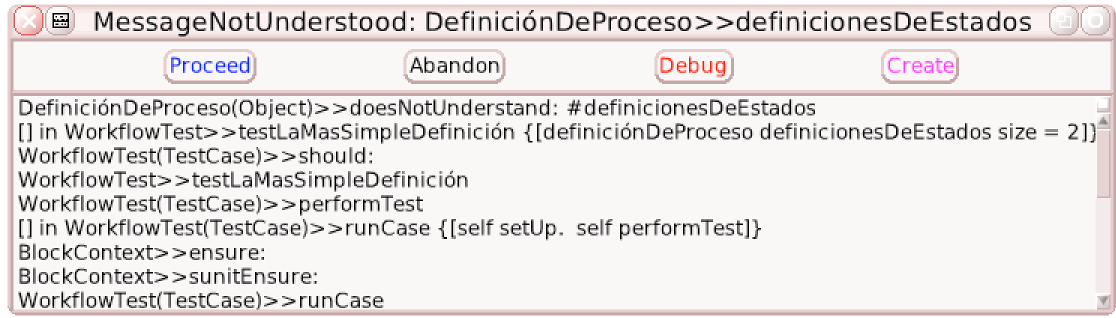
Aceptamos y continuamos presionando el botón `Proceed`.



Ahora implementamos el método en la clase `DefiniciónDeProceso`, en la categoría `'accessing'`, de la siguiente forma:

```
definicionesDeTransiciones
    "Responde las definiciones de transiciones del receptor"
    ^ transiciones
```

Aceptamos y continuamos.



Ahora implementamos el método en la clase `DefiniciónDeProceso`, en la categoría `'accessing'`, de la siguiente forma:

```
definicionesDeEstados
    "Responde las definiciones de estados del receptor"

    | resultado |

    resultado := Set new.
    resultado addAll: (transiciones collect: [:each | each second]).
    resultado addAll: (transiciones collect: [:each | each third]).

    ^ resultado.
```

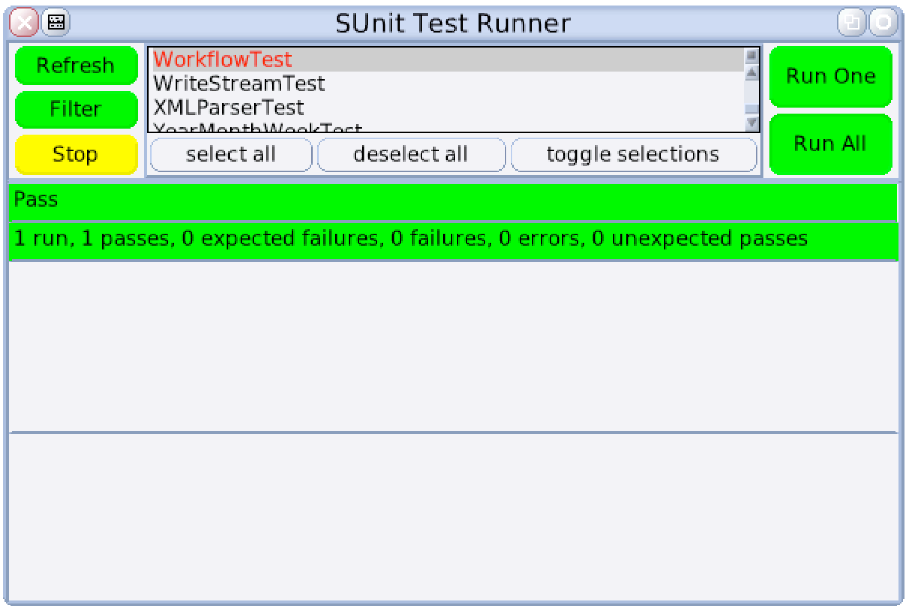
Colecciones – mensaje #collect:

Evalúa el bloque dado por cada elemento del receptor como argumento. Colecciona los resultados en una colección del tipo del receptor. Responde esa colección nueva como resultado.

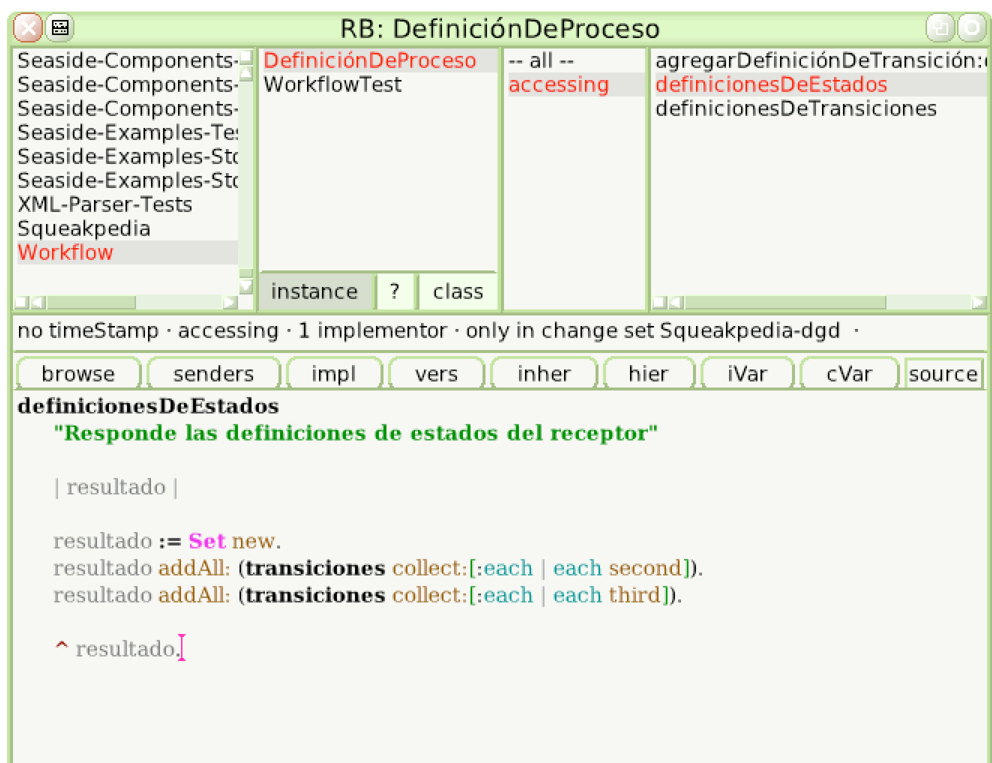
Ejemplos:

```
"la tabla del 2"
(1 to: 10) collect: [:each | each * 2].

"¿par o impar?"
(1 to: 10) collect: [:each | each even].

"todo a mayúsculas"
'una cadena' collect: [:each | each asUppercase].
```

Aceptamos y continuamos. Ahora la ejecución termina bien, podemos volver a ejecutar para obtener el verde (presionando el botón `Run One`).



Según el TDD, ahora debemos refactorizar el código para remover todos los pecados que hallamos cometido en el paso anterior. El método que implementamos no quedó muy claro, así que vamos a limpiarlo un poquito. Buscamos el método `DefiniciónDeProceso>>definicionesDeEstados` en el Browser de Clases:



Lo primero que vamos a hacer es mejorar la legibilidad del método usando el patrón llamado “Explaining Temporary Variable”.

Consejo: Es más barato escribir código limpio

El código se lee y se modifica muchísimas más veces de las que se escribe. Cualquier inversión en generar código limpio y entendible pagará muchos beneficios en el futuro.

Explaining Temporary Variable

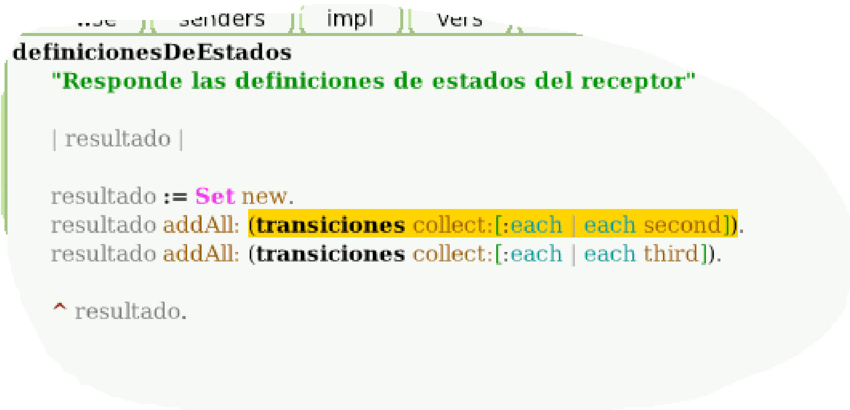
(Variable Temporal Explicativa)

¿Cómo se puede simplificar una expresión compleja dentro de un método?

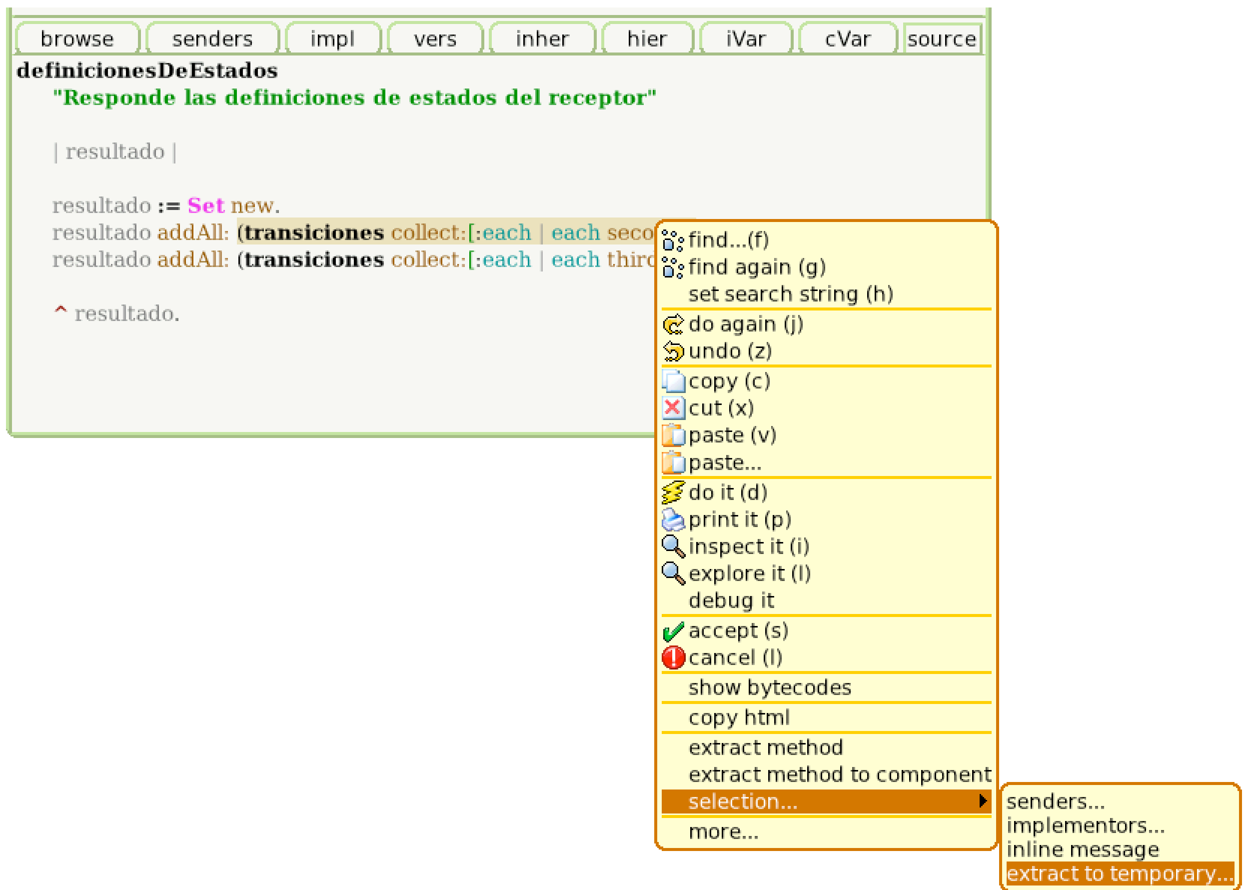
- Extrae una subexpresión desde la expresión compleja, asigna el valor a una variable temporal antes de la expresión compleja. Reemplazar la subexpresión por la variable. El nombre de la variable indicará que significa la subexpresión aumentando la legibilidad del método.

Más información en el libro: “Smalltalk Best Practice Patterns” (ver bibliografía).

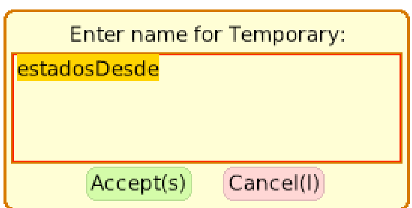
Para hacer esta refactorización utilizaremos una de la funcionalidades del Refactoring Browser. Seleccionamos la subexpresión que queremos extraer.



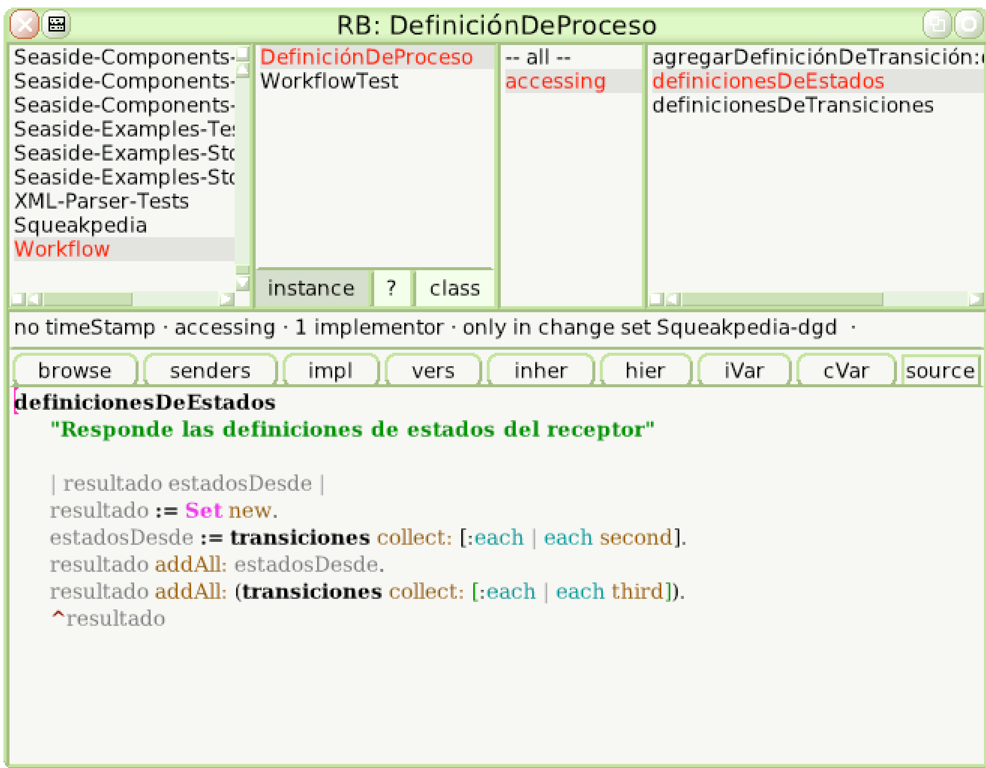
Pedimos el menú contextual y elegimos la opción `“extract to temporary...”` que está dentro del submenú `“selection...”`.



Tecleamos, en el cuadro de diálogo, el nombre de la variable temporal: `estadosDesde`.



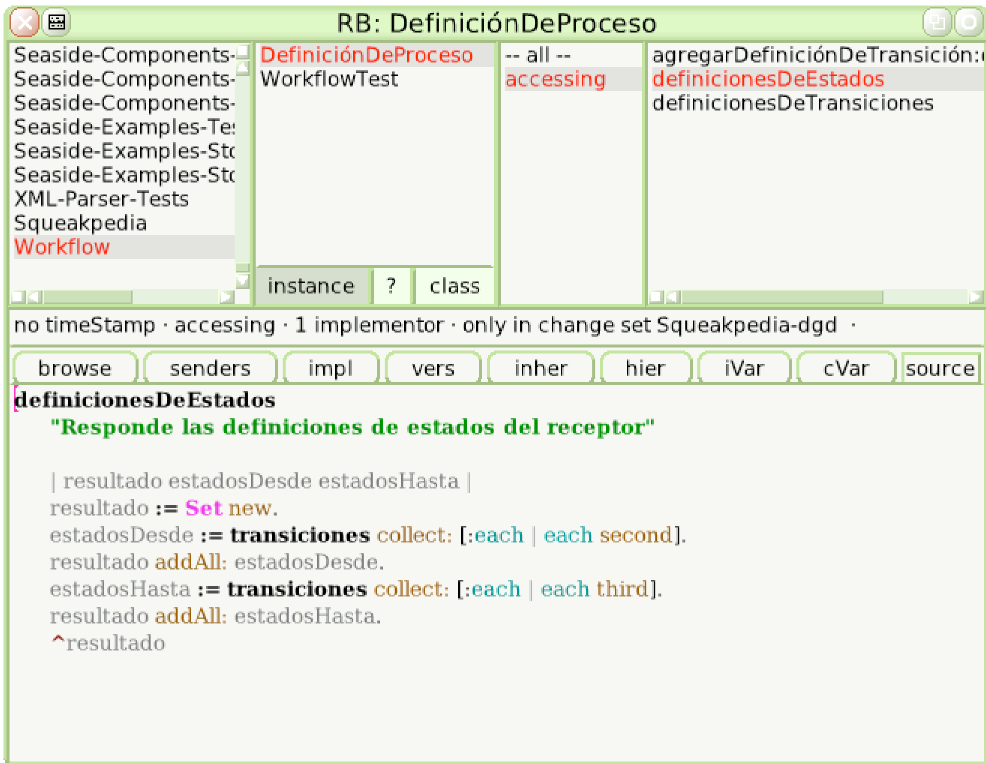
Y aceptamos.



Repetimos la operación para la otra subexpresión.

```
| resultado estadosDesde |
resultado := Set new.
estadosDesde := transiciones collect: [:each | each second].
resultado addAll: estadosDesde.
resultado addAll: (transiciones collect: [:each | each third]).
^resultado
```

Usando `estadosHasta` como nombre de variable temporal.



Ahora reescribimos el método de la siguiente forma:

```
definicionesDeEstados
    "Responde las definiciones de estados del receptor"

    | estadosDesde estadosHasta |
    estadosDesde := transiciones collect: [:each | each second].
    estadosHasta := transiciones collect: [:each | each third].
    ^ (estadosDesde , estadosHasta) asSet.
```

Colecciones – mensaje #,

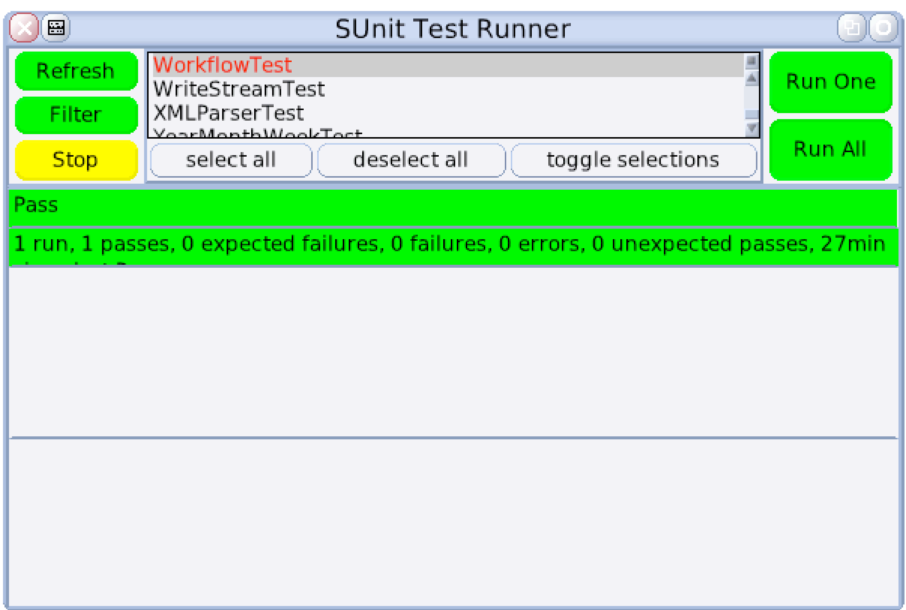
Concatena el receptor y el argumento en una sola colección y responde la nueva colección.

Ejemplos:

```
'un string' , ' ' , 'otro string'.

#(1 2) , #( 3 4).

#(1 2) , 'diego'.

(1 to: 10) , (1 to: 5).
```

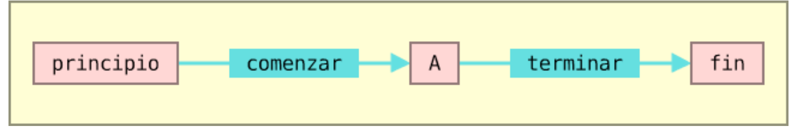
El método, escrito así, deja muy claro que todas las definiciones de estados son la concatenación de los estados desde y los estados hasta, sin duplicados.

Ahora es el momento de ejecutar nuevamente los tests automáticos para asegurarnos que no hemos roto nada con la refactorización.



Hemos cumplido uno de los ciclos del TDD. Escribimos un test antes de implementar la funcionalidad, obtuvimos un rojo, pasamos al verde lo más rápido posible, refactorizamos el código para que quede CLQF (Código Limpio Que Funciona).

Ahora escribamos otro test que instancie una definición de proceso un poco más complicada:

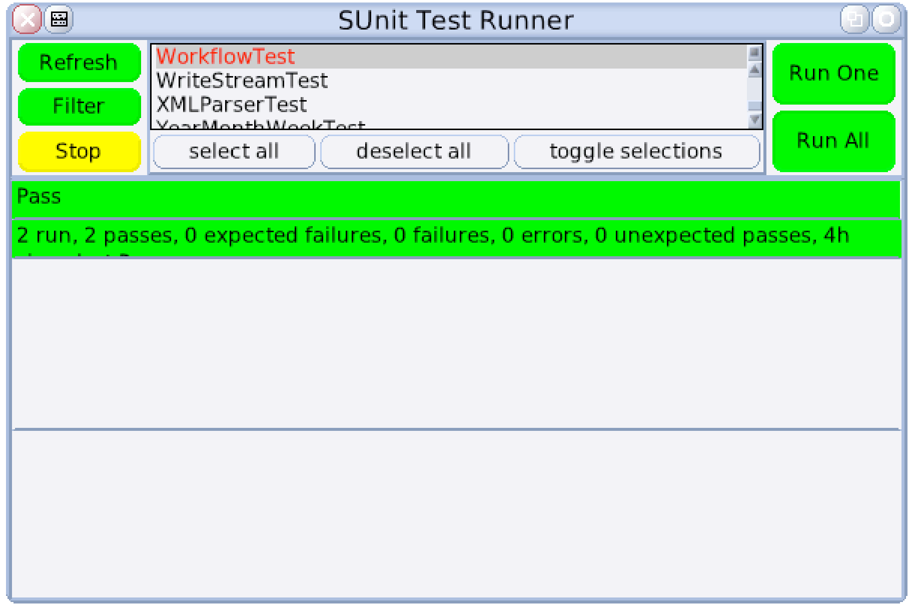


Introducimos otro test, en la misma clase.

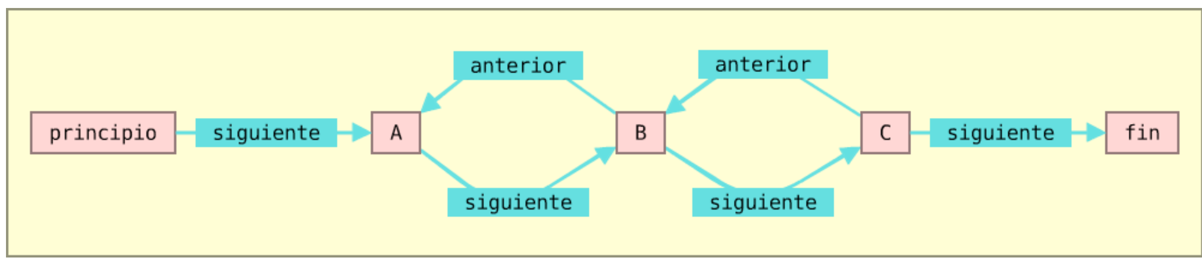
```
testDefiniciónSimple
	"Prueba una definición simple"
	| definiciónDeProceso |

	definiciónDeProceso := DefiniciónDeProceso new.
```

```
definiciónDeProceso
	agregarDefiniciónDeTransición: 'comenzar'
	desde: 'principio'
	hasta: 'A'.
definiciónDeProceso
	agregarDefiniciónDeTransición: 'terminar'
	desde: 'A'
	hasta: 'fin'.

self should:[definiciónDeProceso definicionesDeTransiciones size = 2].
self should:[definiciónDeProceso definicionesDeEstados size = 3].
```

Y ejecutamos nuevamente los tests.



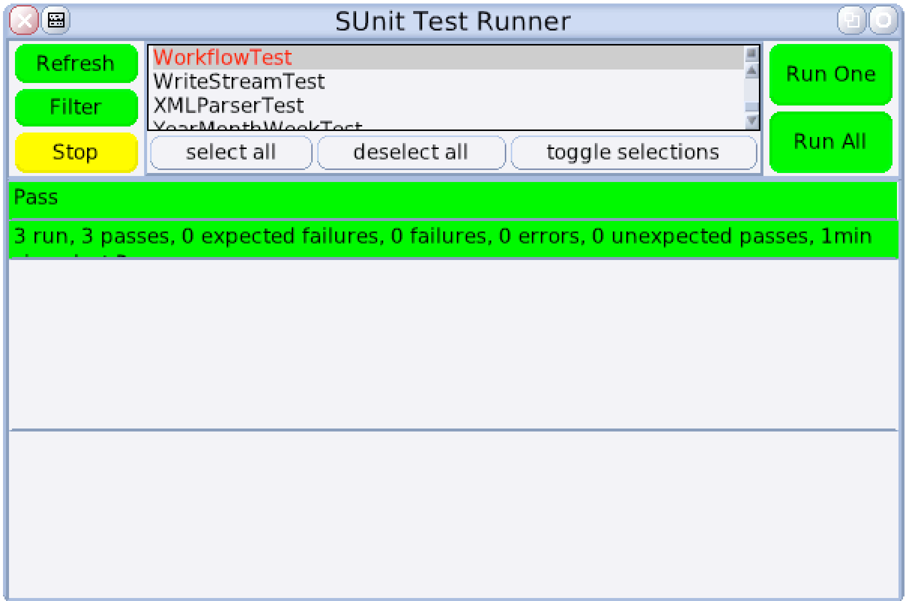
Nada mal. La implementación que tenemos hecha es suficiente para el otro caso. Sigamos introduciendo otro caso, este será aún más complicado que el anterior.



Introducimos otro test.

```
testDefiniciónNoTrivial
	"Prueba una definición no trivial"
	| definiciónDeProceso |

	definiciónDeProceso := DefiniciónDeProceso new.

	definiciónDeProceso
		agregarDefiniciónDeTransición: 'siguiente'
		desde: 'principio'
		hasta: 'A'.

	definiciónDeProceso
		agregarDefiniciónDeTransición: 'siguiente'
		desde: 'A'
		hasta: 'B'.
	definiciónDeProceso
		agregarDefiniciónDeTransición: 'anterior'
		desde: 'B'
		hasta: 'A'.

	definiciónDeProceso
		agregarDefiniciónDeTransición: 'siguiente'
		desde: 'B'
		hasta: 'C'.
	definiciónDeProceso
		agregarDefiniciónDeTransición: 'anterior'
		desde: 'C'
		hasta: 'B'.

	definiciónDeProceso
		agregarDefiniciónDeTransición: 'siguiente'
		desde: 'C'
		hasta: 'fin'.

	self should:[definiciónDeProceso definicionesDeTransiciones size = 6].
	self should:[definiciónDeProceso definicionesDeEstados size = 5].
```

Y volvemos a ejecutar los tests.



Ya tenemos 3 tests en verde con la misma implementación que hicimos para el primero de ellos. Sin embargo ya es hora de pensar de otra forma. Tenemos 3 tests que prueban diferentes instanciaciones válidas pero ¿qué pasa si creamos instancias erróneas?.

Consejo: Probar las situaciones límite

Cuando se hacen tests es importante probar, también, las situaciones de límite.

Ejemplo: Si por alguna causa tenemos que probar que un determinado valor esté entre 5 y 10, la forma “correcta” sería que escribamos, además de los tests par los casos válidos, tests que especifiquen que 4 y 11 son casos inválidos.

En la implementación actual enviamos varios mensajes (uno por transición) para crear una definición completa y no nos es posible validar en cada llamada ya que, el grafo, no sería correcto en los estadios intermedios. La forma más simple que podemos implementar es pedir, explícitamente, a la definición que se valide a si misma y nos indique si es una instancia válida o no.

Modificamos los tests agregando lo siguiente:

```
self should:[definiciónDeProceso esValida].
```

Quedando los tests de la siguiente forma:

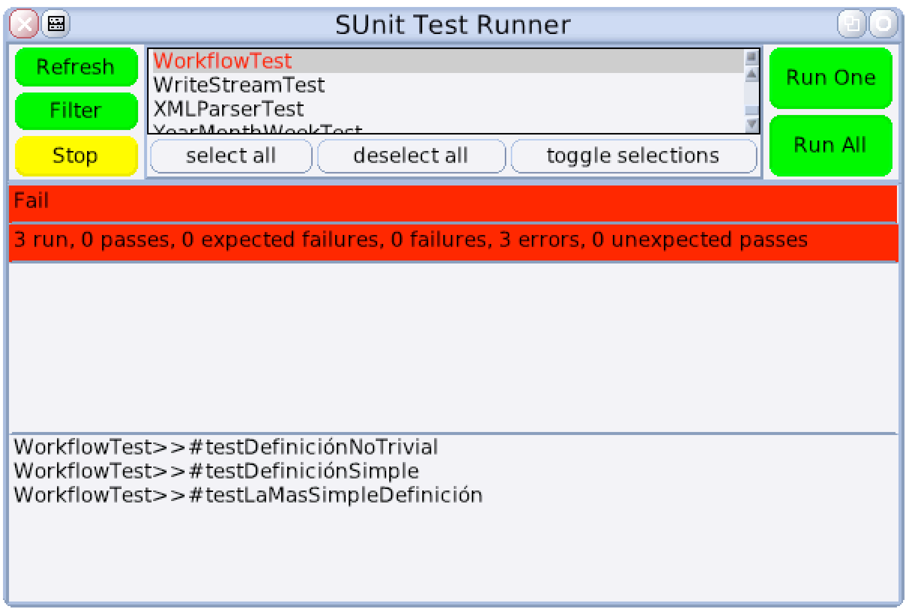
```
testLaMasSimpleDefinición
   "Prueba la definición más simple posible"
   | definiciónDeProceso |

   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
         agregarDefiniciónDeTransición: 'comenzar y terminar'
         desde: 'principio'
         hasta: 'fin'.

   self should:[definiciónDeProceso esValida].
   self should:[definiciónDeProceso definicionesDeTransiciones size = 1].
   self should:[definiciónDeProceso definicionesDeEstados size = 2].
```

```
testDefiniciónSimple
   "Prueba una definición simple"
   | definiciónDeProceso |

   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
         agregarDefiniciónDeTransición: 'comenzar'
         desde: 'principio'
         hasta: 'A'.
   definiciónDeProceso
         agregarDefiniciónDeTransición: 'terminar'
         desde: 'A'
         hasta: 'fin'.

   self should:[definiciónDeProceso esValida].
   self should:[definiciónDeProceso definicionesDeTransiciones size = 2].
   self should:[definiciónDeProceso definicionesDeEstados size = 3].
```

```
testDefiniciónNoTrivial
   "Prueba una definición no trivlal"
   | definiciónDeProceso |

   definiciónDeProceso := DefiniciónDeProceso new.

   definiciónDeProceso
         agregarDefiniciónDeTransición: 'siguiente'
         desde: 'principio'
         hasta: 'A'.

   definiciónDeProceso
         agregarDefiniciónDeTransición: 'siguiente'
```

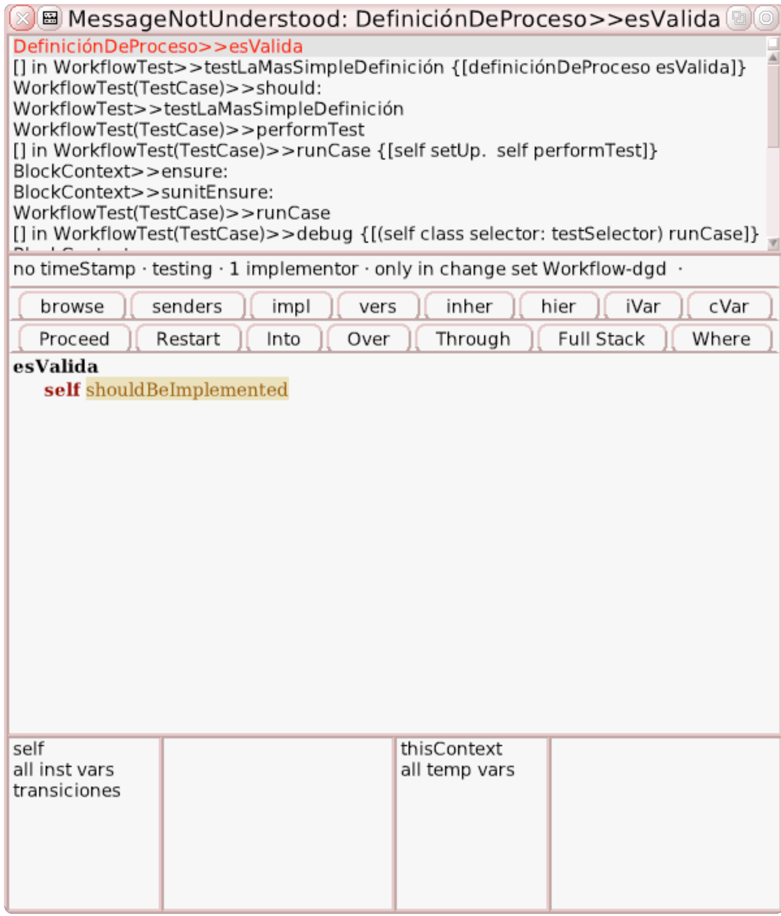
```
        desde: 'A'
        hasta: 'B'.
    definiciónDeProceso
        agregarDefiniciónDeTransición: 'anterior'
        desde: 'B'
        hasta: 'A'.

    definiciónDeProceso
        agregarDefiniciónDeTransición: 'siguiente'
        desde: 'B'
        hasta: 'C'.
    definiciónDeProceso
        agregarDefiniciónDeTransición: 'anterior'
        desde: 'C'
        hasta: 'B'.

    definiciónDeProceso
        agregarDefiniciónDeTransición: 'siguiente'
        desde: 'C'
        hasta: 'fin'.

    self should:[definiciónDeProceso esValida].
    self should:[definiciónDeProceso definicionesDeTransiciones size = 6].
    self should:[definiciónDeProceso definicionesDeEstados size = 5].
```

Ahora ejecutamos los tests.



Previsiblemente obtenemos 3 rojos.

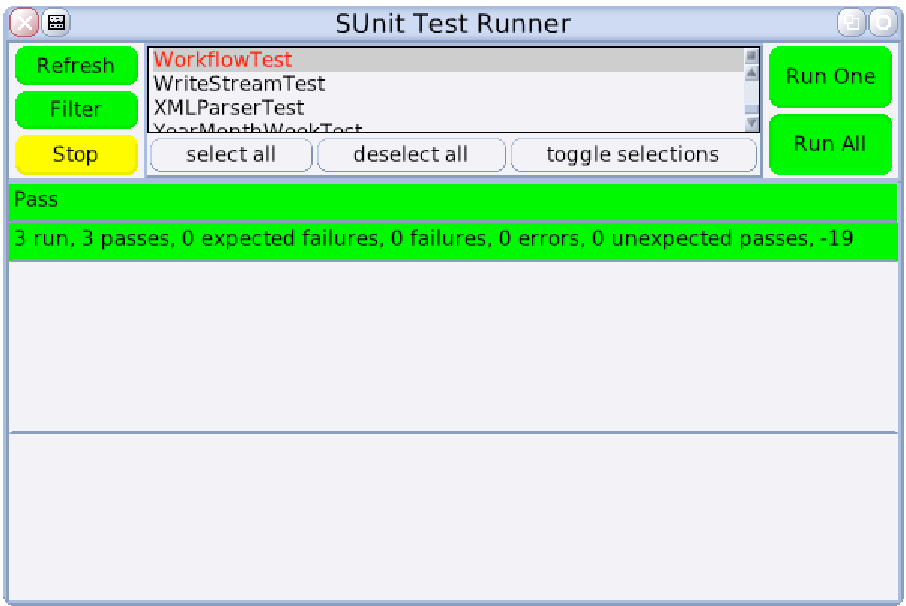
Presionamos sobre cualquier de los tests para obtener el pre-depurador y presionamos el botón `Create` para crear el método faltante en la clase `DefiniciónDeProceso`, en la categoría `'testing'`.



Y lo implementamos, por ahora (recuerden que tenemos que escaparnos del rojo lo más rápido que podamos), de la siguiente forma:

```
esValida
   "Responde si el receptor es una instancia válida"
   ^ true
```

Cerramos el depurador y ejecutamos todos los tests.

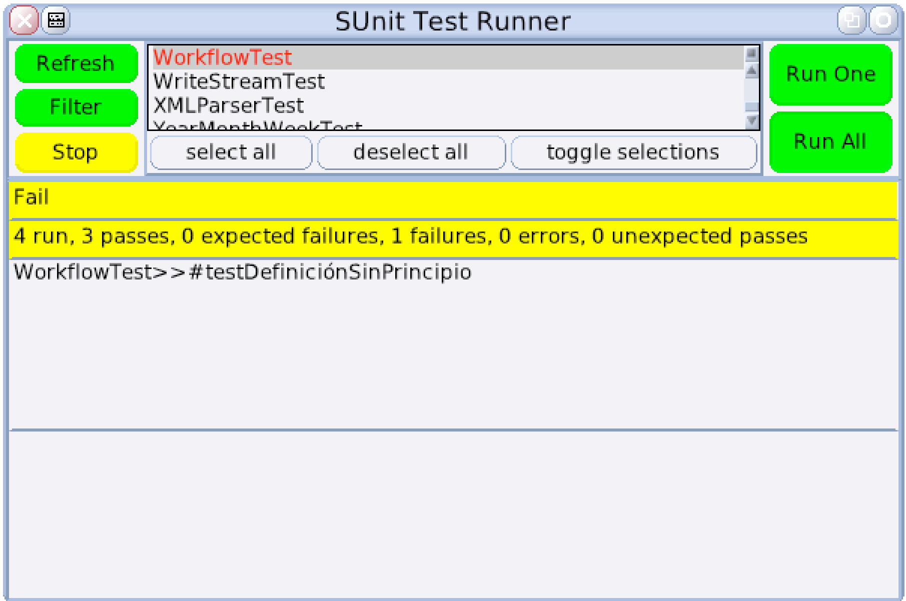


Ahora deberíamos escribir algunos tests que muestren cuales son los casos inválidos.

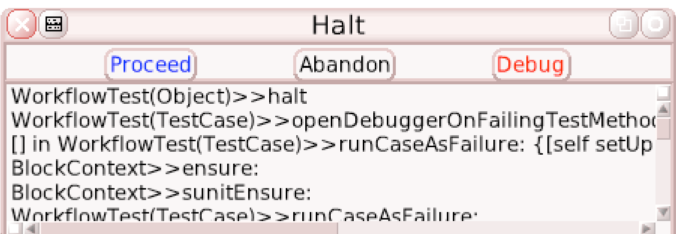
Uno de los casos inválidos sería una definición de proceso que no tenga un estado inicial. Volcamos eso en un test:

```
testDefiniciónSinPrincipio
   "Prueba una definición inválida que no tiene un estado inicial llamado
'principio'"
   | definiciónDeProceso |

   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
         agregarDefiniciónDeTransición: 'comenzar'
         desde: 'A'
         hasta: 'fin'.

   self shouldnt:[definiciónDeProceso esValida].
```

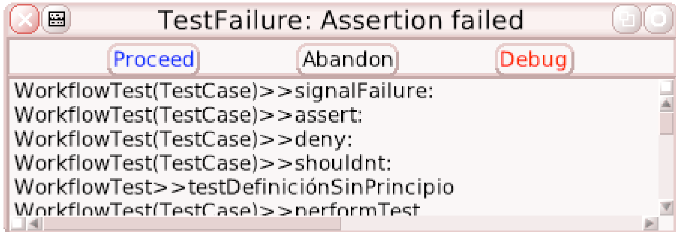
Y ejecutamos los tests.



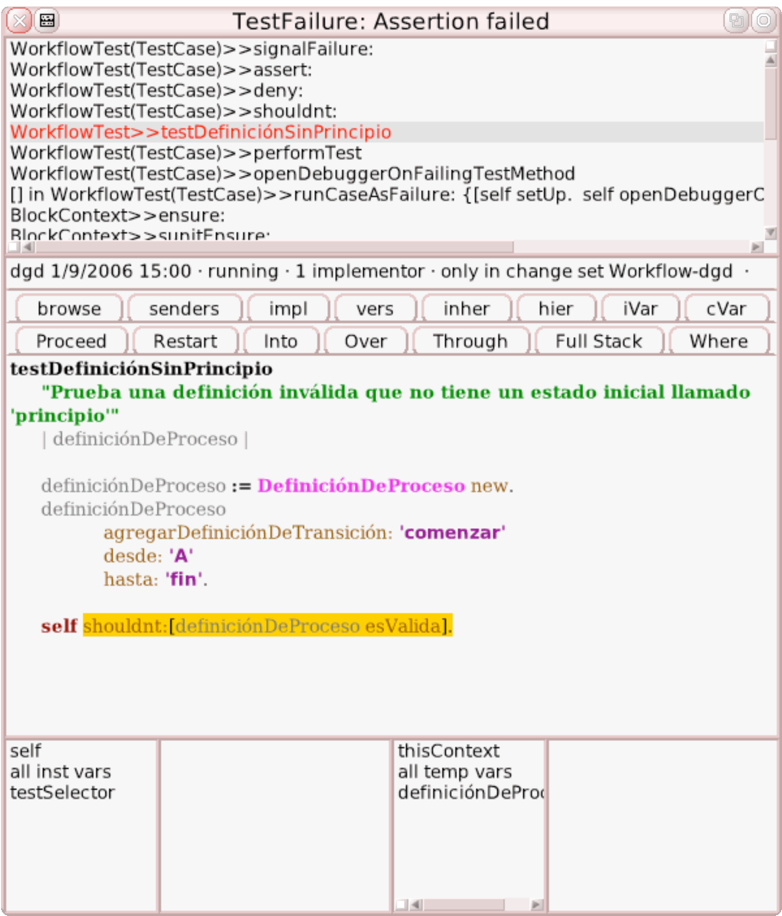
Obtenemos un test amarillo (recuerden que los tests amarillos son aquellos que no causan error, pero que alguna de las condiciones no se cumplen). Hacemos clic sobre el test amarillo y obtenemos el pre-depurador.



Cuando hacemos clic en un test en amarillo, el framework para la ejecución en un punto anterior a nuestro test. Esto se hace para tener la oportunidad de depurar paso a paso el test antes de llegar a las condiciones. En este caso no nos interesa ejecutar el test paso a paso, así que presionamos el botón `Proceed` para seguir y obtener el depurador en el punto de fallo.



Ahora la ejecución se detiene en el punto exacto del fallo y obtenemos el pre-depurador para poder indagar sobre lo que ocurrió. Presionamos el botón `Debug` y buscamos, en el panel superior, el contexto de ejecución correspondiente a nuestro método.



Y vemos claramente que el fallo se da en la condición que no se cumple. Queda en evidencia que tenemos que implementar más funcionalidad en el método `DefiniciónDeProceso>>esValida`.

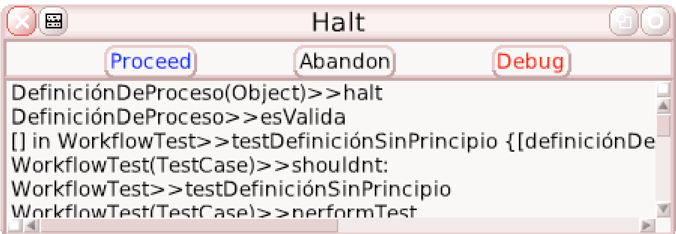
Usando algún Browser modificamos el método `DefiniciónDeProceso>>esValida` de la siguiente forma:

```
esValida
    "Responde si el receptor es una instancia válida"
    self halt.
    ^ true
```

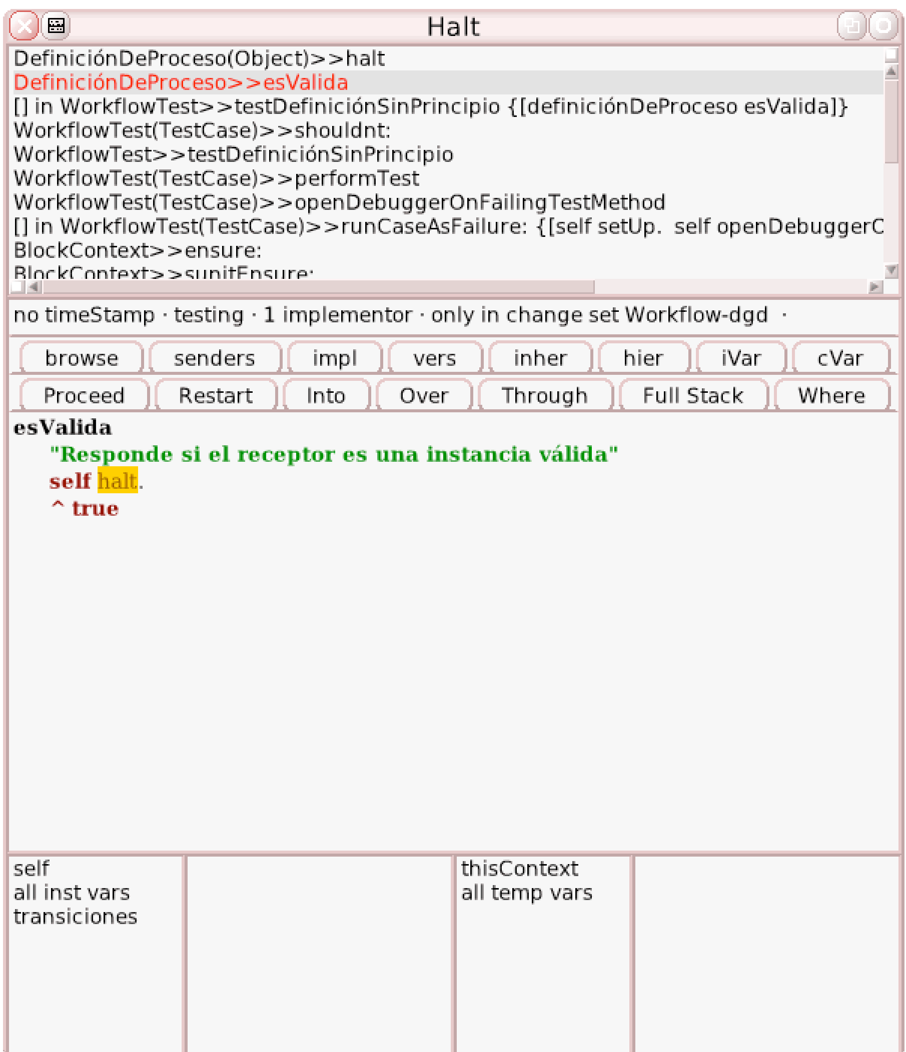
Mensaje #halt

A cualquier objeto se le puede enviar el mensaje `#halt` para hacer "saltar" al depurador. Podemos enviar este mensaje en cualquier punto del código donde necesitemos un breakpoint.

Hacemos clic sobre el test en amarillo, y presionamos el botón `Proceed` en el pre-depurador.



Presionamos el botón `Debug`. y buscamos, en el depurador, el contexto correspondiente al método `DefiniciónDeProceso>>esValida`.



Consejo – El depurador nos brinda más información a la hora de

implementar

Implementar un método en el depurador tiene una ventaja muy importante con respecto a hacerlo en los browser: Disponemos del contexto en el cual se activó el método.

Disponemos de información sobre el receptor (en los 2 paneles de abajo a la izquierda), información del contexto como argumentos y valores de variales temporales (en los 2 paneles de abajo a la derecha), información de la pila de contextos (el call-stack), etc.

Modificar algún método, agregando un `#halt` para que el depurador nos de información de contexto, es una costumbre muy extendida entre los programadores de Smalltalk.

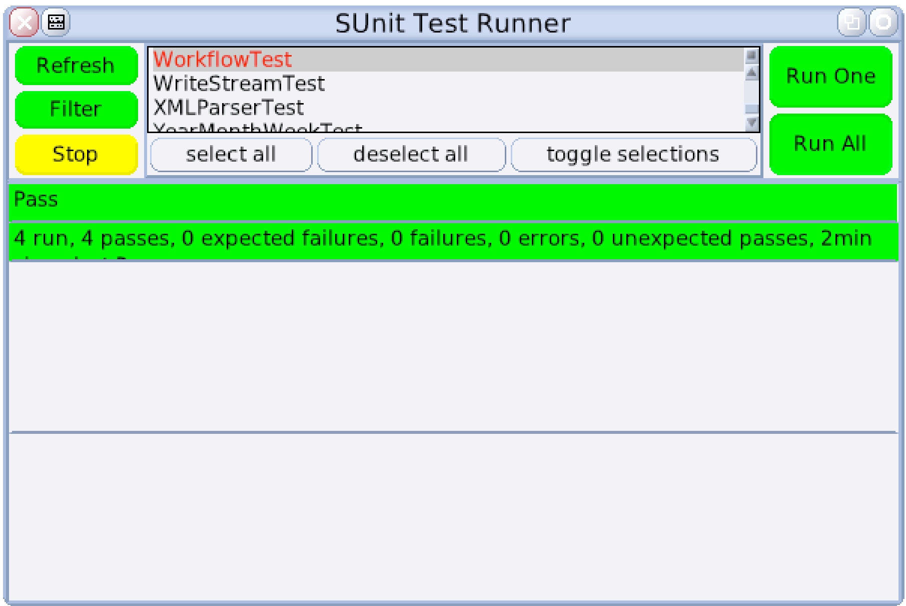
Reimplementamos el método de la siguiente forma:

```
esValida
   "Responde si el receptor es una instancia válida"

   (self definicionesDeEstados includes: 'principio')
      ifFalse:[^ false].

   ^ true
```

Colecciones – mensaje #includes:

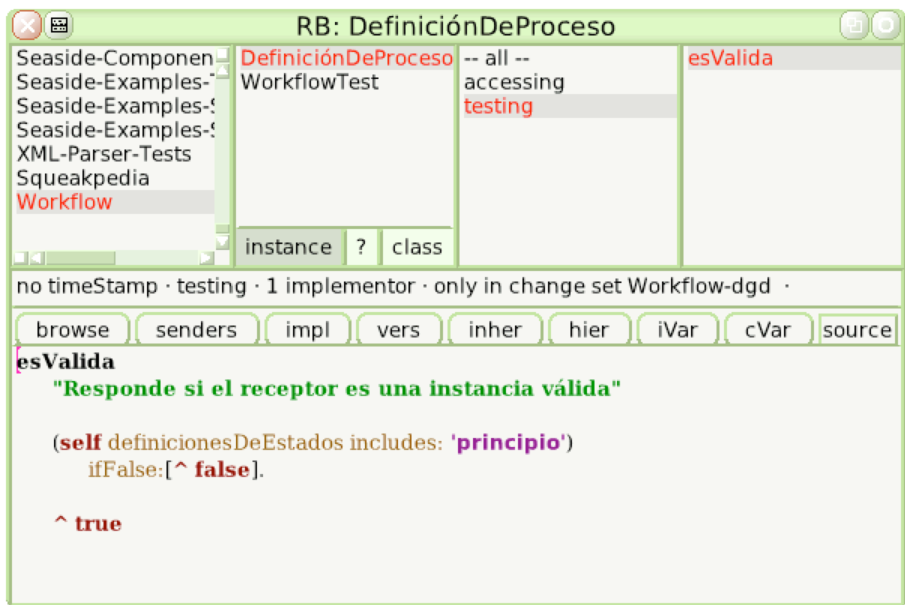
Responde si el objeto dado como argumento es uno de los elementos del receptor.

Y ejecutamos los tests.

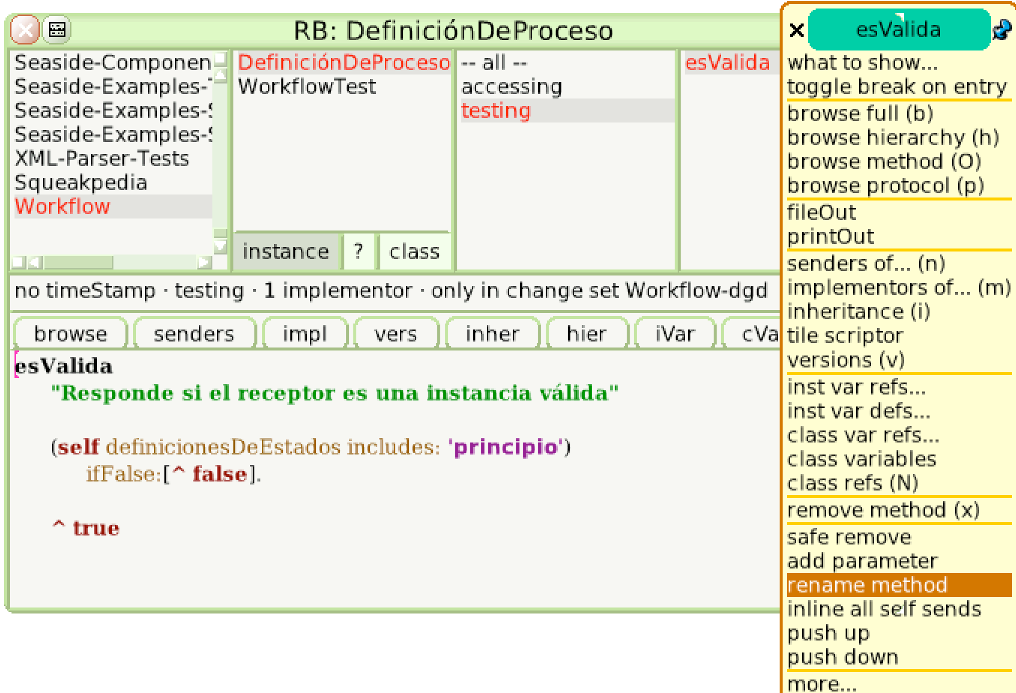


Verde.

Ahora que me doy cuenta, cometimos un error ortográfico en el nombre de método `#esValida`. Nuevamente utilizaremos una de las opciones del Refactoring Browser para corregirlo. Buscamos y seleccionamos el método en cuestión en un Browser de Clases.



Pedimos el menú contextual sobre el método, y seleccionamos la opción `'rename method'`.

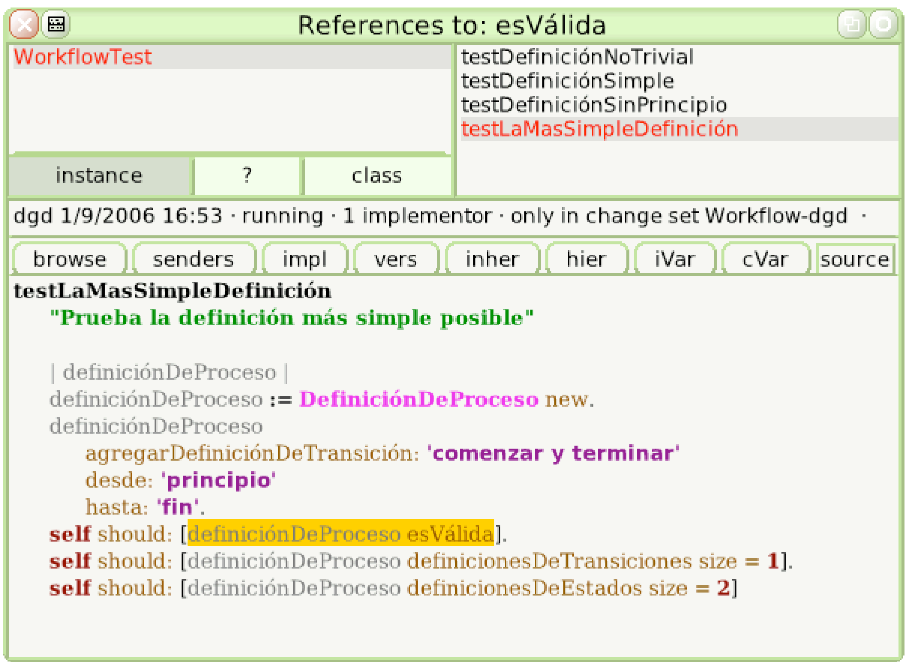


Obtenemos una ventana que nos permite cambiar el nombre de método, como así también cambiar el orden de los argumentos (si los hubiera).

En el primer panel, donde dice `esValida`, tipeamos `esVálida` (esta vez con tilde), aceptamos con ALT-s y presionamos el botón `ok`. En este momento, el Refactoring Browser, cambió el nombre del método y corrigió todos los métodos donde se enviaba el mensaje `#esValida`.

Para validar los cambios que hizo el Refactoring Browser, podemos presionar el botón `senders`

del Browser de Clases, seleccionamos `esVálida`, y vemos que métodos de toda la imagen envían ese mensaje.

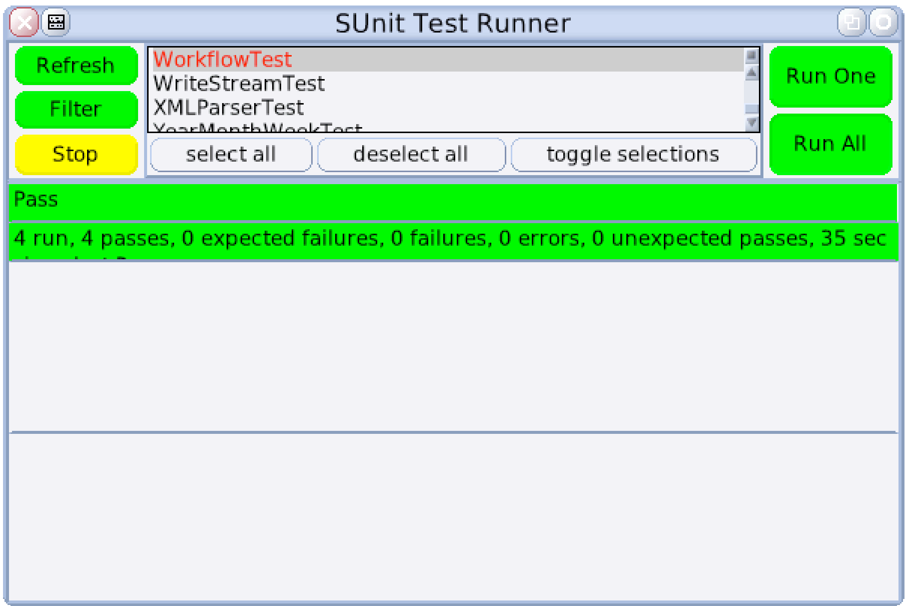


Senders

La función senders (disponible como un botón en los browser, como una opción del menú contextual sobre el método o presionando ALT-n) nos brinda un browser con todos los métodos de la imagen que envíen dicho mensaje.

Esta es una de la principales herramientas para buscar en Smalltalk. Cada caso que el 'senders' nos muestra es un ejemplo de uso del mensaje. Y no es un ejemplo cualquiera, sino que son ejemplos que están funcionando en la imagen y que, llegado el caso, podemos depurar paso a paso. Mucho mejor que una documentación estática en algún archivo externo.

Además deberíamos ejecutar, nuevamente, los tests para asegurarnos que no hemos roto nada.



Ahora que estamos en verde, es el momento para limpiar un poco el código que acabamos de generar.

```
esVálida
   "Responde si el receptor es una instancia válida"

   (self definicionesDeEstados includes: 'principio') ifFalse: [^false].
   ^true
```

Una de las cosas que restan claridad en el código es el uso de constantes 'mágicas'. Una constante mágica es una constante (muchas veces un literal) que aparece en el código sin más, sin especificar su significado. Para evitar este mal vamos a utilizar el patrón 'Constant Method'.

Constant Method

(Método Constante)

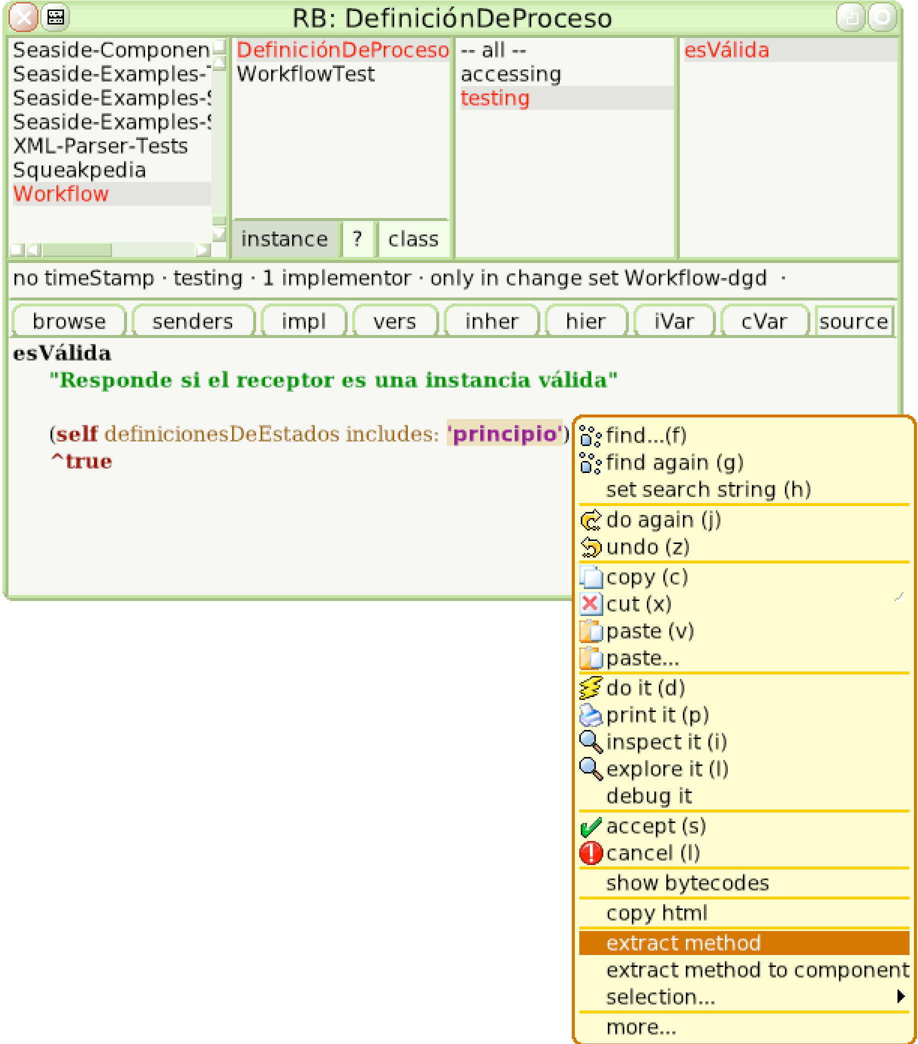
¿Cómo se puede codificar una constante brindando información sobre su significado?

- Crear un método que devuelve la constante. Aprovecharse del nombre del método, y el comentario de este, para explicar el significado de la constante.

Más información en el libro: “Smalltalk Best Practice Patterns” (ver bibliografía).

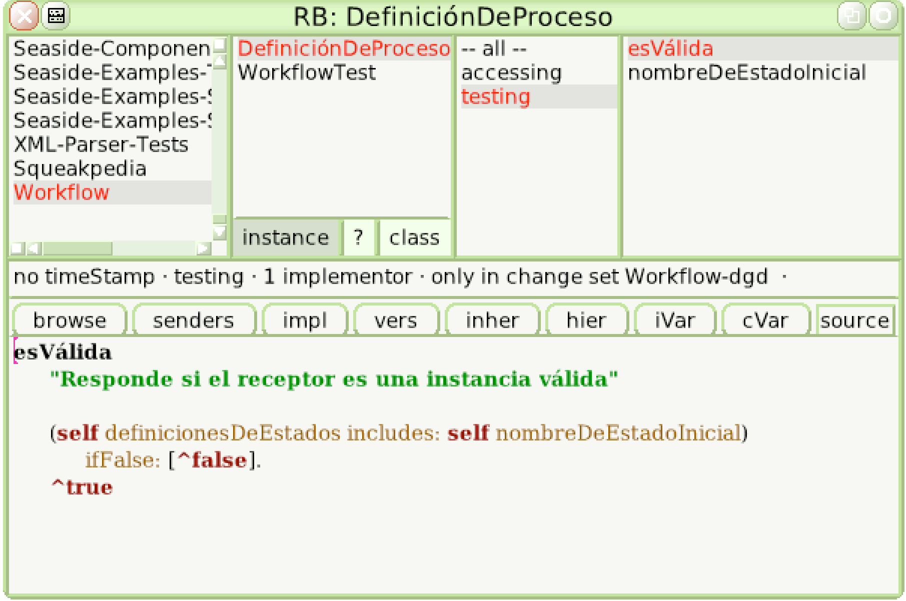
Nos valdremos, nuevamente, de una de las funcionalidades del Refactoring Browser (¡Cómo me gusta el Refactoring Browser!). Seleccionamos la parte que corresponde a la constante, pedimos el

menú contextual y seleccionamos la opción `'extract method'`.

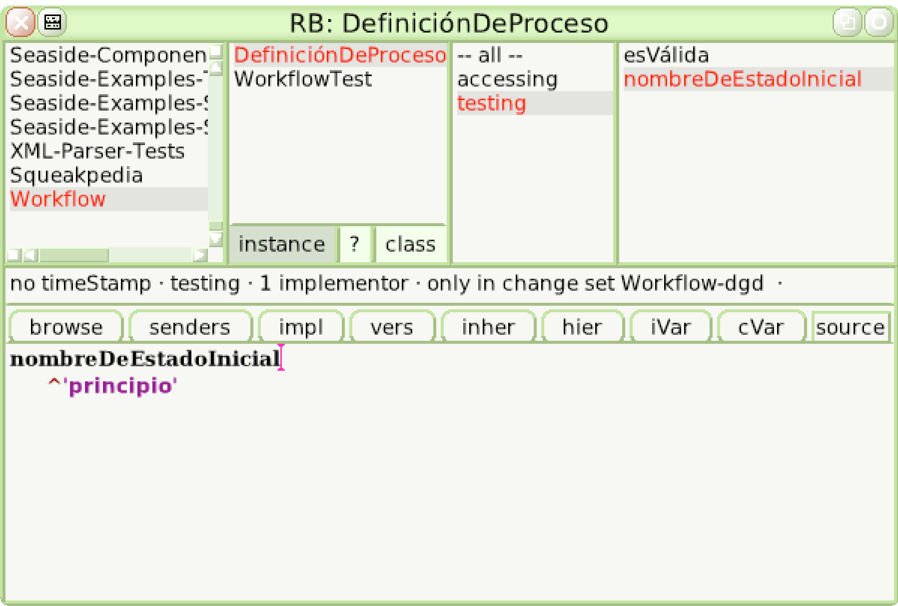


Ingresamos el nombre del método (de la misma forma que lo hicimos cuando cambiamos el nombre al método `#esValida`). Ingresamos: `nombreDeEstadoInicial`.

Aceptamos (con ALT-s) y presionamos el botón `ok`. El método nos queda así:



Y el recién creado método queda así:



Comentamos el método de la siguiente forma:

```
nombreDeEstadoInicial
   "Responde el nombre del estado inicial"

   ^ 'principio'
```

Seguimos introduciendo los casos inválidos, ahora hacemos lo propio para el estado final.

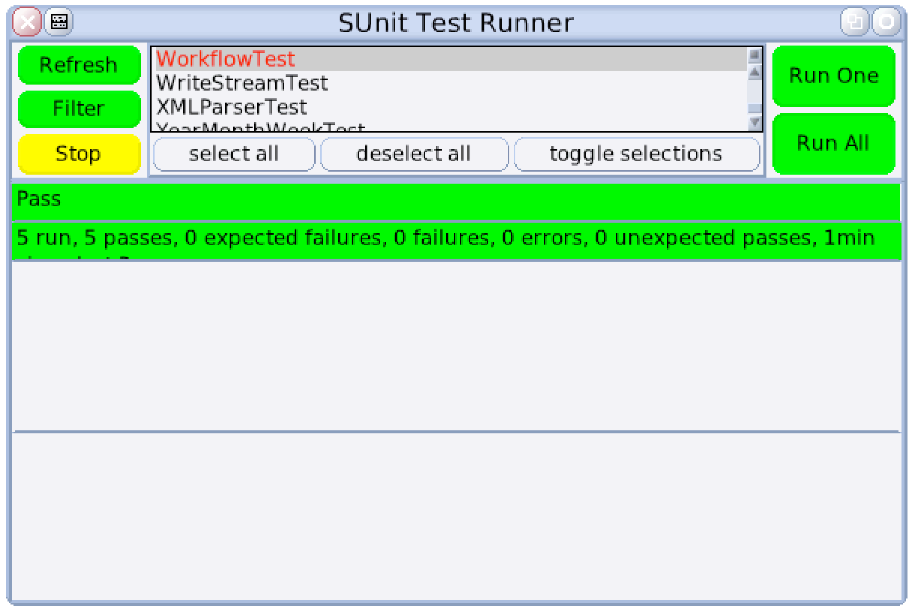
```
testDefiniciónSinFinal
   "Prueba una definición inválida que no tiene un estado final llamado 'fin'"

   | definiciónDeProceso |
   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'comenzar'
      desde: 'principio'
      hasta: 'A'.
   self shouldnt: [definiciónDeProceso esVálida]
```

Obtenemos el amarillo y, a punto seguido, implementamos la funcionalidad de forma análoga a la anterior.

```
esVálida
   "Responde si el receptor es una instancia válida"

   (self definicionesDeEstados includes: self nombreDeEstadoInicial)
      ifFalse: [^false].

   (self definicionesDeEstados includes: self nombreDeEstadoFinal)
      ifFalse: [^false].

   ^true
```

```
nombreDeEstadoFinal
   "Responde el nombre del estado final"

   ^ 'fin'
```

Ejecutamos los tests y estamos en verde nuevamente.

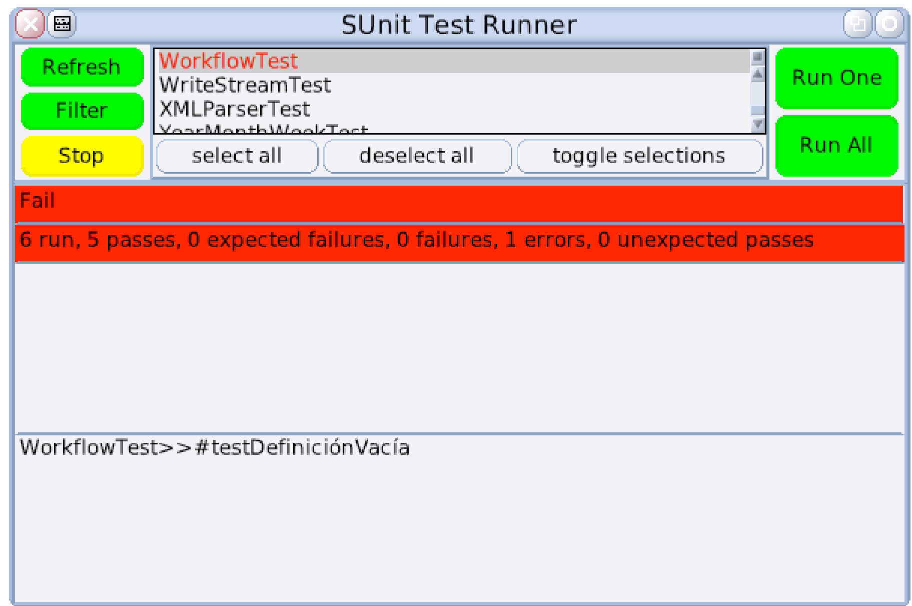


Nos olvidamos de, tal vez, el caso inválido más simple: Una definición sin transiciones ni estados.

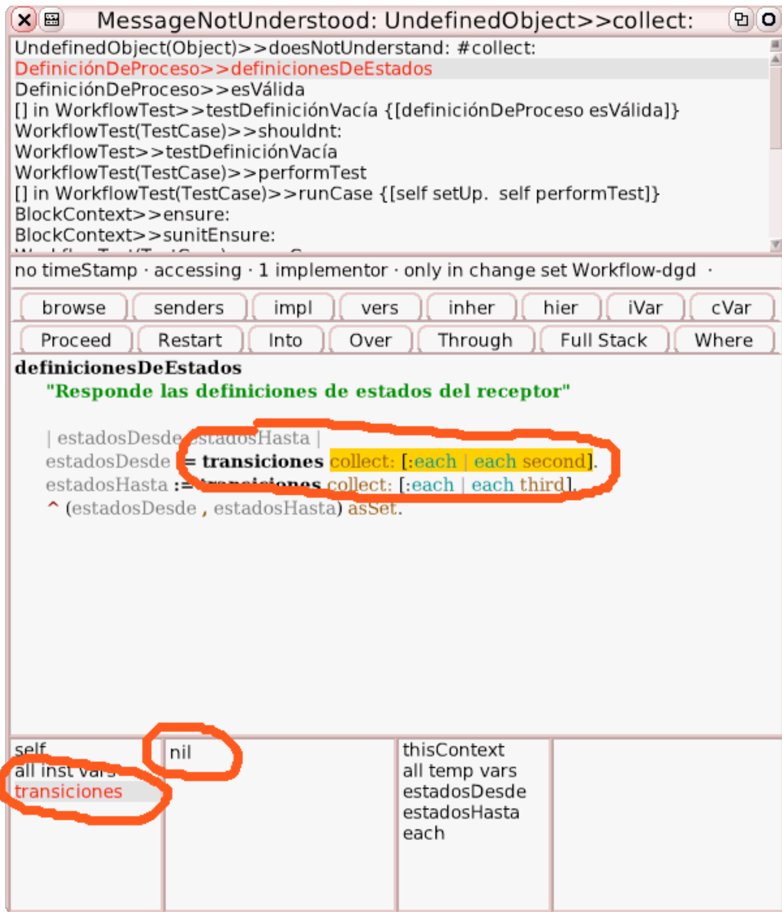
```
testDefiniciónVacía
   "Prueba una definición inválida por estar vacía"

   | definiciónDeProceso |
   definiciónDeProceso := DefiniciónDeProceso new.
```

```
self shouldnt: [definiciónDeProceso esVálida]
```

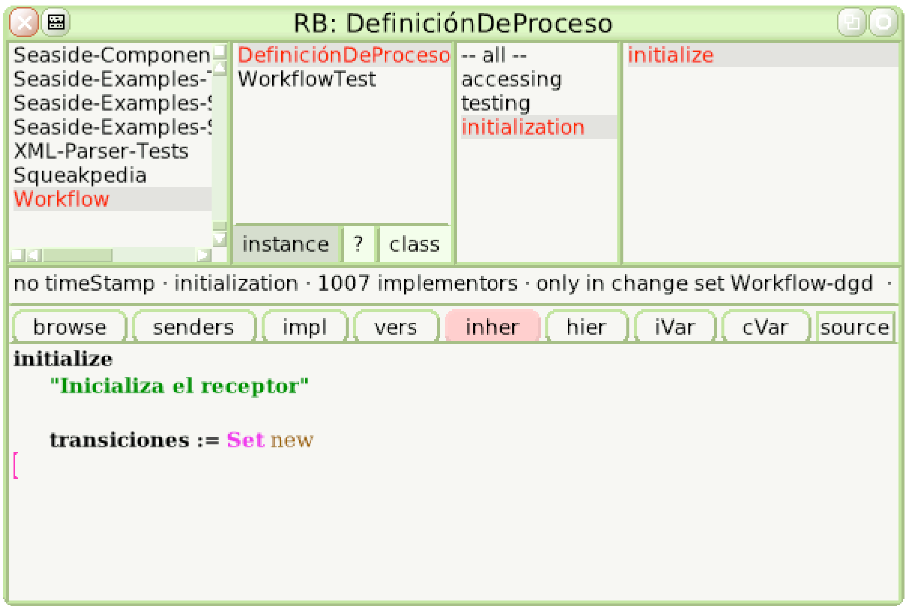
Y ejecutamos los tests.



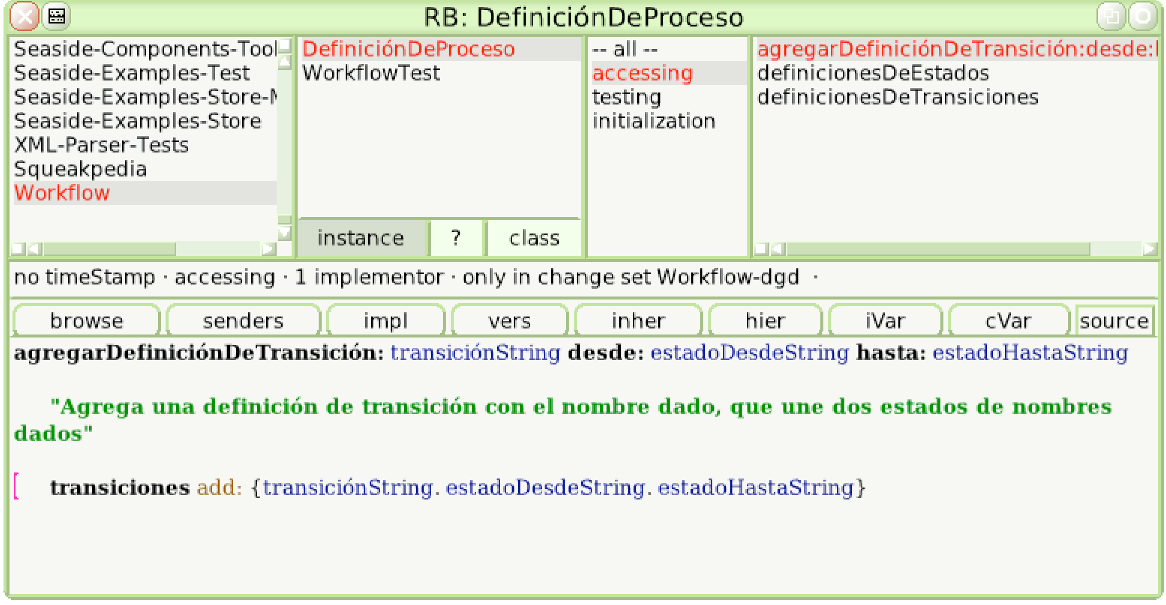
Hacemos clic sobre el test con error y abrimos el depurador para ver que es lo que ocurrió.



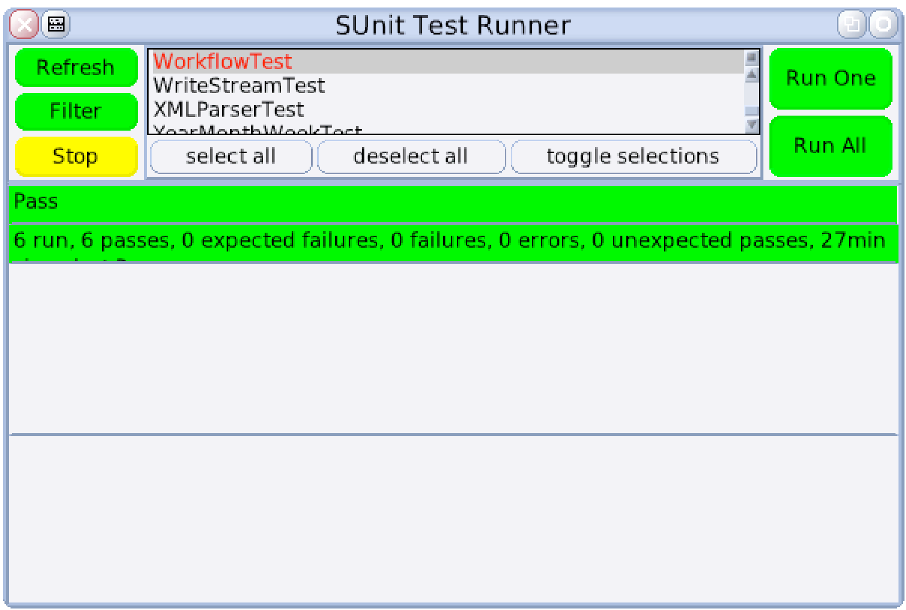
Nuevamente estamos ante el caso que la variable de instancia `transiciones` no está inicializada. El caso anterior lo resolvimos con lazy-initialization en el método `#agregarDefiniciónDeTransición:desde:hasta:`, pero ese método no fue invocado en este caso. Ya es hora que movamos la inicialización de esa variable al método `#initialize` (en la categoría `'initialization'`).



Y removemos el lazy-initialization que hicimos anteriormente.



Ahora ejecutamos, nuevamente, los tests.

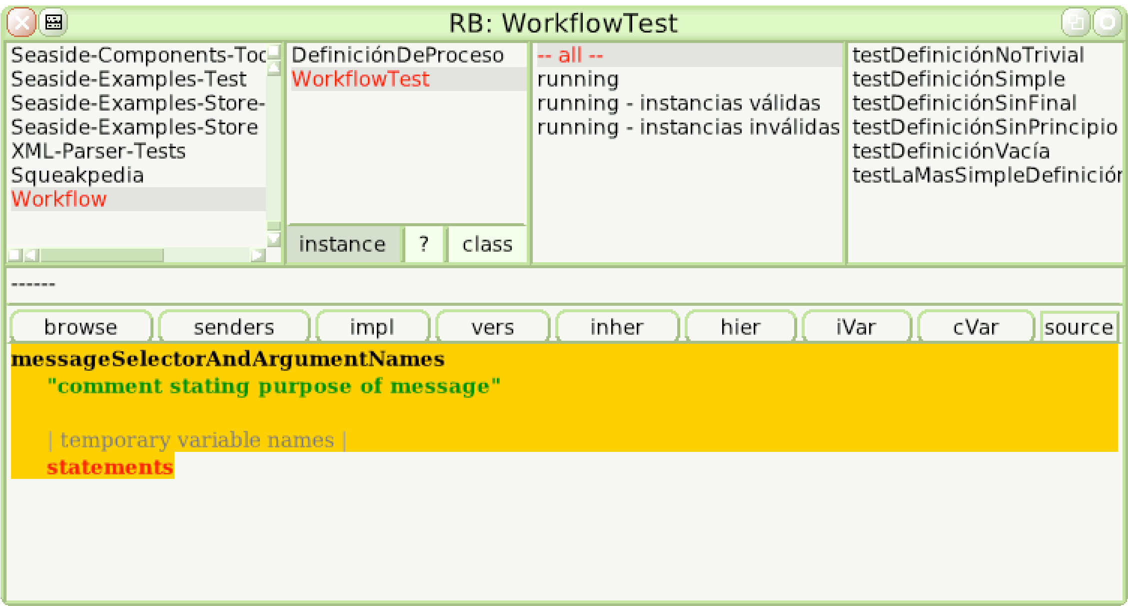


Los tests son, también, documentación

Si respetamos las simples reglas que dominan el TDD, no tendremos ninguna funcionalidad en el software desarrollado que no tenga, como mínimo, 1 test que la valide. Esto implica que el conjunto de los test de un sistema son una excelente documentación.

Los tests componen una documentación que se puede, además de leer, ejecutar en el depurador paso a paso. También sabemos a ciencia exacta si esa documentación está actualizada o no según el test esté en verde o en rojo. Y, por si fuera poco, los tests son documentación que se escribe a la vez que se desarrolla el sistema.

Para mejorar la documentación del sistema que estamos desarrollando, vamos a recategorizar los tests para indicar cuales corresponden a instanciaciones válidas y cuales a instanciaciones inválidas. Usamos el Browser de Clases para crear 2 nuevas categorías de métodos: `'running - instancias válidas'` y `'running - instancias inválidas'` (usamos el menú contextual del panel de categorías de métodos, la opción `'new category...'`).



Ahora arrastramos el método (desde el panel de métodos) hasta la categoría donde queremos que resida. Reorganizamos los métodos y luego usamos opción `'remove empty categories'` desde el menú contextual del panel de categorías para borrar la categoría vacía.

Seguimos con la implementación. Otra condición que tiene que cumplir las definiciones de proceso para ser válidas es que el estado inicial sea realmente el primero. Escribimos otro test con esta condición.

```
testDefiniciónPrincipioInválido
   "Prueba una definición donde el estado inicial no es realmente el inicial"

   | definiciónDeProceso |

   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'comenzar'
      desde: 'A'
      hasta: 'principio'.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'terminar'
      desde: 'principio'
      hasta: 'fin'.

   self shouldnt: [definiciónDeProceso esVálida]
```

Ejecutamos los tests y obtenemos el amarillo esperado. Para saber si el estado inicial es realmente el inicial, debemos asegurarnos que ninguna transición tenga ese estado como destino.

Modificamos el método `#esVálida` de la siguiente forma.

```
esVálida
    "Responde si el receptor es una instancia válida"

    (self definicionesDeEstados includes: self nombreDeEstadoInicial)
        ifFalse: [^false].

    (self definicionesDeEstados includes: self nombreDeEstadoFinal)
        ifFalse: [^false].

    (transiciones anySatisfy: [:each | each third = self nombreDeEstadoInicial])
        ifTrue: [^false].

    ^true
```

Colecciones – mensaje #anySatisfy:

Evalúa el bloque con los elementos del receptor. Si el bloque responde `true` (verdadero) para cualquiera de los elementos, devuelve `true`. Sino devuelve `false` (falso).

Ejecutamos los tests y estamos en verde nuevamente.

De la misma forma, tenemos que asegurarnos que el estado final sea realmente el estado final. Escribimos otro test.

```
testDefiniciónFinalInválido
    "Prueba una definición donde el estado final no es realmente el final"

    | definiciónDeProceso |

    definiciónDeProceso := DefiniciónDeProceso new.
    definiciónDeProceso
        agregarDefiniciónDeTransición: 'comenzar'
        desde: 'principio'
        hasta: 'fin'.
    definiciónDeProceso
        agregarDefiniciónDeTransición: 'terminar'
        desde: 'fin'
        hasta: 'A'.
```

```
self shouldnt: [definiciónDeProceso esVálida]
```

Ejecutamos los tests y obtenemos otra vez el amarillo.

```
esVálida
   "Responde si el receptor es una instancia válida"

   (self definicionesDeEstados includes: self nombreDeEstadoInicial)
      ifFalse: [^false].

   (self definicionesDeEstados includes: self nombreDeEstadoFinal)
      ifFalse: [^false].

   (transiciones anySatisfy: [:each | each third = self nombreDeEstadoInicial])
      ifTrue: [^false].

   (transiciones anySatisfy: [:each | each second = self nombreDeEstadoFinal])
      ifTrue: [^false].

   ^true
```

Y de nuevo estamos en verde, pero la claridad del método `#esVálida` deja mucho que desear. Vamos a limpiarlo un poco.

Las expresiones que determinan los 4 casos inválidos son un poco crípticas. Nos valdremos del patrón 'Composed Method'.

Composed Method

(Método Compuesto)

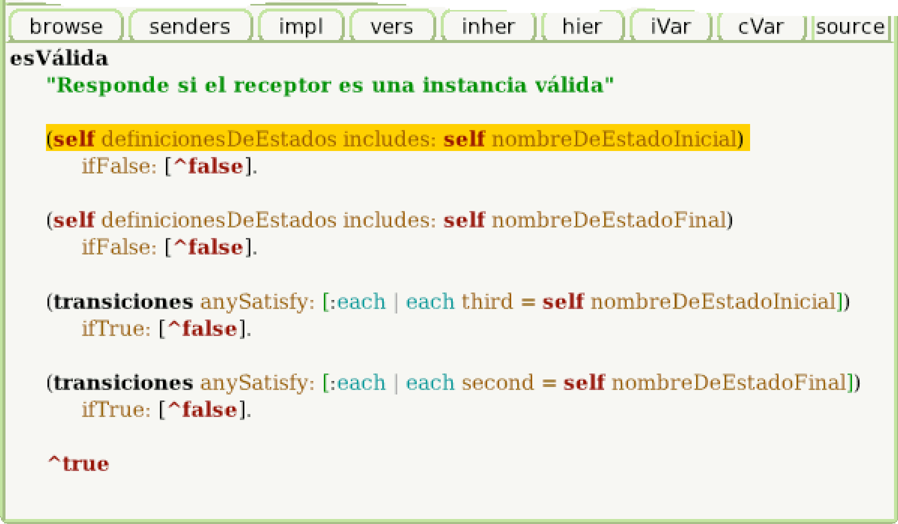
¿Cómo se divide un programa en métodos?

– Dividir el programa en métodos que lleven a cabo una tarea bien identificable. Mantener todas las operaciones de un método en el mismo nivel de abstracción.

Más información en el libro: “Smalltalk Best Practice Patterns” (ver bibliografía).

Usaremos, nuevamente, la funcionalidad `'extract method'` del Refactoring Browser para desmenuzar el método grande en pequeñas partes.

Primero seleccionamos la expresión de la primera condición.



Y lo extraemos a un método llamado `tieneEstadoInicial`. Hacemos lo mismo para la próxima expresión extrayéndola en un método llamado `tieneEstadoFinal`.

Seguimos con la siguiente expresión y la extraemos a un método llamado `llegaAlgunaTransicionAlEstadoInicial`. Y hacemos lo mismo para la última condición extrayéndola a un método de nombre `saleAlgunaTransicionDelEstadoFinal`

El método, ahora, luce como verdadera documentación.

```
esVálida
    "Responde si el receptor es una instancia válida"

    self tieneEstadoInicial ifFalse: [^false].
    self tieneEstadoFinal ifFalse: [^false].
    self llegaAlgunaTransicionAlEstadoInicial ifTrue: [^false].
    self saleAlgunaTransicionDelEstadoFinal ifTrue: [^false].
    ^true
```

Y comentamos los métodos extraídos por el Refactoring Browser.

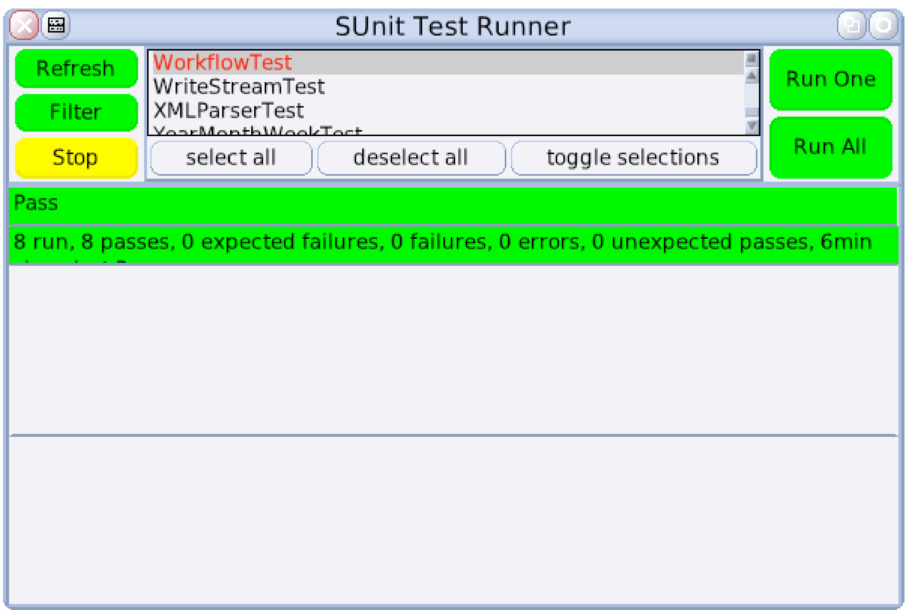
```
tieneEstadoInicial
    "Responde si el receptor tiene algún estado inicial"

    ^self definicionesDeEstados includes: self nombreDeEstadoInicial
```

```
tieneEstadoFinal
    "Responde si el receptor tiene algún estado final"

    ^self definicionesDeEstados includes: self nombreDeEstadoFinal
```

```
llegaAlgunaTransicionAlEstadoInicial
    "Responde si alguna transición del receptor tiene como llegada el estado
inicial"

    ^transiciones anySatisfy: [:each | each third = self nombreDeEstadoInicial]
```

```
saleAlgunaTransicionDelEstadoFinal
    "Responde si alguna transición del receptor tiene como partida el estado
final"

    ^transiciones anySatisfy: [:each | each second = self nombreDeEstadoFinal]
```

Ejecutamos los tests y seguimos en verde.



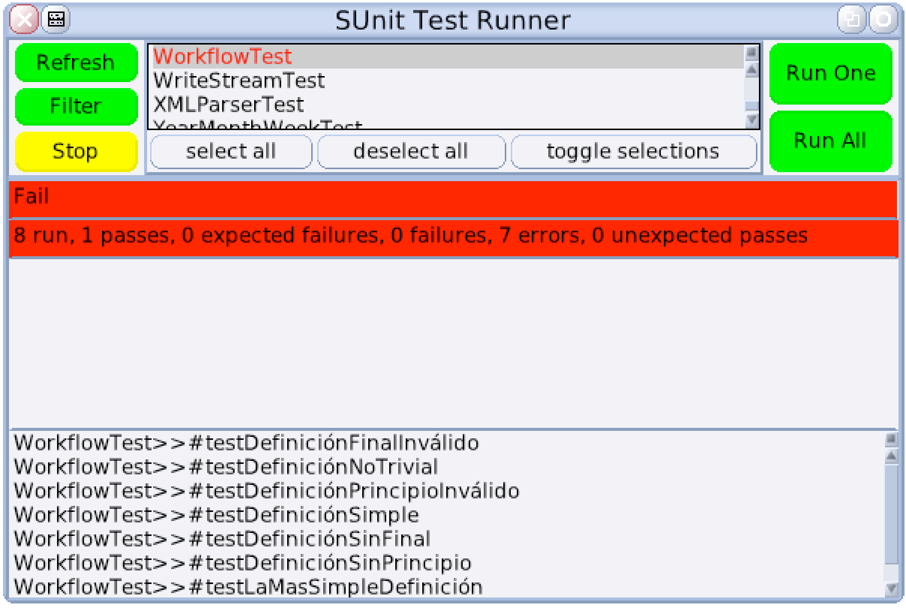
Hasta este momento tenemos guardada toda la información de las transiciones en un Array. No es algo muy orientado a objetos que digamos, así que vamos a cambiar el método `#agregarDefiniciónDeTransición:desde:hasta:` de la siguiente forma:

```
agregarDefiniciónDeTransición: transiciónString desde: estadoDesdeString
```

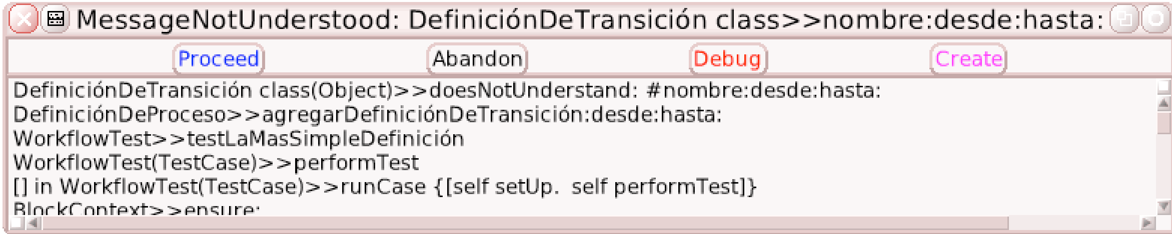
```
hasta: estadoHastaString

   "Agrega una definición de transición con el nombre dado, que une dos
estados de nombres dados"

   | transición |

   transición := DefiniciónDeTransición
                    nombre: transiciónString
                    desde: estadoDesdeString
                    hasta: estadoHastaString.

   transiciones add: transición
```

Creamos la clase `DefiniciónDeTransición` al aceptar, en la categoría de clases `'Workflow'`. Y confirmamos que el selector `#nombre:desde:hasta:` es correcto.

Ejecutamos los tests.



Ahora elegimos alguno de los test en rojo para arreglar. Creo que sería mejor comenzar por el test `#testLaMasSimpleDefinición` ya que parece ser el caso más simple de arreglar. Pedimos el depurador sobre ese test (haciendo clic en el panel inferior).



Y vemos que tenemos que crear el método en la clase `DefiniciónDeTransición class`, en la categoría de métodos llamada `'instance creation'`. y lo implementamos así:

```
nombre: nombreString desde: desdeString hasta: hastaString
   "Responde una nueva instancia del receptor"

   ^ self new initializeNombre: nombreString desde: desdeString hasta: hastaString
```

Aceptamos y continuamos.

Ahora debemos implementar el método `#initializeNombre:desde:hasta:` en la clase `DefiniciónDeTransición`, en una nueva categoría de nombre `'initialization'`, de la siguiente forma:

```
initializeNombre: nombreString desde: desdeString hasta: hastaString
   "Inicializa el receptor"

   nombre := nombreString.
   desde := desdeString.
   hasta := hastaString.
```

Y creamos las variables de instancia `nombre`, `desde` y `hasta` al aceptar.

Continuamos.

Ahora vemos que un objeto de clase `DefiniciónDeTransición` no entiende el mensaje `#second`. Eso es natural ya que reemplazamos un `Array` por una clase nuestra. Modificamos el método `#definicionesDeEstados` de la siguiente forma:

```
definicionesDeEstados
   "Responde las definiciones de estados del receptor"
```

```
| estadosDesde estadosHasta |
estadosDesde := transiciones collect: [:each | each desde].
estadosHasta := transiciones collect: [:each | each hasta].
^ (estadosDesde , estadosHasta) asSet.
```

Aceptamos y continuamos.

Ahora debemos implementar los métodos `#desde` y `#hasta`, en la clase `DefiniciónDeTransición`, en la categoría `'accessing'` de la siguiente forma:

```
desde
   "Responde el nombre de estado de partida del receptor"
   ^ desde
```

```
hasta
   "Responde el nombre de estado de llegada del receptor"
   ^ hasta
```

Continuamos.

Ahora vemos que nuestra clase no entiende el mensaje `#third`. Modificamos el método `#llegaAlgunaTransicionAlEstadoInicial` de la siguiente forma:

```
llegaAlgunaTransicionAlEstadoInicial
   "Responde si alguna transición del receptor tiene como llegada el estado
inicial"

   ^transiciones anySatisfy: [:each | each hasta = self nombreDeEstadoInicial]
```

Continuamos.

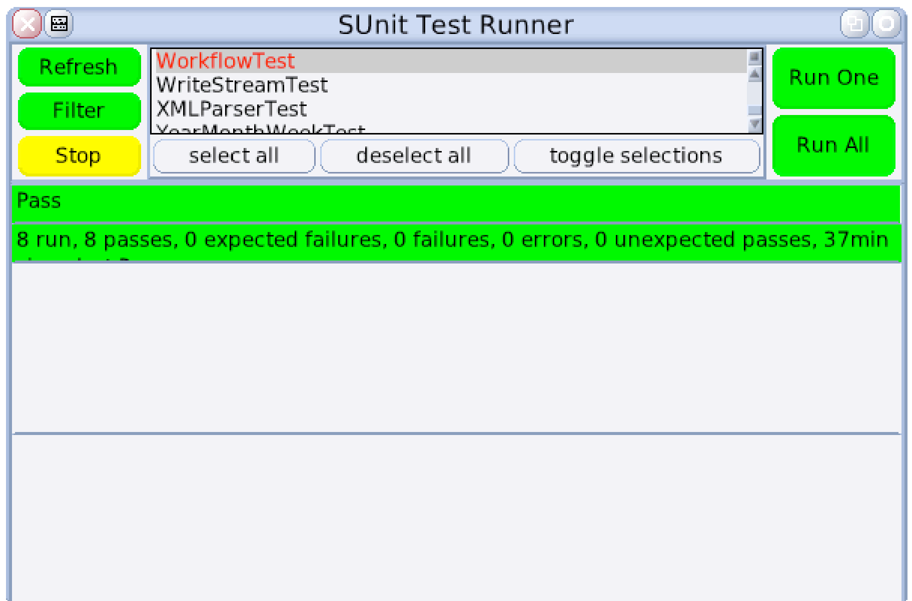
Ahora cambiamos el método `#saleAlgunaTransicionDelEstadoFinal`.

```
saleAlgunaTransicionDelEstadoFinal
   "Responde si alguna transición del receptor tiene como partida el estado
final"
```

```
^transiciones anySatisfy: [:each | each desde = self nombreDeEstadoFinal]
```

Continuamos.

Ahora ejecutamos todos los tests nuevamente para ver que nos hace falta arreglar.



No nos queda nada para arreglar.

Los tests aumentan la confianza

¿Cuantas veces les ha pasado no querer cambiar código mal escrito por no estar seguro si romperíamos algo?

Los sistemas desarrollados con TDD están “controlados” por decenas o centenares de tests. Podemos estar prácticamente seguros que, si refactorizamos algo y los tests funcionan, no hemos roto nada.

Otra validación que vamos a hacer es asegurarnos que todos los estados sean accesibles desde el estado inicial.

Ingresamos el siguiente test en la categoría `'running - instancias inválidas'`:

```
testDefiniciónConEstadosInalcanzables
   "Prueba una definición donde hay estados que no se pueden alcanzar desde el
estado inicial"

   | definiciónDeProceso |

   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'comenzar y terminar'
      desde: 'principio'
      hasta: 'fin'.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'terminar'
      desde: 'A'
      hasta: 'fin'.

   self shouldnt: [definiciónDeProceso esVálida]
```

Ejecutamos los tests y estamos en amarillo.

Modificamos el método `#esVálida` de la siguiente forma.

```
esVálida
   "Responde si el receptor es una instancia válida"

   self tieneEstadoInicial ifFalse: [^false].
   self tieneEstadoFinal ifFalse: [^false].
   self llegaAlgunaTransicionAlEstadoInicial ifTrue: [^false].
   self saleAlgunaTransicionDelEstadoFinal ifTrue: [^false].
   self tieneEstadosInalcanzables ifTrue:[^false].
   ^true
```

Ejecutamos los tests nuevamente para tener la oportunidad de invocar el pre-depurador y usarlo para implementar el método `#tieneEstadosInalcanzables` en la categoría `'testing'`.

```
tieneEstadosInalcanzables
   "Responde si el receptor tiene estados inalcanzables desde el estado inicial"

   ^ self estadosInalcanzables notEmpty
```

Luego implementamos el método `#estadosInalcanzables` en la categoría `'accessing'`.

```
estadosInalcanzables
   "Responde los estados inalcanzables desde el estado inicial"

   ^ self definicionesDeEstados copyWithoutAll: self estadosAlcanzables
```

Y el método `#estadosAlcanzables`, también en la categoría `'accessing'`.

```
estadosAlcanzables
   "Responde los estados alcanzables desde el estado inicial"

   | aVisitar resultado |

   aVisitar := OrderedCollection with: self nombreDeEstadoInicial.
   resultado := aVisitar asSet.

   [aVisitar isEmpty]
      whileFalse: [
         | actual nuevos |

         actual := aVisitar removeFirst.
         nuevos := (self estadosAlcanzablesDesde: actual) copyWithoutAll: resultado.

         resultado addAll: nuevos.
         aVisitar addAll: nuevos.
      ].

   ^ resultado
```

Y, después, el método `#estadosAlcanzablesDesde:` en la categoría `'accessing'`.

```
estadosAlcanzablesDesde: aDefiniciónDeEstado
   "Responde los estados alcanzables desde el estado dado"

   ^ transiciones
         select:[:each | each desde = aDefiniciónDeEstado]
         thenCollect:[:each | each hasta]
```

Colecciones – mensaje #select:thenCollect:

Mensaje de utilidad que se comporta como un #select: seguido de un #collect:.

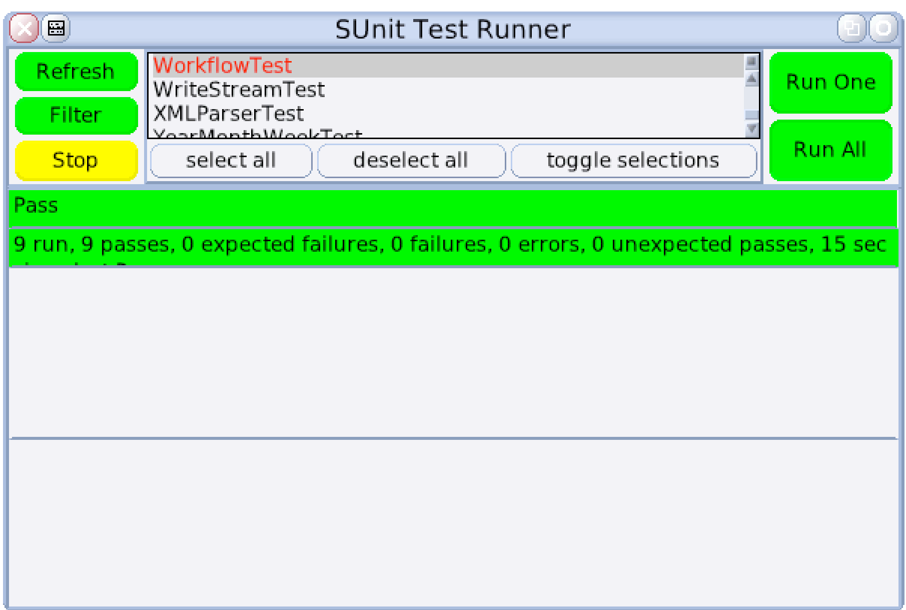
Colecciones – mensaje #select:

Evalúa el bloque dado por cada elemento del receptor como argumento. Colecciona los elementos en los cuales el bloque evalúa a `true` (verdadero) en una colección del tipo del receptor. Responde esa colección nueva como resultado.

Ejemplos:

```
'Un String' select: [:each | each isUppercase].

#(1 2 3 4 5 6 7 8 9 10) select: [:each | each odd].

Smalltalk allClasses select:[:each | each superclass == TestCase].
```

Y estamos en verde.



El último código ingresado quedó muy poco orientado a objetos. La causa principal es que todavía no creamos una clase para representar las definiciones de estado y estamos manipulando strings. Vamos a crear la clase que corresponde, comenzamos modificando el método `DefiniciónDeProceso>>agregarDefiniciónDeTransición:desde:hasta:` de la siguiente forma:

```
agregarDefiniciónDeTransición: transiciónString desde: estadoDesdeString
hasta: estadoHastaString

   "Agrega una definición de transición con el nombre dado, que une dos
estados de nombres dados"
```

```
| estadoDesde estadoHasta transición |

estadoDesde := self estadoDeNombre: estadoDesdeString.
estadoHasta := self estadoDeNombre: estadoHastaString.

transición := DefiniciónDeTransición
                    nombre: transiciónString
                    desde: estadoDesde
                    hasta: estadoHasta.

transiciones add: transición
```

Y ejecutamos los tests. Casi todos fallan. Hacemos clic sobre alguno de los tests en rojo para hacernos del depurador y poder implementar allí.

Creamos el método `#estadoDeNombre:`, en la categoría `'accessing'`, de la siguiente forma:

```
estadoDeNombre: aString
   "Responde un estado de nombre dado. Crea uno nuevo si es necesario."

   ^ estados
         at: aString
         ifAbsentPut:[DefiniciónDeEstado nombre: aString].
```

Colecciones – mensaje #at:ifAbsentPut:

Responde el elemento de nombre dado. Si no hay ningún elemento con ese nombre, se evalúa el bloque dado y el resultado de dicha evaluación se guarda en el receptor y se responde al remitente.

Creamos la variable de instancia `estados` y la clase `DefiniciónDeEstado` (en la categoría `Workflow`).

La nueva variable de instancia `estados` tendrá un diccionario, así que modificamos el método `DefiniciónDeProceso>>initialize` de la siguiente forma:

```
initialize
   "Inicializa el receptor"
```

```
transiciones := Set new.
estados := Dictionary new.
```

Colecciones – Dictionary

El Dictionary (Diccionario) es una colección que puede verse desde 2 puntos de vista:

– Como un Set de asociaciones clave->valor

– Como un contenedor donde los elementos son nombrados desde el exterior, donde el nombre puede ser cualquier objeto que responda al mensaje #=.

Ejemplos:

```
"instanciar, y llenar, un diccionario"
| diccionario |

diccionario := Dictionary new.

diccionario at: 'hoy' put: Date today.
diccionario at: 'ahora' put: Time now.
diccionario at: false put: 'es false'.

"algunas formas de acceder al diccionario"
diccionario at: 'hoy'.
diccionario at: 'mañana' ifAbsent:[nil].
diccionario at: 'mañana'.
diccionario at: 'mañana' ifAbsentPut:[Date tomorrow].
diccionario at: 'mañana'.

"otras formas de acceso"
diccionario keys.
diccionario values.

"algunas iteraciones"
Transcript clear.

Transcript cr; show: '#do:'; cr.
diccionario do: [:each |
    Transcript show: ' ' , each asString; cr.
].

Transcript cr; show: '#associationsDo:'; cr.
diccionario associationsDo: [:each |
    Transcript show: ' ' , each asString; cr.
].

Transcript cr; show: '#keysAndValuesDo:'; cr.
diccionario keysAndValuesDo:[:eachKey :eachValue |
    Transcript show: ' ' , eachKey asString , ' - ' , eachValue asString; cr.
].
```

Transcript

Transcript es una variable global que apunta a una ventana de Transcript (si es que está abierta). Normalmente el Transcript se usa para mostrar pequeños mensajes que ayudan a la depuración del código.

Para obtener una ventana de Transcript hay que usar el menú del mundo >> `'open...'` >> `'transcript (t)'` (o presionar ALT-t).

El Transcript responde a una serie de mensajes muy útiles a la hora de depurar:

`#show:` Muestra el argumento, convertido a `String`, en la ventana.

`#cr` Hace un salto de línea en la ventana.

`#clear` Limpia el contenido de la ventana.

Implementamos el método `DefiniciónDeProceso class>>nombre:`, en la categoría `'instance creation'`, de la siguiente forma:

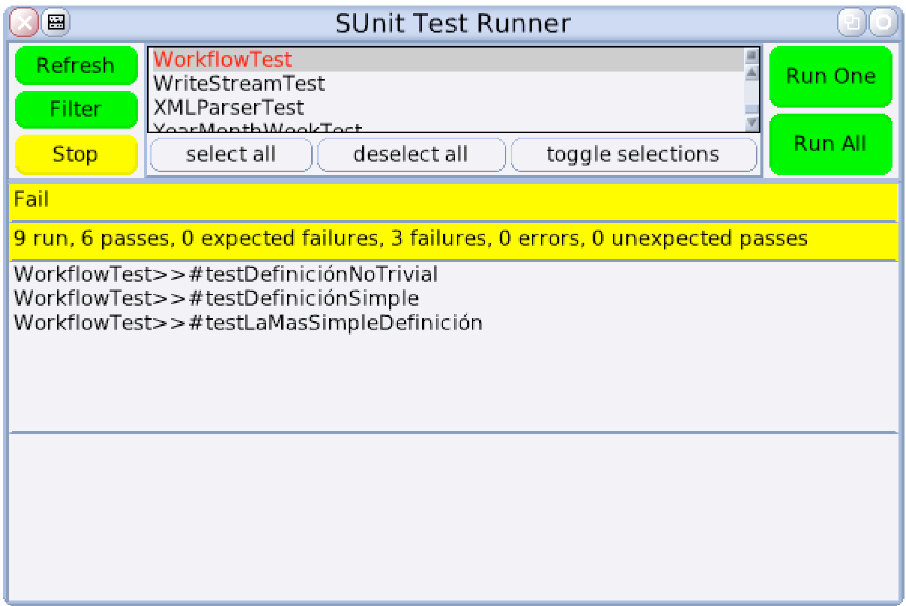
```
nombre: aString
   "Responde una nueva instancia del receptor"

   ^ self new initializeNombre: aString
```

Y el método `DefiniciónDeProceso>>initializeNombre:`, en la nueva categoría `'initialization'`, así:

```
initializeNombre: aString
   "Inicializa el receptor"

   nombre := aString
```

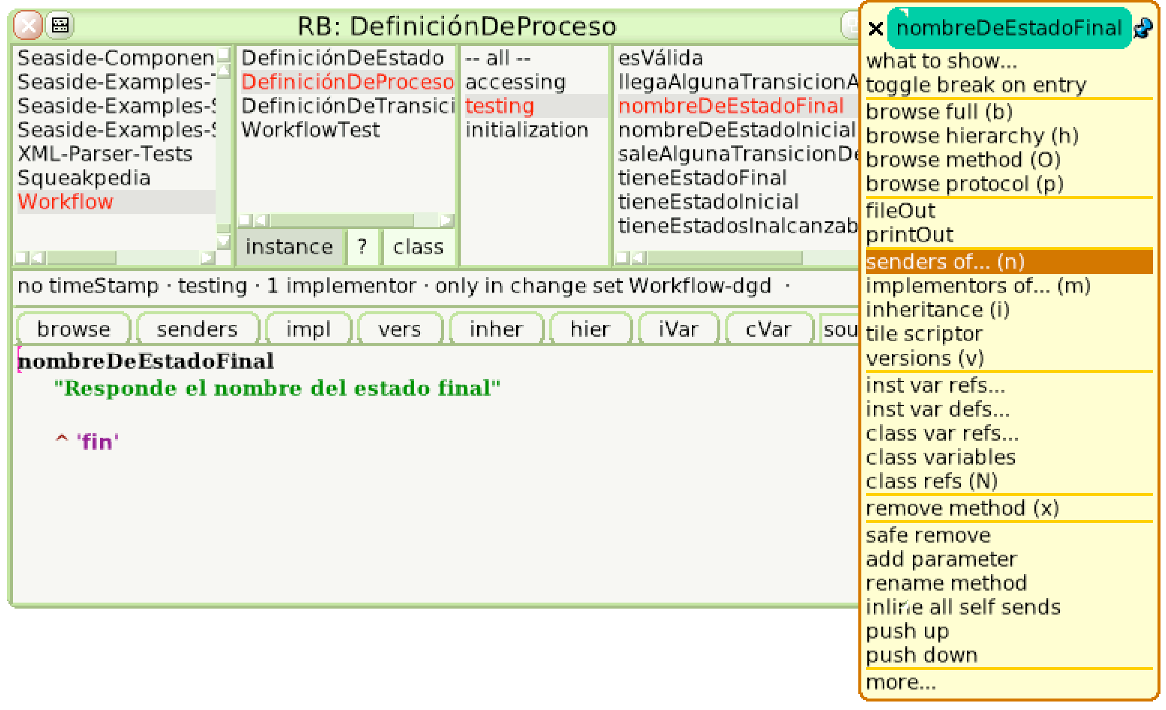
Y creamos la variable de instancia `nombre`.

Ejecutamos los tests y estamos en amarillo.

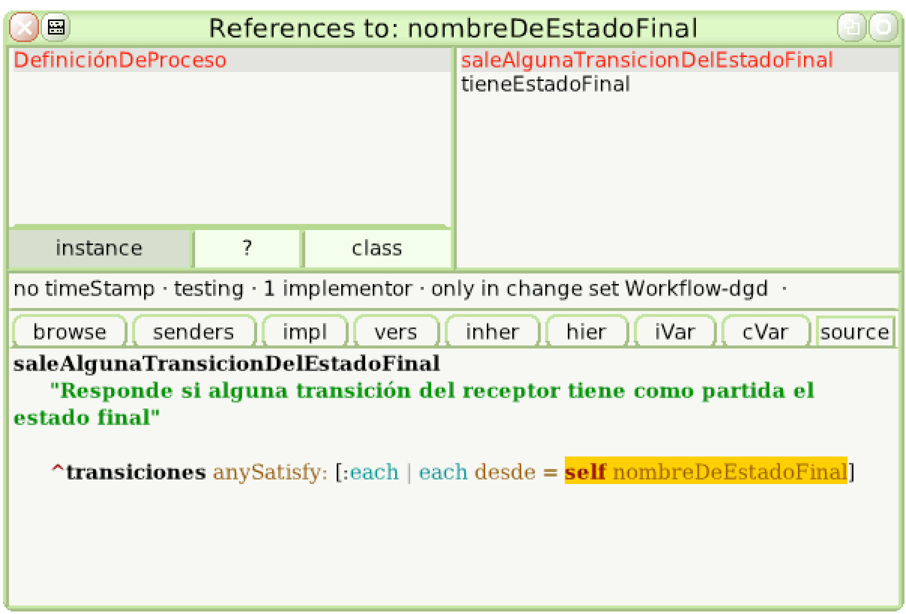


¿Qué paso? Hasta ahora veníamos utilizando sólo un `String` en lugar de un objeto de clase `DefiniciónDeEstado` para los estados. Estamos usando 2 constantes (también `String`) para los especiales estados inicial y final. Si buscamos los senders de ambos métodos de constante (`#nombreDeEstadoInicial` y `#nombreDeEstadoFinal`) seguramente encontraremos el lugar (o los lugares) donde tenemos que meter mano.

Para encontrar los senders de `#nombreDeEstadoInicial` podemos usar la opción `'senders of... (n)'` del menú contextual del panel de métodos en un Browser de Clases.



Y obtenemos un Browser como el siguiente:



Y encontramos 2 métodos que tenemos que cambiar. Repetimos el procedimiento para el método `#nombreDeEstadoInicial` y encontramos otros 3 métodos a arreglar. Los arreglamos de la siguiente forma:

```
tieneEstadoInicial
   "Responde si el receptor tiene algún estado inicial"

   ^self definicionesDeEstados
            anySatisfy:[:each | each nombre = self nombreDeEstadoInicial]
```

```
tieneEstadoFinal
   "Responde si el receptor tiene algún estado final"

   ^self definicionesDeEstados
            anySatisfy:[:each | each nombre = self nombreDeEstadoFinal]
```

```
llegaAlgunaTransicionAlEstadoInicial
   "Responde si alguna transición del receptor tiene como llegada el estado inicial"

   ^transiciones anySatisfy: [:each | each hasta nombre = self nombreDeEstadoInicial]
```

```
saleAlgunaTransicionDelEstadoFinal
   "Responde si alguna transición del receptor tiene como partida el estado final"

   ^transiciones anySatisfy: [:each | each desde nombre = self nombreDeEstadoFinal]
```

```
estadosAlcanzables
   "Responde los estados alcanzables desde el estado inicial"

   | aVisitar resultado |

   aVisitar := OrderedCollection with: (self estadoDeNombre: self nombreDeEstadoInicial).
   resultado := aVisitar asSet.

   [aVisitar isEmpty]
      whileFalse: [
         | actual nuevos |

         actual := aVisitar removeFirst.
         nuevos := (self estadosAlcanzablesDesde: actual) copyWithoutAll: resultado.

         resultado addAll: nuevos.
         aVisitar addAll: nuevos.
      ].

   ^ resultado
```

E implementamos el método `DefiniciónDeProceso>>nombre`, en la categoría `'accessing'`.

```
nombre
   "Responde el nombre del receptor"
   ^ nombre
```

Ahora estamos en verde, tiempo de limpiar el código.

La vieja implementación del método `DefiniciónDeProceso>>definicionesDeEstados` puede ser reemplazada por la siguiente:

```
definicionesDeEstados
   "Responde las definiciones de estados del receptor"
```

```
^ estados values
```

Y seguimos en verde.

La expresión de abajo, que introdujimos en el método `DefiniciónDeProceso>>estadosAlcanzables`, es clara candidata para un “extract method”.

```
(self estadoDeNombre: self nombreDeEstadoInicial)
```

Lo hacemos y dejamos los métodos de la siguiente forma:

```
estadosAlcanzables
   "Responde los estados alcanzables desde el estado inicial"

   | aVisitar resultado |

   aVisitar := OrderedCollection with: self estadoInicial.
   resultado := aVisitar asSet.

   [aVisitar isEmpty]
      whileFalse: [
         | actual nuevos |

         actual := aVisitar removeFirst.
         nuevos := (self estadosAlcanzablesDesde: actual) copyWithoutAll: resultado.

         resultado addAll: nuevos.
         aVisitar addAll: nuevos.
      ].

   ^ resultado
```

```
estadoInicial
   "Responde el estado inicial"

   ^ self estadoDeNombre: self nombreDeEstadoInicial
```

Podemos refactorizar algunos otros 'senders' de `#nombreDeEstadoInicial` para aprovecharnos del nuevo método creado y evitar comparaciones por nombre.

```
llegaAlgunaTransicionAlEstadoInicial
   "Responde si alguna transición del receptor tiene como llegada el estado
inicial"

   ^transiciones anySatisfy: [:each | each hasta = self estadoInicial]
```

Y hacemos lo mismo para el estado final.

```
estadoFinal
   "Responde el estado final"

   ^ self estadoDeNombre: self nombreDeEstadoFinal
```

```
saleAlgunaTransicionDelEstadoFinal
   "Responde si alguna transición del receptor tiene como partida el estado
final"

   ^transiciones anySatisfy: [:each | each desde = self estadoFinal]
```

Y seguimos en verde, pero con el código un poco más limpio.

Lo último que debemos validar es que no existan más de 1 transición con el mismo nombre para el estado de salida. Dicho de otra forma: la combinación estadoDesde-nombreDeTransición debe ser única. Volcamos eso en un test.

```
testDefiniciónConTransicionesDuplicadas
   "Prueba una definición donde hay transiciones con el mismo nombre que
salen de un determinado estado"

   | definiciónDeProceso |

   definiciónDeProceso := DefiniciónDeProceso new.

   definiciónDeProceso
      agregarDefiniciónDeTransición: 'comenzar'
      desde: 'principio'
      hasta: 'fin'.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'comenzar'
      desde: 'principio'
      hasta: 'A'.
   definiciónDeProceso
```

```
        agregarDefiniciónDeTransición: 'terminar'
        desde: 'A'
        hasta: 'fin'.

    self shouldnt: [definiciónDeProceso esVálida]
```

Obtenemos el amarillo.

Implementamos lo siguiente:

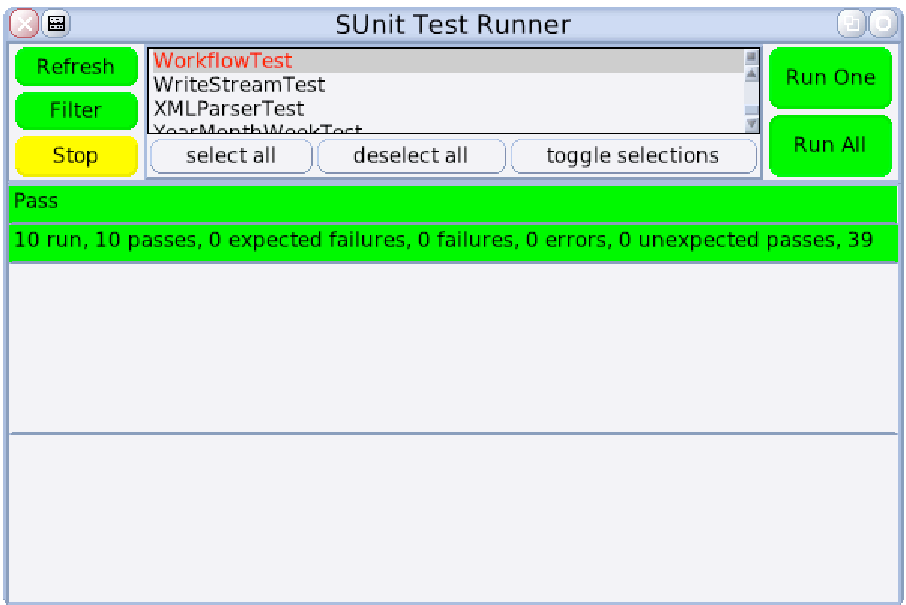
```
esVálida
    "Responde si el receptor es una instancia válida"

    self tieneEstadoInicial ifFalse: [^false].
    self tieneEstadoFinal ifFalse: [^false].
    self llegaAlgunaTransicionAlEstadoInicial ifTrue: [^false].
    self saleAlgunaTransicionDelEstadoFinal ifTrue: [^false].
    self tieneEstadosInalcanzables ifTrue:[^false].
    self tieneAlgunEstadoConTransicionesDuplicadas ifTrue:[^false].
    ^true
```

```
tieneAlgunEstadoConTransicionesDuplicadas
    "Responde si el receptor tiene algún estado con transiciones
    duplicadas "
    ^ self definicionesDeEstados
        anySatisfy: [:eachEstado |
            | transicionesDesde nombresDesde |
            transicionesDesde := self transicionesDesde: eachEstado.
            nombresDesde := transicionesDesde
                            collect: [:eachTransicion | eachTransicion nombre].
            transicionesDesde size ~= nombresDesde asSet size
        ]
```

```
transicionesDesde: aDefiniciónDeEstado
    "Responde las transiciones que tienen el estado dado como 'desde'"
    ^ transiciones
        select:[:each | each desde = aDefiniciónDeEstado]
```

```
nombre
    "Responde el nombre del receptor"
    ^ nombre
```

Y estamos en verde.



Ahora deberíamos mover la funcionalidad que tenemos en `DefiniciónDeProceso` pero que le corresponde a la recién creada clase `DefiniciónDeEstado`.

Para mover esa funcionalidad vamos a modificar la instanciación de las transiciones. Haremos que la transición le indique a cada uno de los estados involucrados sobre su propia existencia para que estos, a su vez, conozcan la estructura de la cual son parte.

Cambiamos los nombres de los argumentos en el constructor de `DefiniciónDeTransición`, indicando que ahora esperamos objetos `DefiniciónDeEstado` y no objetos `String`.

```
nombre: nombreString desde: desdeDefiniciónDeEstado hasta: hastaDefiniciónDeEstado
   "Responde una nueva instancia del receptor"

   ^ self new
         initializeNombre: nombreString
         desde: desdeDefiniciónDeEstado
         hasta: hastaDefiniciónDeEstado
```

Ahora modificamos el método de inicialización arreglando, también, los nombres de los argumentos y notificando a las definiciones de estado sobre la transición recién creada.

```
initializeNombre: nombreString desde: desdeDefiniciónDeEstado hasta:
```

```
hastaDefiniciónDeEstado
    "Inicializa el receptor"

    nombre := nombreString.
    desde := desdeDefiniciónDeEstado.
    hasta := hastaDefiniciónDeEstado.

    desde nuevaTransiciónSaliente: self.
    hasta nuevaTransiciónEntrante: self.
```

Implementamos los métodos `#nuevaTransiciónSaliente:` y `#nuevaTransiciónEntrante:`, en la clase `DefiniciónDeEstado`, creando las 2 nuevas variables de instancia necesarias.

```
nuevaTransiciónSaliente: aDefiniciónDeTransición
    "Agrega, en la colección de transiciones salientes del receptor, la definición
de transición dada"

    transicionesSalientes add: aDefiniciónDeTransición
```

```
nuevaTransiciónEntrante: aDefiniciónDeTransición
    "Agrega, en la colección de transiciones entrantes del receptor, la definición
de transición dada"

    transicionesEntrantes add: aDefiniciónDeTransición
```

E inicializamos las recién creadas variables de instancia:

```
initialize
    "Inicializa al receptor"

    transicionesSalientes := Set new.
    transicionesEntrantes := Set new.
```

Implementamos el método `#estadosAlcanzables`, en la clase `DefiniciónDeEstado`, que será el que reemplace al método `DefiniciónDeProceso>>estadosAlcanzablesDesde:`.

```
estadosAlcanzables
    "Responde los estados alcanzables desde el receptor"

    ^ transicionesSalientes collect:[:each | each hasta]
```

Modificamos el método `DefiniciónDeProceso>>estadosAlcanzables`

```
estadosAlcanzables
   "Responde los estados alcanzables desde el estado inicial"

   | aVisitar resultado |

   aVisitar := OrderedCollection with: self estadoInicial.
   resultado := aVisitar asSet.

   [aVisitar isEmpty]
      whileFalse: [
         | actual nuevos |

         actual := aVisitar removeFirst.
         nuevos := actual estadosAlcanzables copyWithoutAll: resultado.

         resultado addAll: nuevos.
         aVisitar addAll: nuevos.
      ].

   ^ resultado
```

Borramos el método `DefiniciónDeProceso>>estadosAlcanzablesDesde:` (usando el menú contextual sobre el panel de métodos, opción `'remove method (x)'`).

Ejecutamos los tests, y estamos en verde.

Otra funcionalidad que debemos mover es la prueba por transiciones salientes duplicadas. Agregamos el siguiente método, en la clase `DefiniciónDeEstado`.

```
tieneTransicionesDuplicadas
   "Responde si el receptor tiene más de 1 transición saliente con el mismo
nombre"

   | nombres |

   nombres := (transicionesSalientes collect:[:each | each nombre]) asSet.

   ^ transicionesSalientes size ~= nombres size
```

Y cambiamos el siguiente método en la clase `DefiniciónDeProceso`.

```
tieneAlgunEstadoConTransicionesDuplicadas
   "Responde si el receptor tiene algún estado con transiciones
   duplicadas "
   ^ self definicionesDeEstados
        anySatisfy: [:each | each tieneTransicionesDuplicadas]
```

Borramos el método `DefiniciónDeProceso>>transicionesDesde:`.

Ejecutamos los tests y seguimos en verde, pero esta vez el código escrito es mucho más orientado a objetos.

Llegó la hora de hacer algo con la Definiciones de Proceso. Lo primero que tiene que hacer una Definición de Proceso es crear una Instancia de Proceso. Cabe aclarar que cuando hablamos de una Instancia de Proceso estamos usando terminología de Workflow y no nos estamos refiriendo a instancias de Smalltalk.

La Instancia de Proceso más simple que se me ocurre es la siguiente:

```
testLaMasSimpleInstanciaDeProceso
   "Prueba la definición más simple posible, con su instancia"

   | definiciónDeProceso definiciónEstadoInicial proceso |

   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'comenzar y terminar'
      desde: 'principio'
      hasta: 'fin'.

   definiciónEstadoInicial := definiciónDeProceso estadoInicial.

   proceso := definiciónDeProceso nuevaInstancia.
   self should:[proceso class = Proceso].
   self should:[proceso estado definición = definiciónEstadoInicial].
```

Creamos la clase `Proceso` al aceptar el método, en la categoría `'Workflow'`.

Ejecutamos los tests y, previsiblemente, estamos en rojo.

Creamos el método `#nuevaInstancia` en la clase `DefiniciónDeProceso`, en una nueva

categoría de nombre `'instancias'`.

```
nuevaInstancia
   "Crea una nueva instancia de proceso"

   ^ Proceso definición: self.
```

Ahora creamos el método `Proceso class>>definición:`, en la categoría `'instance creation'`, de la siguiente forma:

```
definición: aDefiniciónDeProceso
   "Crea una nueva instancia del receptor"

   ^ self new initializeDefinición: aDefiniciónDeProceso
```

Y el método `Proceso>>initializeDefinición:`, en la categoría `'initialization'`.

```
initializeDefinición: aDefiniciónDeProceso
   "Inicializa la definición del receptor"

   definición := aDefiniciónDeProceso
```

Y creamos la variable de instancia `definición` al aceptar.

Ahora debemos implementar el método `DefiniciónDeProceso>>estado`.

Las instancias de proceso siempre tienen un estado asociado. Los procesos recién empezados están en el estado inicial, y los procesos terminados están en el estado final.

Modificamos el método `Proceso>>initializeDefinición:` de la siguiente forma:

```
initializeDefinición: aDefiniciónDeProceso
   "Inicializa la definición del receptor"

   definición := aDefiniciónDeProceso.

   estado := definición estadoInicial nuevaInstancia.
```

Creamos la variable de instancia `estado` al aceptar. También creamos el siguiente método:

```
estado
   "Responde el estado actual del receptor"

   ^ estado
```

Ahora creamos el siguiente método en la clase `DefiniciónDeEstado`, en la categoría `'instancias'`.

```
nuevaInstancia
   "Crea una nueva instancia de estado"

   ^ Estado definición: self
```

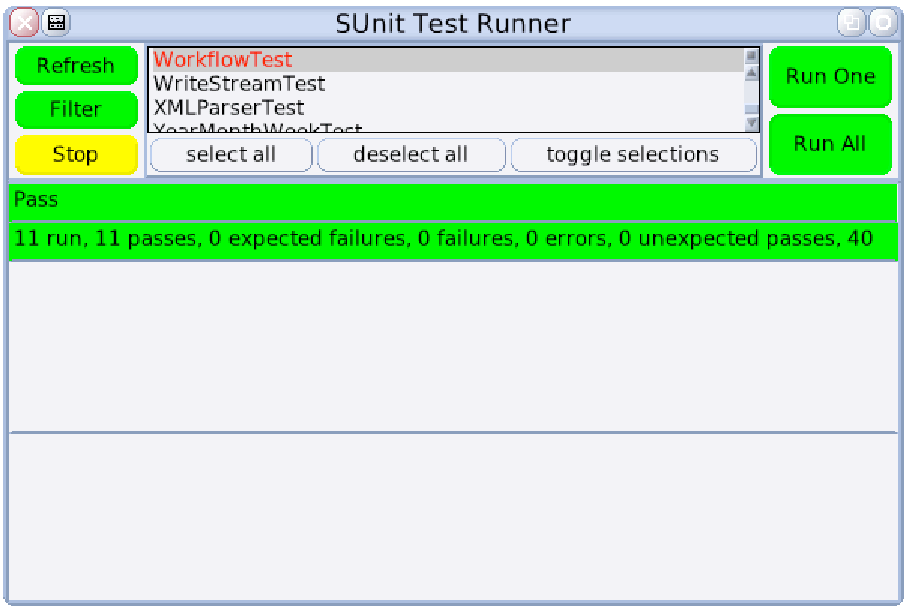
Creamos la clase `Estado`, en la categoría `'Workflow'`, al aceptar.

Ahora creamos, en la clase `Estado`, el método de clase `#definición:`, el método de instancia `#initializeDefinición:` y la variable de instancia `definición` de la misma forma que lo hicimos en la clase `Proceso`.

También implementamos, en la clase `Estado`, así:

```
definición
   "Responde la definición del receptor"

   ^ definición
```

Ejecutamos los tests y ¡Estamos en verde!



De nada sirve un motor de workflow si no podemos ejecutar código arbitrario en cada cambio de estado. A su vez, este código será el responsable de decidir que transición tomar para pasar de un estado a otro. Para eso vamos a decirle a la Definición de Proceso que utilice un objeto "gestor" para comunicarle los cambios de estado. El gestor, además de hacer lo que se necesite por los cambios de estado, será el que decidirá que transición usar para pasar del estado actual al próximo.

Vamos a hacer el caso más simple posible, con un gestor que haga avanzar el proceso del estado inicial al estado final.

```
testLaMasSimpleInstanciaDeProcesoConGestor
   "Prueba la definición más simple posible, con su instancia de proceso y su gestor"

   | definiciónDeProceso proceso gestor estados |

   definiciónDeProceso := DefiniciónDeProceso new.
   definiciónDeProceso
      agregarDefiniciónDeTransición: 'comenzar y terminar'
      desde: 'principio'
      hasta: 'fin'.

   gestor := GestorConBloque bloque: [:proc |
            (proc estado nombre = 'principio')
               ifTrue:['comenzar y terminar']
               ifFalse:[nil]
         ].
   definiciónDeProceso gestor: gestor.

   proceso := definiciónDeProceso nuevaInstancia.
```

```
estados := proceso estados.

self should:[estados size = 2].
self should:[estados first definición = definiciónDeProceso estadoInicial].
self should:[estados second definición = definiciónDeProceso estadoFinal].
self should:[estados second transición nombre = 'comenzar y terminar'].
```

Con este test estamos definiendo prácticamente el funcionamiento completo del motor de Workflow.

Antes de ponernos a trabajar para que el test funcione, vamos a desmenuzar su contenido para ver que es lo que esperamos del funcionamiento del motor. Lo primero diferente es que, a la Definición de Proceso, le indicamos cual será su "gestor".

El gestor es un objeto que tiene la oportunidad de hacer lo que se necesite en cada cambio de estado, a su vez es el responsable de indicarnos que transición tomar para pasar del estado actual al siguiente. Vamos a crear una clase `GestorConBloque` que será un *Adapter* que transforme la interfaz de un bloque en la interfaz de un `Gestor`.

Patrón de Diseño – Adapter

El patrón Adapter se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

Problema que soluciona

Se tiene un componente con cierta funcionalidad que se desea aprovechar, pero este componente no cumple determinada interfaz, siendo esto último imperativo.

Implementación

Crear una nueva clase que será el Adaptador, que extienda del componente existente e implemente la interfaz obligatoria. De este modo tenemos la funcionalidad que queríamos y cumplimos la condición de implementar la interfaz.

La diferencia entre los patrones Adapter y Facade, es que el primero reutiliza una interfaz ya existente, mientras que el segundo define una nueva.

Para más información:

- http://es.wikipedia.org/wiki/Adapter_(patrón_de_diseño)
- Libro "Design Patterns - Elements of Reusable Object Oriented Software" (ver bibliografía)

Lo próximo nuevo es el envío del mensaje `#estados` (en plural). Este mensaje tendrá como

respuesta una colección con todos los estados que tuvo el proceso. Es diferente al mensaje `#estado` (en singular), que usamos anteriormente, que sólo contesta el estado actual.

Y por último tenemos el mensaje `#transición` que le enviamos a un Estado. Este mensaje responderá que transición fue la responsable de que ese estado se convirtiera en estado actual.

Al aceptar el método creamos la clase `GestorConBloque`, en la categoría `'Workflow'`.

Ejecutamos los tests, estamos en rojo.

Implementamos el método constructor `GestorConBloque class>>bloque:`, en la categoría `'instance creation'`, de la siguiente forma:

```
bloque: aBlockContext
    "Responde una nueva instancia del receptor"
    ^ self new initializeBloque: aBlockContext
```

Y el método `GestorConBloque>>initializeBloque:`, en la categoría `'initialization'`. Creamos la variable de instancia `bloque` al aceptar el método.

```
initializeBloque: aBlockContext
    "Inicializa el bloque del receptor"
    bloque := aBlockContext
```

Ahora implementamos el método `DefiniciónDeProceso>>gestor:`, en la categoría `'accessing'`. Creamos la variable de instancia `gestor` al aceptar el método.

```
gestor: aGestor
    "Cambia el gestor del receptor"
    gestor := aGestor
```

Ahora implementamos el método `Proceso>>estados`, en la categoría `'accessing'`.

```
estados
    "Responde una colección con todos los estados por los que atravesó el
receptor. Primero el más viejo."
```

```
| resultado estadoActual |

resultado := OrderedCollection new.

estadoActual := self estado.
[estadoActual isNil]
    whileFalse: [
        resultado add: estadoActual.
        estadoActual := estadoActual estadoAnterior.
    ].

^ resultado reversed
```

Y el método `Estado>>estadoAnterior`, en la categoría `'accessing'`, y creamos la variable de instancia `transición` al aceptar.

```
estadoAnterior
    "Responde el estado anterior.
    Si no hay ningún estado anterior, responde nil"

    ^ transición isNil
        ifTrue: [nil]
        ifFalse: [transición desde]
```

Ahora creamos el método `Proceso>>cambiarEstado:`, en la categoría `'private'`.

```
cambiarEstado: aEstado
    "PRIVADO - Cambia el estado actual del receptor."

    estado := aEstado.
    self cambioDeEstado.
```

Métodos privados

Los métodos privados, por convención, se organizan dentro de la categoría `'private'`.

Y el método `#cambioDeEstado`, en la categoría `'private'`.

```
cambioDeEstado
```

```
    "PRIVADO - El estado del receptor acaba de cambiar. Notificar al gestor si
existe."

    | nombreDeTransición |

    definición tieneGestor
        ifFalse:[^ self].

    nombreDeTransición := definición gestor estadoCambiadoEn: self.

    nombreDeTransición isNil
        ifTrue:[
            estado definición tieneTransicionesSalientes
                ifTrue:[self error: 'Nombre de transición inválida: ', nombreDeTransición asString]
        ]
        ifFalse:[
            self aplicarTransiciónDeNombre: nombreDeTransición
        ]
```

Y cambiamos el método `Proceso>>initializeDefinición:` que es, por ahora, el único lugar donde cambiamos de estado.

```
initializeDefinición: aDefiniciónDeProceso
    "Inicializa la definición del receptor"

    definición := aDefiniciónDeProceso.

    self cambiarEstado: definición estadoInicial nuevaInstancia.
```

Implementamos el método `DefiniciónDeProceso>>tieneGestor`, en la categoría `'testing'`.

```
tieneGestor
    "Responde si el receptor tiene un gestor o no"
    ^ gestor notNil
```

Ahora implementamos el método `DefiniciónDeProceso>>gestor`, en la categoría `'accessing'`.

```
gestor
    "Responde el gestor del receptor"
```

```
    ^ gestor
```

Ahora creamos el método `GestorConBloque>>estadoCambiadoEn:`, en la categoría `'events'`.

```
estadoCambiadoEn: aProceso
    "Un proceso ha cambiado su estado actual"
    ^ bloque value: aProceso
```

Y el método `Estado>>nombre`, en la categoría `'accessing'`.

```
nombre
    "Responde el nombre del receptor"
    ^ definición nombre
```

Y `DefiniciónDeEstado>>tieneTransicionesSalientes`, en la categoría `'testing'`.

```
tieneTransicionesSalientes
    "Responde si el receptor tiene o no transiciones salientes"
    ^ transicionesSalientes notEmpty
```

Ahora el método `Proceso>>aplicarTransiciónDeNombre:`, en la categoría `'private'`.

```
aplicarTransiciónDeNombre: aString
    "PRIVADO - Aplica la transición de nombre dado"

    | definiciónDeTransición transición |

    definiciónDeTransición := definición transiciónDeNombre: aString desde: estado.

    transición := definiciónDeTransición nuevaInstanciaDesde: estado.

    self cambiarEstado: transición hasta
```

Y el método `DefiniciónDeProceso>>transiciónDeNombre:desde:`, en la categoría `'accessing'`.

```
transiciónDeNombre: aString desde: anEstado
   "Responde una transición de nombre dado y que sea saliente del estado dado"

   ^ transiciones
         detect:[:each | each nombre = aString and:[each desde = anEstado definición]]
```

Ahora el método `DefiniciónDeTransición>>nuevaInstanciaDesde:`, en la categoría `'instancias'`. Creamos la clase `Transición`, en la categoría `'Workflow'`, al aceptar el método.

```
nuevaInstanciaDesde: anEstado
   "Crea una nueva instancia de transición"
   ^ Transición definición: self desde: anEstado
```

Ahora creamos el método `Transición class>>definición:desde:`, en la categoría `'instance creation'`.

```
definición: aDefiniciónDeTransición desde: anEstado
   "Crea una nueva instancia del receptor"
   ^ self new initializeDefinición: aDefiniciónDeTransición desde: anEstado
```

Y el método `Transición>>initializeDefinición:desde:`, en la categoría `'initialization'`. Creamos las variables de instancia `definición`, `desde` y `hasta` al aceptar el método.

```
initializeDefinición: aDefiniciónDeTransición desde: anEstado
   "Inicializa el receptor"

   definición := aDefiniciónDeTransición.
   desde := anEstado.
   hasta := definición hasta nuevaInstanciaTransición: self.
```

Ahora creamos el método `DefiniciónDeEstado>>nuevaInstanciaTransición:`, en la categoría `'instancias'`.

```
nuevaInstanciaTransición: aTransición
   "Crea una nueva instancia de estado"
```

```
    ^ Estado definición: self transición: aTransición
```

Ahora creamos el método `Estado class>>definición:transición:`, en la categoría `'instance creation'`.

```
definición: aDefiniciónDeEstado transición: aTransición
    "Crea una nueva instancia del receptor"

    ^ self new initializeDefinición: aDefiniciónDeEstado transición: aTransición
```

Y el método `Estado>>initializeDefinición:transición:`, en la categoría `'initialization'`.

```
initializeDefinición: aDefiniciónDeEstado transición: aTransición
    "Inicializa la definición y la transición del receptor"

    definición := aDefiniciónDeEstado.
    transición := aTransición
```

Y ahora implementamos los métodos `Transición>>hasta`, `Transición>>desde`, `Estado>>transición` y `Transición>>nombre`, en la categoría `'accessing'`.

```
hasta
    "Responde el estado 'hasta' del receptor"
    ^ hasta
```

```
desde
    "Responde el estado 'desde' del receptor"
    ^ desde
```

```
transición
    "Responde la transición que dio origen al receptor"
    ^ transición
```

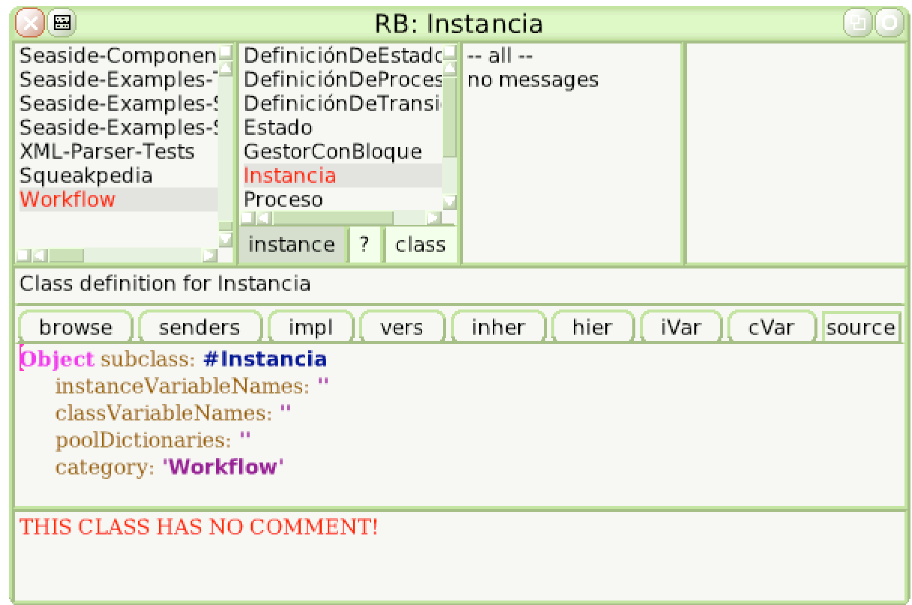
```
nombre
    "Responde el nombre del receptor"
```

```
^ definición nombre
```

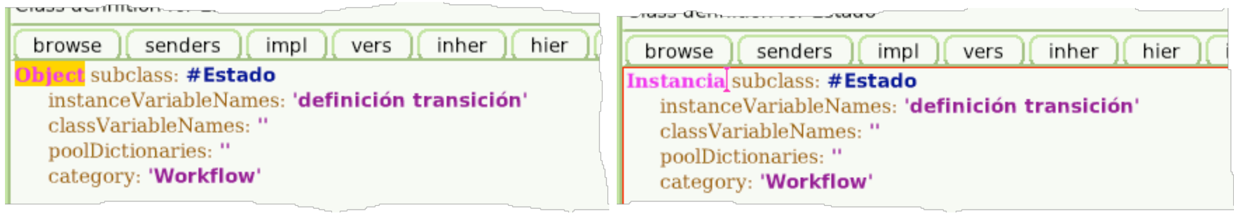
Y, después de tanto trabajo, ejecutamos los tests y ¡estamos en verde!

Ahora llegó el momento de limpiar el código. Hicimos tanto para el último test que, adivino, tendremos un buen trabajo de limpieza.

Lo primero que vemos es algo que ya habíamos previsto al comenzar. Tenemos 2 tipos de objetos: las Definiciones y las Instancias. Si prestamos atención vemos que todas las instancias (`Proceso`, `Estado` y `Transición`) comparten cosas en común y eso suele ser una alarma de que la jerarquía de clases no es correcta. Lo que nos está pasando es que sabemos que los Procesos, Estados y Transiciones tienen algo en común, pero todavía no volcamos ese conocimiento en el ambiente. Para hacerlo vamos a crear una nueva clase llamada `Instancia`.

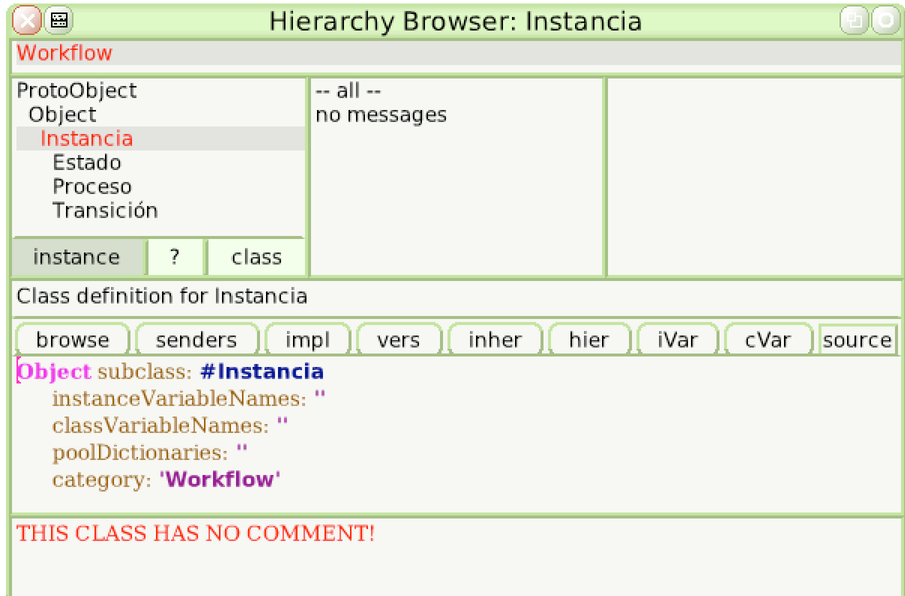


Modificamos las clases `Proceso`, `Estado` y `Transición` para que ahora sean subclases de `Instancia` (en lugar de ser subclases de `Object`). Para modificar la superclase usamos el Browser de Clases y en la definición de la clase (vemos la definición cuando tenemos seleccionada la clase pero no tenemos seleccionados ni categoría de métodos ni métodos) reemplazamos donde dice `Object` por `Instancia`, y aceptamos.



Repetimos el mismo proceso para las clases `Proceso` y `Transición`.

Ahora seleccionamos la clase `Instancia`, en el Browser, y presionamos el botón `hier` (hierarchy – jerarquía). Con eso abrimos el Browser Jerárquico.

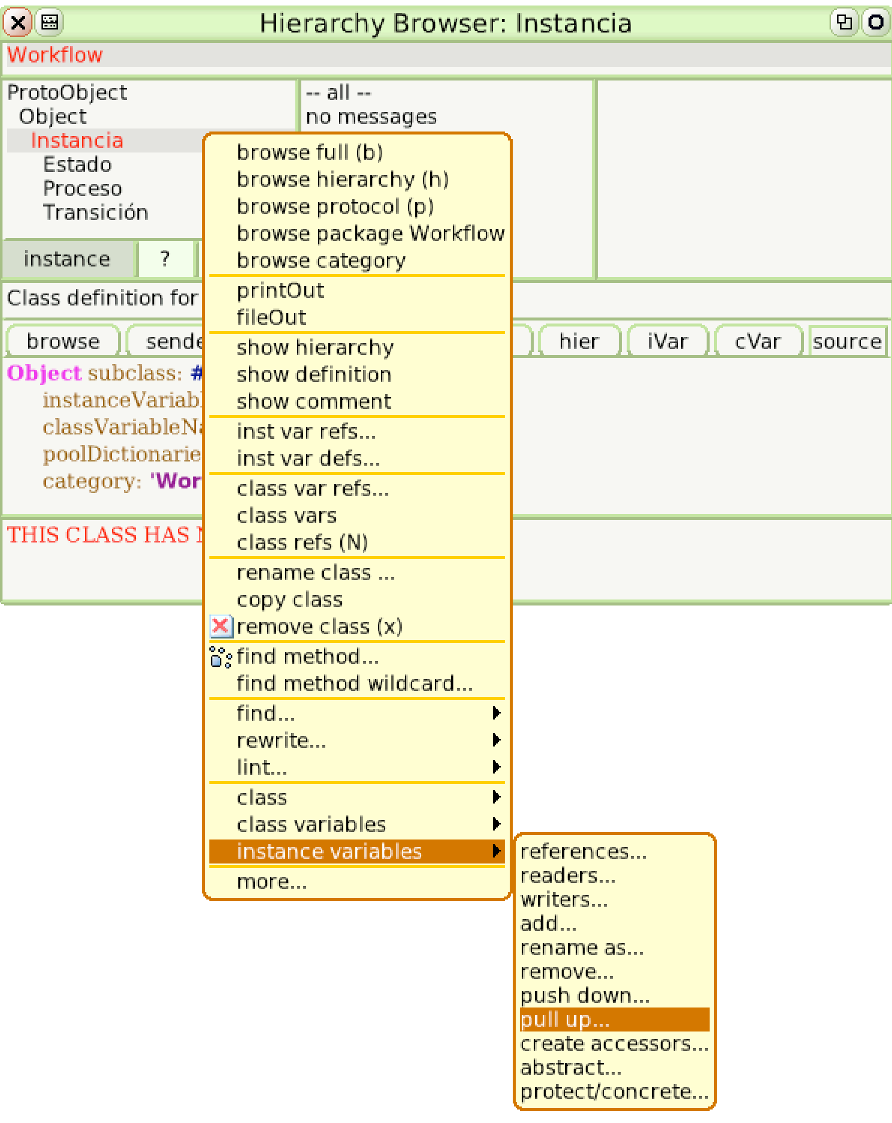


Browser Jerárquico

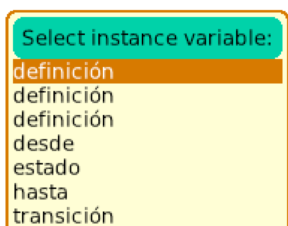
El Browser Jerárquico ofrece un formato diferente a la hora de presentar las clases. En lugar de presentar 2 paneles con las categorías de clases y las clases, este Browser muestra sólo un panel con un árbol que indica la relación superclase-subclase.

Con este Browser confirmamos, visualmente, que hicimos los cambios de superclase para las 3 clases.

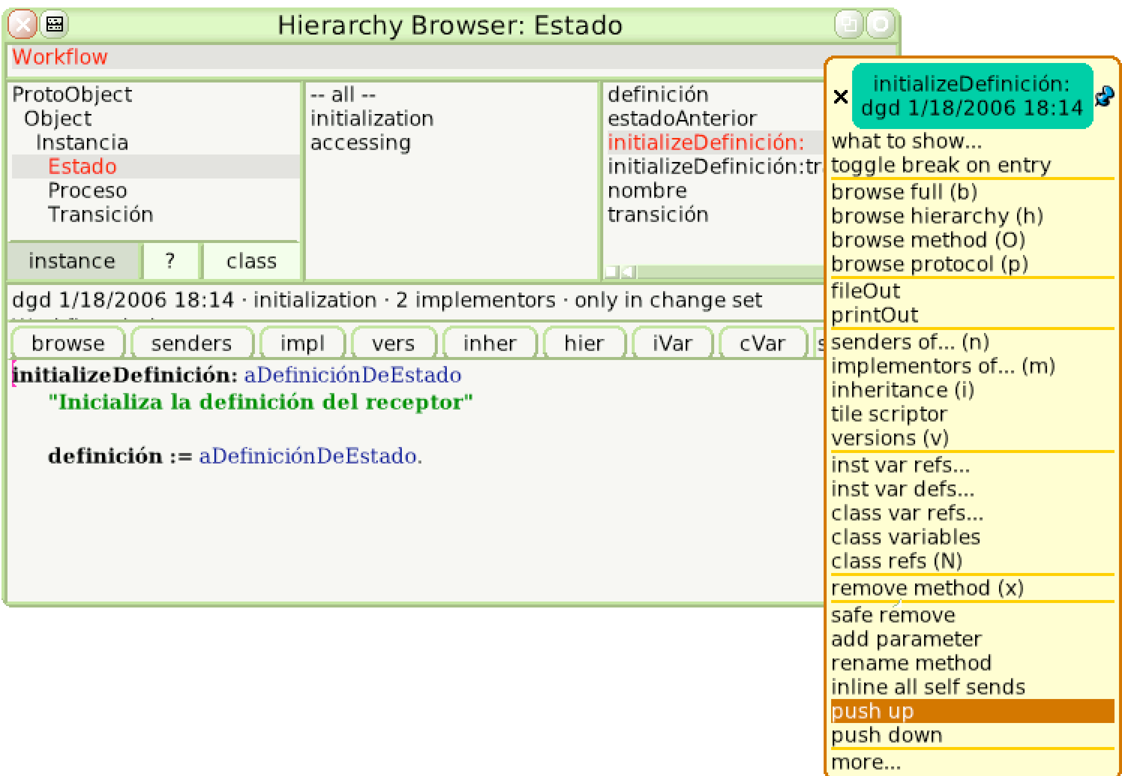
Entre las cosas que tienen en común las 3 clases encontramos la variable de instancia `definición`. Usamos otra funcionalidad del Refactoring Browser para “subir” la variable desde las subclases a la clase `Instancia`. Seleccionamos la clase `Instancia`, pedimos el menú contextual y seleccionamos la opción `'pull up...'` del submenú `'instance variables'`.



Seleccionamos una de las variables `definición`.



Ahora buscamos el método `Estado>>initializeDefinición:` y, desde el menú contextual del método, seleccionamos la opción `'push up'`.



Repetimos la operación para el método `Estado>>nombre` y decimos que `'yes'` (sí) cuando el Refactoring Browser nos pregunta `'Do you want to remove duplicate subclass methods?'` (¿Quiere remover los métodos duplicados en las subclases?). Al contestar afirmativamente esta pregunta, el Refactoring Browser borrará cualquier implementación igual a la subida a la clase `Instancia` en sus subclases.

Repetimos, nuevamente, la operación para el método `Estado>>definición`.

Y ahora hacemos lo mismo para el método `Estado class>>definición:`. y, nuevamente, contestamos afirmativamente a la pregunta de remover métodos duplicados.

Ahora cambiamos el método `Proceso>>initializeDefinición:` de la siguiente forma.

```
initializeDefinición: aDefiniciónDeProceso
    "Inicializa la definición del receptor"

    super initializeDefinición: aDefiniciónDeProceso.

    self cambiarEstado: definición estadoInicial nuevaInstancia.
```

Ahora modificamos el constructor `Estado class>>definición:transición:`

```
definición: aDefiniciónDeEstado transición: aTransición
    "Crea una nueva instancia del receptor"

    ^ (self definición: aDefiniciónDeEstado) initializeTransición: aTransición
```

Y creamos el método `Estado>>initializeTransición:`, en la categoría `'initialization'`.

```
initializeTransición: aTransición
    "Inicializa la transición del receptor"

    transición := aTransición
```

Usar el fuente de un método para crear otro método parecido

Cuando necesitamos crear un método que es similar a uno ya existente podemos modificar el método viejo a modo de plantilla, modificando su nombre, y aceptando.

Cuando hacemos eso el Browser crea otro método y deja el anterior intacto.

Y borramos el método `Estado>>initializeDefinición:transición:`

Ahora modificamos el constructor `Transición class>>definición:desde:`

```
definición: aDefiniciónDeTransición desde: anEstado
    "Crea una nueva instancia del receptor"

    ^ (self definición: aDefiniciónDeTransición) initializeDesde: anEstado
```

Creamos el método `Transición>>initializeDesde:`

```
initializeDesde: anEstado
    "Inicializa el 'desde' del receptor"

    desde := anEstado
```

Y el método `Transición>>initializeDefinición:`

```
initializeDefinición: aDefiniciónDeTransición
   "Inicializa la definición del receptor"

   super initializeDefinición: aDefiniciónDeTransición.

   hasta := definición hasta nuevaInstanciaTransición: self.
```

Y borramos el método `Transición>>initializeDefinición:desde:`

Ejecutamos todos los tests y vemos que seguimos en verde. Sigamos limpiando.

Las 2 especializaciones del método `Instancia>>initializeDefinición:` (en las clases `Proceso` y `Transición`) son para hacer algo con la definición apenas la instancia la conoce. Modificamos el método `Instancia>>initializeDefinición:` de la siguiente forma.

```
initializeDefinición: aDefiniciónDeEstado
   "Inicializa la definición del receptor"

   definición := aDefiniciónDeEstado.

   self definiciónCambiada.
```

E implementamos el método `Instancia>>definiciónCambiada` así:

```
definiciónCambiada
   "La definición ha cambiado en el receptor"
```

Ahora, usando la función "Extract Method", extraemos las expresiones que están a continuación de la llamada a `super`, a nuevos métodos de nombre `#definiciónCambiada`.

```
definiciónCambiada
   "La definición ha cambiado en el receptor"

   self cambiarEstado: definición estadoInicial nuevaInstancia
```

```
definiciónCambiada
   "La definición ha cambiado en el receptor"

   hasta := definición hasta nuevaInstanciaTransición: self
```

Y borramos los métodos `Proceso>>initializeDefinición:` y `Transición>>initializeDefinición:`

Ejecutamos los tests y seguimos en verde.

Ahora nos tocaría refactorizar la jerarquía de Definición. Crearíamos una clase de nombre `Definición`, modificaríamos la superclase de `DefiniciónDeProceso`, `DefiniciónDeEstado` y `DefiniciónDeTransición`. Subiríamos la variable de instancia `nombre`, los métodos de instancia `#nombre` y `#initializeNombre:`, el método de clase `#nombre:`, etc. Al crear la jerarquía nos hubiésemos dado cuenta que nos olvidamos de ponerle nombre a las DefiniciónDeProceso y, al colgarla de la clase `Definición`, la heredaría.

Queda para el lector la tarea de completar la factorización de la jerarquía `Definición`.

Lo último que nos queda hacer es crear la clase `Gestor`, y hacer que `GestorConBloque` sea una subclase.

Creamos la clase e implementamos el método `#estadoCambiadoEn:`

```
estadoCambiadoEn: aProceso
   "Un proceso ha cambiado su estado actual.

   Hace lo que se necesite por el cambio de estado, luego se responde el nombre
de la transición a tomar para pasar al próximo estado, o nil cuando el estado sea
el final"

   ^ self subclassResponsibility
```

Mensaje #subclassResponsibility

El mensaje #subclassResponsibility es la forma de indicar que ese método debe ser implementado en las subclases. Es la forma de marcar un método como abstracto.

Con esto damos por concluido el ejemplo del motor de Workflow. Se puede descargar el código

completo del ejemplo desde el sitio web del libro (http://smalltalk.consultar.com) aunque lo que realmente recomiendo es que se haga el ejemplo, paso a paso, siguiendo las instrucciones del libro.

La yapa

YAPA: *Sust. Regalo que el vendedor hace al comprador. (De....): además, por añadidura.*

En este capítulo hablaremos, sólo un poco, de algunas funcionalidades y características que Smalltalk posee pero que, por el enfoque y el tamaño del libro, no seremos capaces de desarrollar en su plenitud. No oculto que el objetivo principal de este capítulo es despertar la curiosidad de los lectores y que, esa curiosidad, sirva de motor para que continúen en el proceso de aprender Smalltalk.

Mensaje #become:

El mensaje `#become:`, presente en prácticamente todos los dialectos, permite intercambiar un objeto (el receptor) por otro (el argumento) en toda la imagen. Los objetos que referenciaban al receptor, después del envío del mensajes, referenciarán al objeto dado como argumento.

Este mensaje es muy poderoso y, a su vez, peligroso. Como sucede con todas las herramientas poderosas, existen algunos casos donde su uso es la mejor solución y existen muchos más casos donde su uso sería un error.

Uno de los casos donde suele usarse con éxito es cuando queremos reemplazar una instancia “falsa” por la real; es frecuente tener que entregar instancias incompletas o, simplemente, entregar un falso objeto que, al usarse, instancie el objeto real y se intercambie por el (un lazy-instantiation).

Mensaje #doesNotUnderstand:

Cuando el receptor no entiende un mensaje, la máquina virtual le da otra oportunidad para reaccionar enviándole el mensaje `#doesNotUnderstand:` con un objeto de clase `Message` como argumento, donde se almacena toda la información del mensaje fallido.

De igual forma que el caso anterior, el mensaje `#doesNotUnderstand:` es, a la vez, poderoso y peligroso. Y, también igual que en el caso anterior, existen algunos casos donde su uso es la mejor opción. Tal vez el ejemplo más representativo es el de hacer proxies que deleguen todos los mensajes que reciben a un objeto que conozcan. Combinado con el mensaje anterior se puede hacer que en el primer mensaje no entendido por el proxy, este instancie (o localice) el objeto real y se intercambie con el usando el `#become:`.

Mensajes #perform:, #perform:with:, #perform:withAll:, etc.

Estos mensajes permiten enviar un mensaje por nombre. El SUnit, por ejemplo, lo utiliza para enviar al objeto de clase `TestCase` un mensaje por cada uno de los métodos que comienzan por `#test`.

Pseudo-variable thisContext

La pseudo-variable `thisContext` permite acceder al objeto que representa el contexto de activación del método. Con esta variable se puede hacer, entre otras cosas, a los valores de las variables locales, al emisor del mensaje, etc. El depurador hace un uso intensivo de esta pseudo variable. También la utiliza el Seaside para crear los continuations.

SLang, máquina virtual y plugins

El SLang es un subconjunto de la sintaxis y funcionalidad del lenguaje Smalltalk que permite generar código en lenguaje C con la misma funcionalidad. La máquina virtual está escrita en SLang. Esta funcionalidad también permite escribir "plugins" que son fragmentos de código Smalltalk que, al convertirlos a C y compilarlos, se ejecutan a mayor velocidad. El SLang, al ser un subconjunto, permite que la depuración se haga con la ricas herramientas de Smalltalk.

FFI – Foreign Function Interface

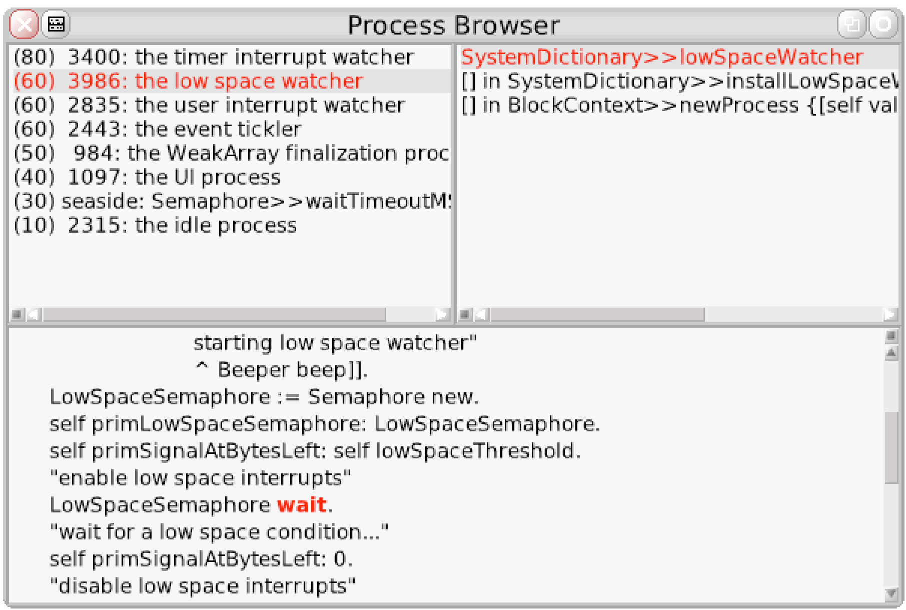
El FFI es un conjunto de objetos que permiten llamar a funciones que residen en librerías compartidas. Por ejemplo, el paquete ODBC hace uso intensivo de esta funcionalidad para acceder a las funciones de la DLL (o el .so de linux) de ODBC.

Metaprogramación

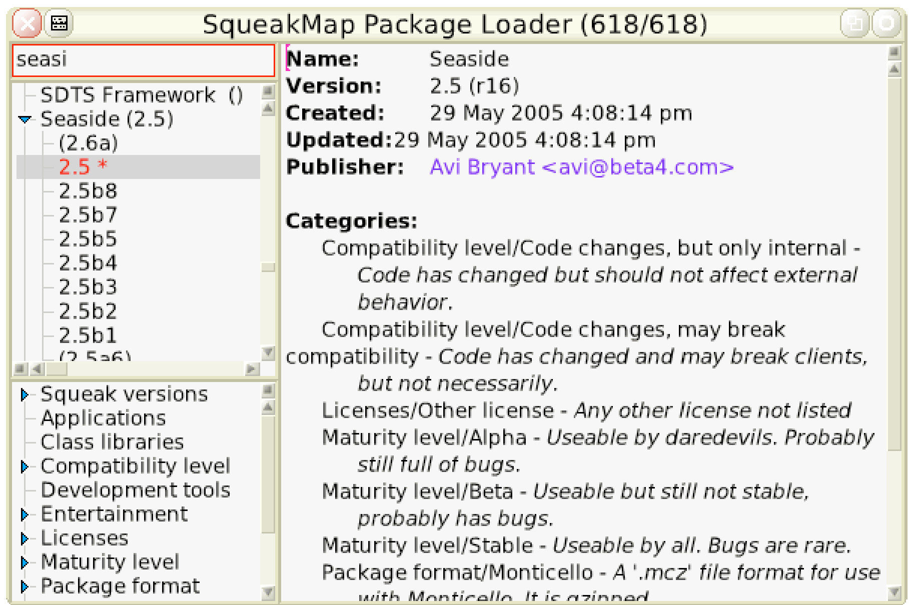
En Smalltalk las clases son objetos y se las puede manipular como se hace con cualquier objeto: enviando mensaje. Disponemos de mensajes para indagar que métodos tiene la clase, para conocer sus superclases, para ver todas las instancias, etc. Pero también podemos cambiar la estructura de las clases usando mensajes. Eso quiere decir, entre otras cosas, que podemos crear programar que creen programas (metaprogramas).

Multithreading

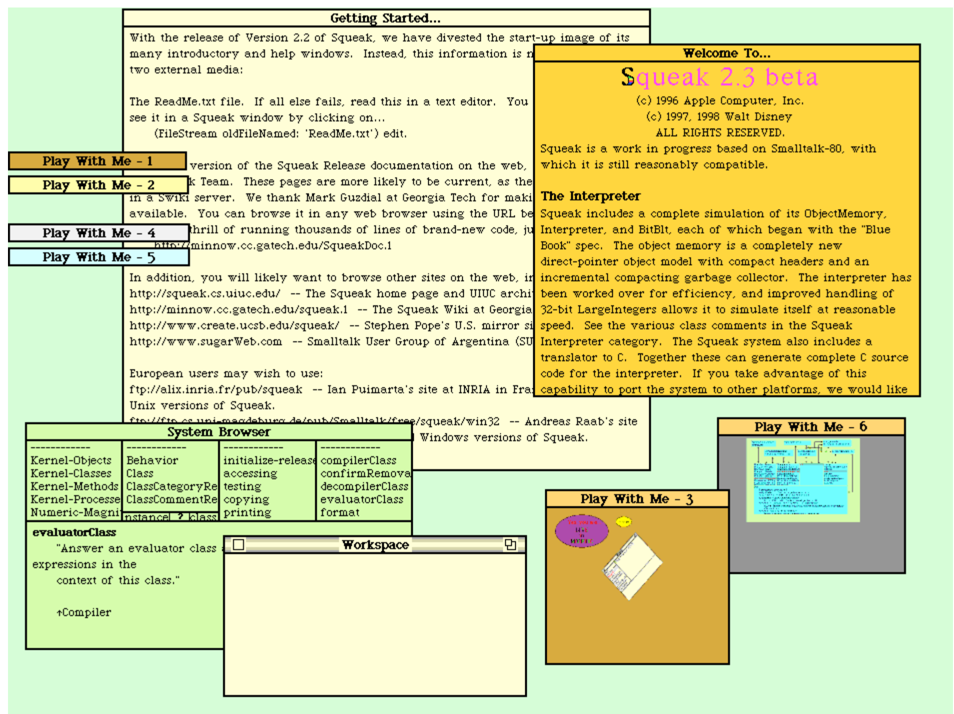
Smalltalk permite la programación con procesos (el nombre que Smalltalk le da a sus threads) desde sus primeras versiones. Contamos con objetos para sincronizar el trabajo entre diferentes procesos como semáforos, colecciones thread-safe, etc.



SqueakMap
SqueakMap es un repositorio de proyectos Squeak. Se pueden descargar e instalar, a la fecha, más de 600 aplicaciones.

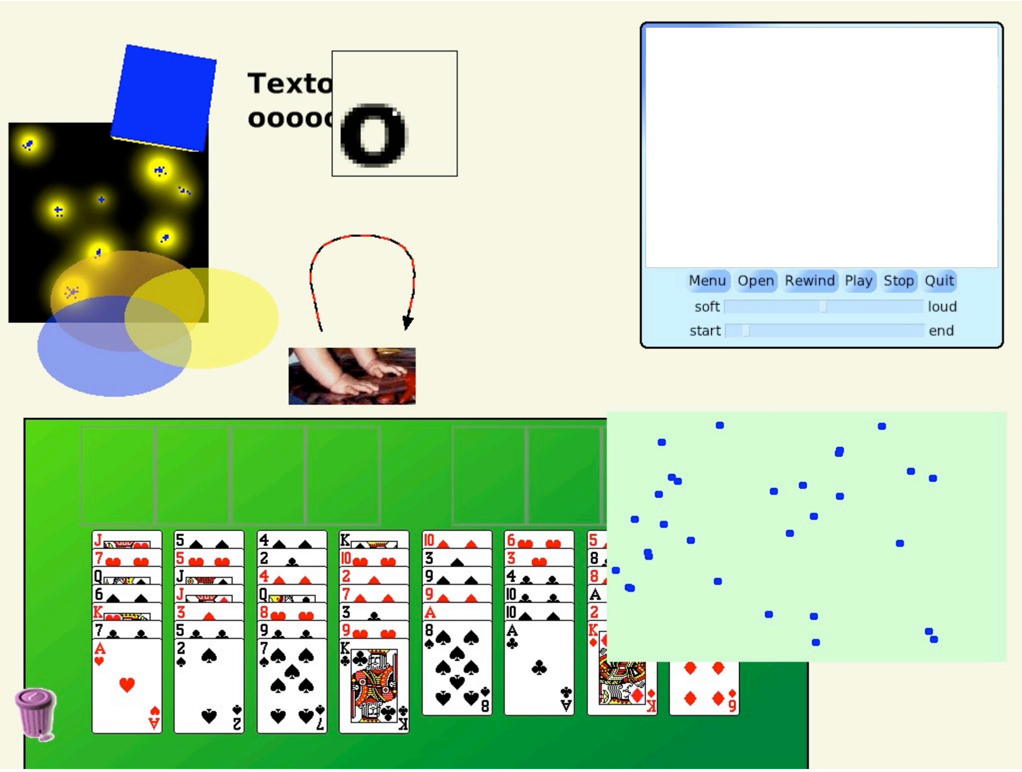


MVC
MVC es el nombre que recibe el tradicional framework para hacer aplicaciones gráficas que venía con el Smalltalk/80. Ese framework recibe el mismo nombre que el patrón de diseño ya que es parte fundamental de su arquitectura. Este es un framework muy liviano que puede ser utilizado para desarrollo de aplicaciones donde el consumo de memoria sea una limitante.

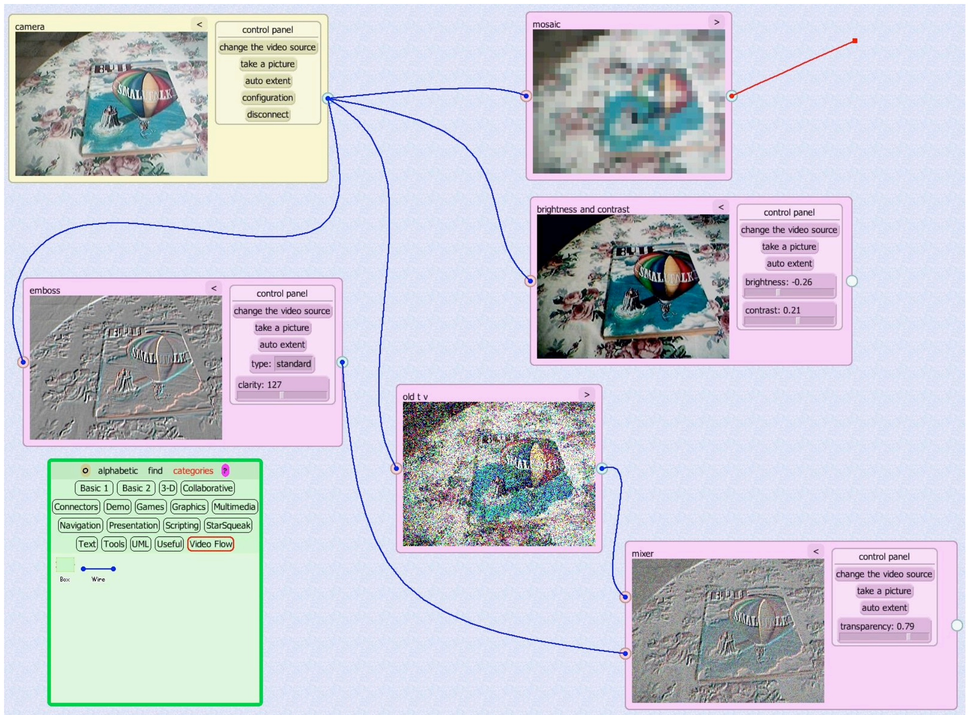


Morphic

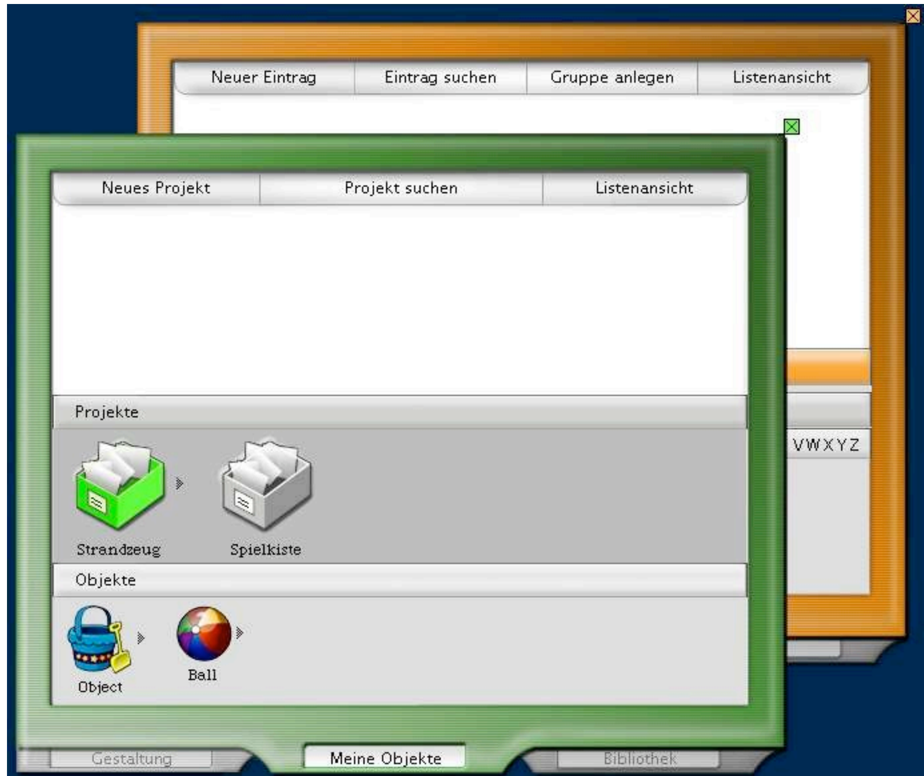
En Squeak existe otro framework para la construcciones de interfaces gráficas de usuario. El Morphic permite una manipulación más concreta de los objetos representados en la pantalla. Los conceptos básicos de Morphic viene del proyecto Self (http://research.sun.com/self/).


Texto
ooooo
O
Menu Open Rewind Play Stop Quit
soft
loud
start
end

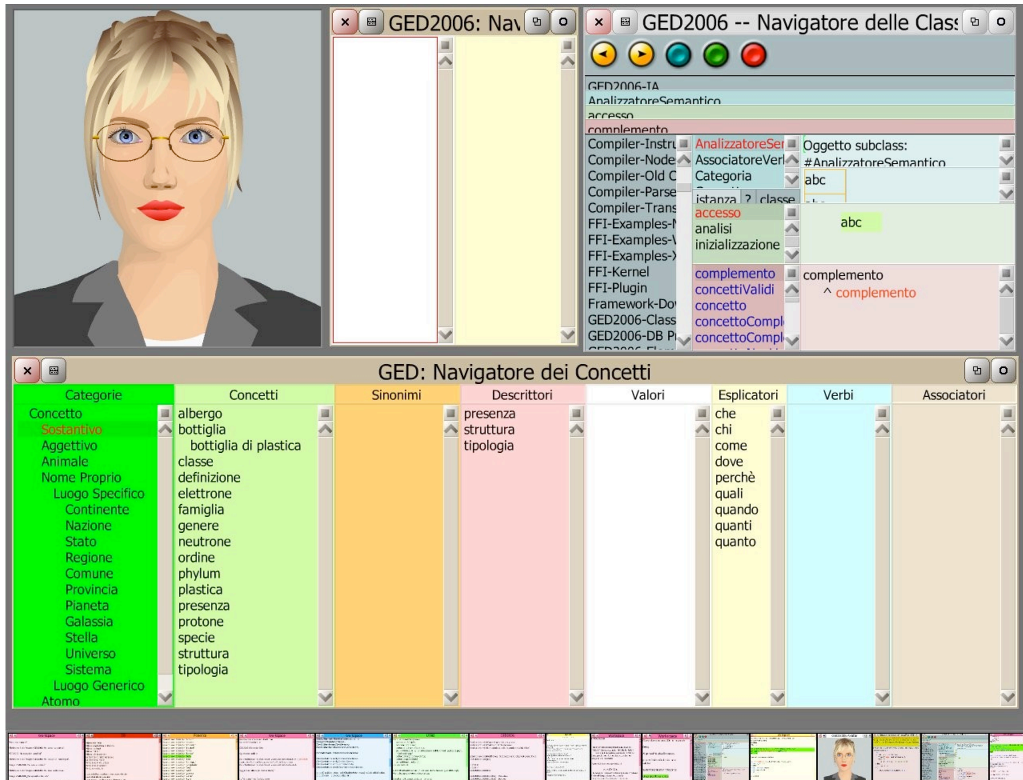
Algunos proyectos con Squeak



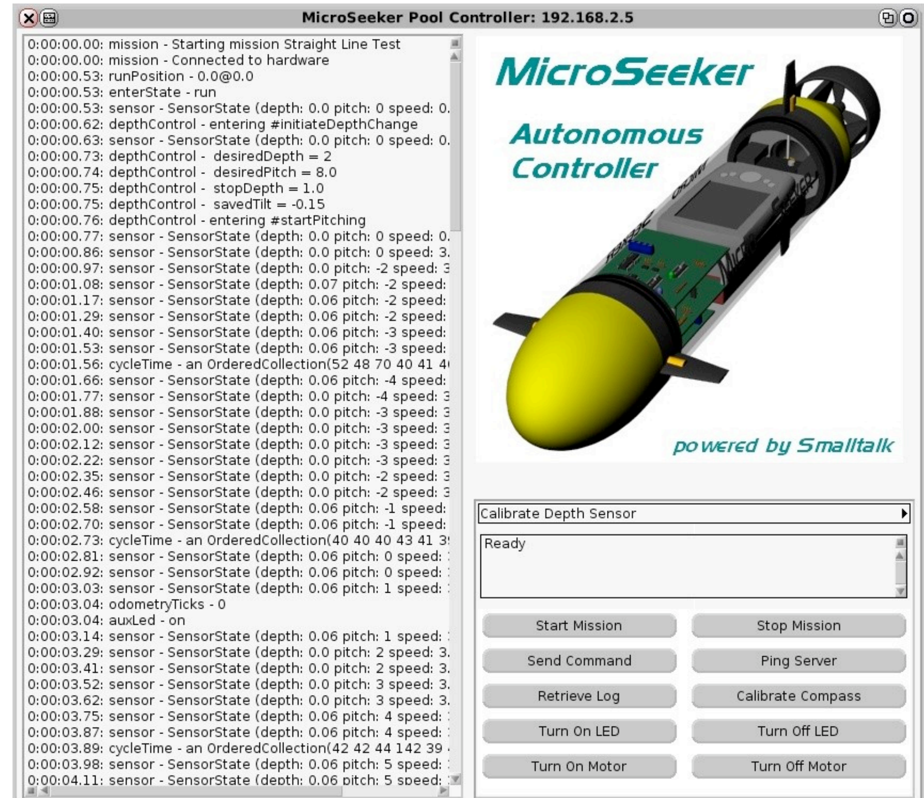
VideoFlow: Procesamiento de vídeo



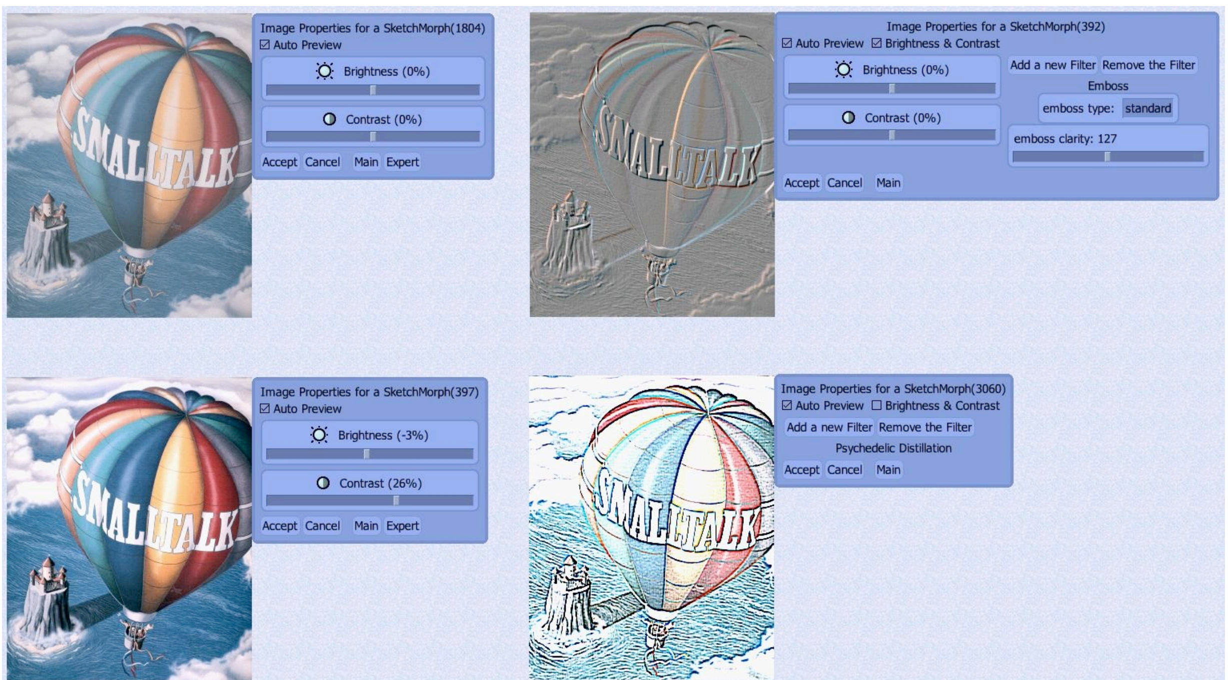
Proyecto Magrathea



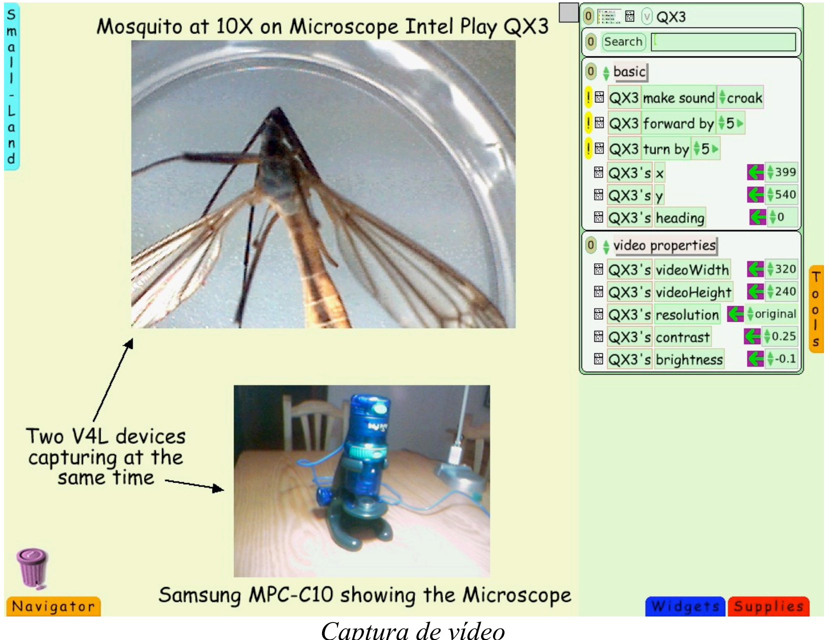
Proyecto E-Alterego



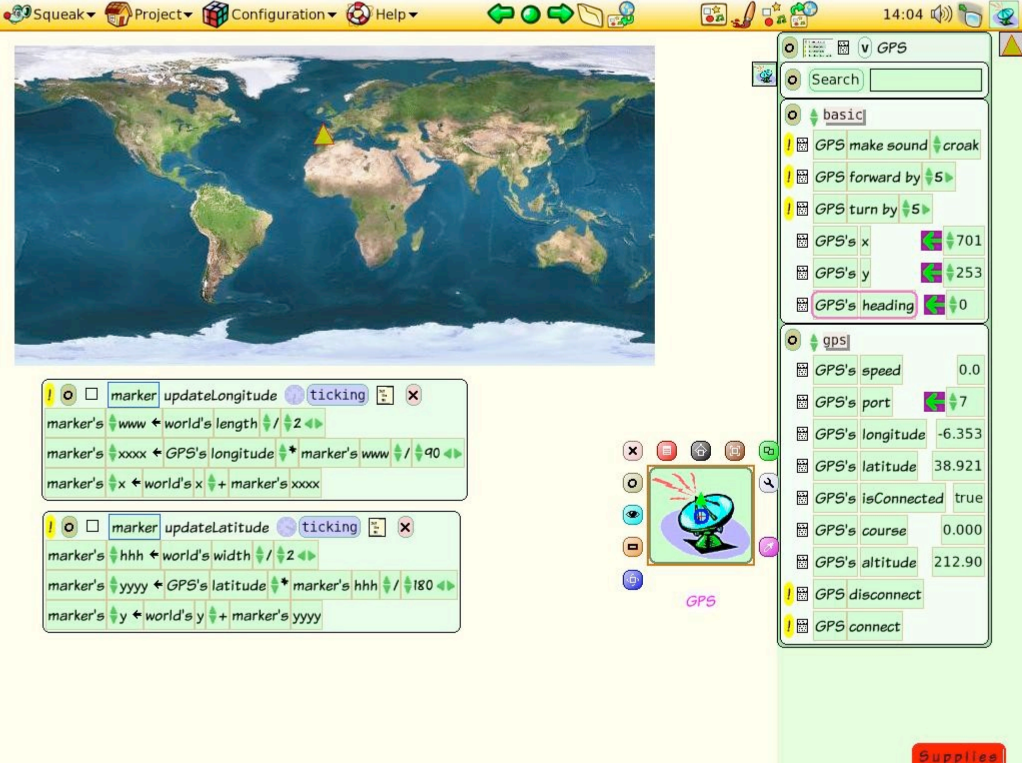
Control de Robots



Procesamiento Fotográfico



Captura de vídeo



Leyendo un GPS

Futuro

El futuro de Smalltalk es servir como herramienta para desarrollar el entorno de computación que vuelva obsoleto al mismo Smalltalk. Es un objetivo ambicioso y el desarrollo se está llevando a cabo en diferentes áreas. A continuación enumero algunos de los proyectos actuales:

Traits

Smalltalk, desde los inicios, optó por un sistema de reutilización basado en herencia simple para, sobre todo, evitar las complicaciones que conlleva un sistema de herencia múltiple como, por ejemplo, el de C++. Traits es un proyecto para ampliar el poder de expresión de un sistema de herencia simple, pero sin caer en las complicaciones de un sistema de herencia múltiple. La próxima versión de Squeak, la 3.9, muy probablemente incluirá una primera versión usable en entornos de producción. Para más información: http://www.iam.unibe.ch/~scg/Research/Traits/

Tweak

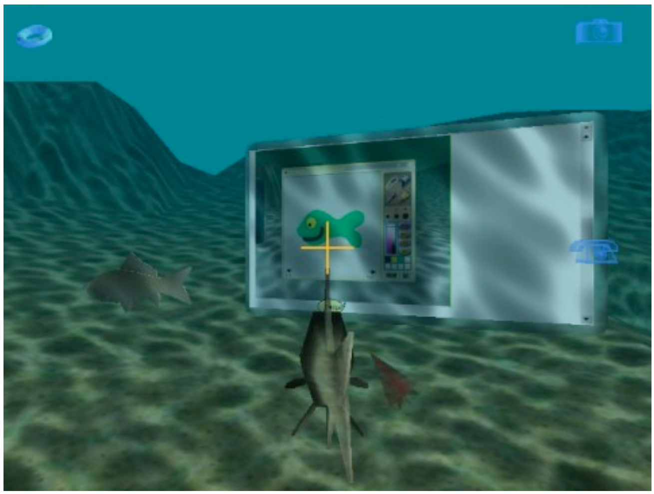
El proyecto Smalltalk fue, desde el principio, uno de los motores que empujó el diseño de interfaces de usuario. Tweak es un nuevo framework que pretende mover la facilidad de la programación tipo eToys a un entorno con la potencia completa de Smalltalk. Para más información: http://tweak.impara.de/

64 bits

Las primeras máquinas virtuales de Smalltalk eran de 16 bits y permitían un muy limitado direccionamiento de objetos. La primera implementación de Squeak ya era de 32 bits, aumentando tremendamente la capacidad de direccionamiento. Sin embargo, para que Smalltalk siga liderando la evolución de entornos de computación es necesario pasar a una arquitectura de objetos de 64 bits. Para más información: http://www.squeakvm.org/squeak64/

OpenCroquet

¿Qué pasaría si desarrollásemos un sistema operativo, hoy en día, sabiendo lo que sabemos hoy en día? Esta es la pregunta que rige el desarrollo de OpenCroquet, un sistema de objetos distribuido, 100% colaborativo, con una rica interfaz de usuario de objetos 3D. Para más información: http://www.opencroquet.org



Como continuar

Sin duda lo más importante para aprender Smalltalk es meter mano en un Smalltalk. No vale de nada leer cientos de libros si no nos metemos de lleno en un ambiente de Smalltalk y comenzamos a hacer cosas.

Pero también es cierto que la experiencia puede ser más fácil si logramos percibir algo de la filosofía del desarrollo con Smalltalk desde algunos libros. A mi, particularmente, me impactaron especialmente las siguientes lecturas:

Libros

- **Smalltalk-80 - The Language:** En este libro se puede entender, de personas que participaron en el desarrollo de Smalltalk, la filosofía del lenguaje y las motivaciones que lo hicieron posible.
- **Smalltalk-80 - Bits of History, Words of Advice:** Este libro me impactó por lo actual de los comentarios hechos por personas que probaron Smalltalk a principios de los años 80.
- **Smalltalk Best Practice Patterns:** Este libro me enseño que una de las cosas más importantes, para ser un buen programador Smalltalk, es prestar muchísima atención a la claridad del código.
- **Smalltalk with Style:** Este libro me mostró la mayoría de las convenciones que se utilizan a la hora de escribir código Smalltalk.
- **Design Patterns - Elements of Reusable Object Oriented Software:** No es posible decir hoy en día que se sabe programar con objetos si no se ha leído este libro. Este libro es el culpable de que hoy esté escribiendo un libro de Smalltalk y no uno de Java o .NET. Me enteré de la existencia de Smalltalk, y de su impacto sobre la tecnología de objetos, a través de las explicaciones de algunos patrones. Este es el libro que despertó mi curiosidad por aprender Smalltalk.
- **The Dessign Patterns Smalltalk Companion:** Este libro me enseñó mucho Smalltalk por comparación con la implementación de los patrones en el libro anterior.

Papers o artículos

- **Design Principles Behind Smalltalk:** Excelente descripción de las motivaciones que rigen el desarrollo del proyecto Smalltalk. Parece ciencia ficción si consideramos que la nota fue publicada en la revista Byte de agosto de 1981.
- **Personal Dynamic Media:** Excelente explicación de las motivaciones detrás del desarrollo de Smalltalk escrita por Alan Kay en el año 1976.
- **The Early History of Smalltalk:** Esta es una historia de Smalltalk, escrita por Alan Kay, en el año 1993 (antes de que comenzara el proyecto Squeak allá por el 1995) donde cuenta de donde vienen las ideas principales. Termina con una frase que también me marcó especialmente: "¿Donde están los Dan (Ingalls) y las Adele (Goldberg) de los 80s y los 90s que nos lleven al próximo nivel de desarrollo?".

Grupos de Usuarios

Existen diferentes grupos de usuarios y listas de discusión en Internet. Las comunidades de usuarios de Smalltalk suelen ser muy amables, a diferencia de otras comunidades, contestando a preguntas de personas que recién comienzan.

Conclusión

Este libro no pretendía ser un manual de Smalltalk, ni tampoco una guía de referencia. Este libro sólo pretendía mostrar un poco la filosofía detrás de Smalltalk y como esa filosofía impacta en el desarrollo de software y, a su vez, como esta forma de trabajar produce software de mayor calidad en menos tiempo que con otras opciones.

Elegí mostrar como, un ambiente del poder de Smalltalk, permite un estilo de programación muy incremental ya que el costo de los cambios es muy bajo. Sin embargo creo que es necesario aclarar que Smalltalk no requiere que se trabaje así y que es perfectamente posible utilizar un ambiente de Smalltalk con una forma de trabajo menos incremental y menos dinámica. Cada programador tiene que decidir, entre otras cosas, que tipo de programación prefiere. En este libro sólo intenté mostrar el tipo de programación que yo prefiero, y que no puedo desarrollar en otras herramientas que no sean Smalltalk.

A lo mejor queda en algún lector la idea que esta metodología no es muy rápida. Si eso les ha ocurrido consideren que tuve que mostrar, paso a paso, las herramientas y como se usan, también tuve que introducir notas con explicaciones sobre tal o cual clase, y eso podría haber impactado negativamente en la percepción del tiempo de desarrollo. Reto a cualquiera que se haya podido quedar con ese sabor de boca a probar la metodología en algún sistema real y que extraigan las conclusiones sólo después de esa experiencia.

Smalltalk es una pieza increíble de software, sin embargo Smalltalk no es la meta en si mismo. Smalltalk es parte de una visión sobre como debiéramos utilizar las computadoras.

Los objetivos planteados en las ideas que rigen el desarrollo de Smalltalk todavía no han sido alcanzados en su plenitud. Smalltalk nos permite, hoy, imaginar como podría ser la experiencia de utilizar una computadora y, para alcanzar el objetivo máximo – que cualquier persona , y no sólo programadores, pueda interactuar con la información y construir conocimiento a través de esa interacción – debemos seguir mejorando al Smalltalk.

Para lograr ese objetivo los programadores tenemos una tarea: Convertirnos en mejores programadores y compartir con todo el mundo – programadores y no programadores – lo que aprendimos.

Bibliografía

Este libro, como ya se aclaró en el capítulo de introducción, no es una guía completa de Smalltalk. Los siguientes libros y papers son complementos ideales para quien decida continuar su desarrollo como programador aprendiendo Smalltalk.

Algunos de ellos se pueden descargar, de forma gratuita, de http://www.iam.unibe.ch/~ducasse/FreeBooks.html

A Little Smalltalk
Budd, Tim
Addison Wesley. 1987.

Back to the future - The Story of Squeak, A Practical Smalltalk Written in Itself
Kay, Alan
http://users.ipa.net/~dwighth/squeak/oopsla_squeak.html
OOPSLA 97 Proceeding. P. 318-326. ACM Press, New York, 1997.
(a postscript by Dan Ingalls in [Guzdial/Rose 2001]).

Design Patterns - Elements of Reusable Object Oriented Software
Gamma, E., R. Helm, R. Johnson, and J. Vlissides
Addison Wesley. 1995.

Design Principles Behind Smalltalk
Ingalls, Dan
http://users.ipa.net/~dwighth/smalltalk/byte_aug81/design_principles_behind_smalltalk.html

eXtreme Programming eXplained - Embrace Change
Beck, Kent
Addison Wesley. 1999.

Guide to Better Smalltalk. A Sorted Collection
Beck, Kent
Cambridge University Press en asociación con SIGS Books. United State of America. 1999.

Inside Smalltalk (vol. one and two)
LaLonde, Wilf. R. Pugh, John. R.
Prentice Hall. 1990.

Object-Oriented Implementation of Numerical Methods. An Introduction with Java and Smalltalk
Besset, Didier H.
Morgan Kaufmann Publishers. United State of America. 2001.

Personal Dynamic Media

Kay, Alan. Goldberg, Adele
http://www.dolphinharbor.org/dh/smalltalk/documents/
Reprinted in The New Reader. 1976.

Powerful Ideas in the Classroom
B. J. Allen Conn. Rose, Kim
Viewpoints Research Institute, Inc. 2003.

Refactoring Browser Tool
Brant, Hohn, and Don Robert.
http://st-www.cs.uiuc.edu/~droberts/tapos/TAPOS.htm

Refactoring. Improving The Design of Existing Code
Fowler, Martin
Addison Wesley. Massachusetts. 1999.

Simple Smalltalk Testing - With Patterns
Beck, Kent
http://www.xprogramming.com/testfram.htm

Smalltalk An Introduction To Aplication Development Using visualwork
Hopkins, Trevor and Horan, Bernard
Pearson Education. 1995.

Smalltalk and Object Orientation - An Introduction
Hunt, John.
Springer-Verlag. 1997.

Smalltalk Best Practice Patterns
Beck, Kent
Pretice Hall, New Jersey. 1997.

Smalltalk by Example - The Developer's Guide
Sharp, Alex
McGraw Hill Text. 1997.

Smalltalk with Style
Skublics, Suzanne. Klimas, Edward J. Thomas, David A.
Pretice Hall. New Jersey. 1996.

Smalltalk-80 - The Language
Goldberg, Adele. Robson, David
Addison-Wesley Proffesional. 1989.

Smalltalk-80 - Bits of History, Words of Advice
Krasner, Glenn
Addison Wesley. California. 1983.

Smalltalk-80 - The Interactive Programming Enviroment
Goldberg, Adele
Addison Wesley. California. 1984.

Smalltalk-80 - The Language and Its Implementation
Goldberg, Adele. Robson, David
Addison Wesley. California. 1983.

Smalltalk, Object and Design
Chamond, Liu
To Excel. New York. 1996.

Squeak - Object-Oriented Design whit Multimedia Applications
Guzdial, Mark
Prentice Hall. New Jersey. 2001.

Squeak. Open Personal Computing and Multimedia
Guzdial, Mark. Rose, Kim
Prentice Hall. New Jersey. 2002.

Test-Driven Development - By Example
Beck, Kent
Adisson Wesley. 2002.

The Art and Science of Smalltalk
Lewis, Simon
Prentice-Hall. 1995-1999.

The Dessign Patterns Smalltalk Companion
Alpert, Sherman. Brown, Kyle. Woolf, Bobby
Addison Wesley. California. 1998.

The Early History of Smalltalk
Kay, Alan
http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

The Joy of Smalltalk
Tomek, Ivan. 2000.

The Taste of Smalltalk
Kaehler, Ted. Patterson, Dave
WW Norton & Co. 1986.

Tools For Thought. The History and Future of Mind-Expanding Technology
Rheingold, Howard
The MIT Press Cambridge, Massachusetts. 2000.
http://www.rheingold.com/texts/tft/

Una Explicación de la Programación Extrema - Aceptar el cambio
Beck, Kent
Adisson-Wesley Iberoamericana España, S.A. Madrid. 2002.

Herramientas usadas en el libro

Para mostrar la forma de trabajo que permite el uso de un ambiente de objetos como Smalltalk, en este libro, elegí desarrollar algunas aplicaciones de ejemplo paso a paso, confiando en que el lector perciba la diferencia de trabajar con Smalltalk de una experiencia de primera mano. Este libro pierde por completo su sentido si no logro que los lectores se animen a probar, en un ambiente de Smalltalk, cuanto se lee y todo lo que se venga en mente.

Para reducir al mínimo el tiempo necesario para entrar en el ambiente, cree una imagen de Squeak pre-configurada y con algunos paquetes extras instalados.

La imagen utilizada es una 3.8-full "oficial" con varias preferencias cambiadas, y con los siguientes paquetes instalados:

Nombre	Versión	Descripción
YAXO	2.2	Parser de XML.
XML Stack Parser	1	Parser de XML basado en una pila.
Copy as HTML	1	Utilidad para copiar al portapapeles una versión HTML del texto Squeak. Incluye el formato con sus colores, etc. Usado para el código mostrado en este libro.
StringEnh	1	Algunos extras para la clase `String`.
Shout	4	Paquete que hace syntax-highlight mientras se escribe.
ShoutMonticello	2	Extensiones Shout para las herramientas relacionadas con el Monticello.
ShoutWorkspace	2	Un Workspace que, valiéndose del Shout, hace syntax-highlight mientras se escribe.
Refactoring Browser for 3.8	3.8.43	Extensiones para los browsers para hacer refactorizaciones del código.
SUnit	3.1.22	Framework para hacer UnitTesting con Smalltalk. Derivado directo del primer framework de UnitTesting escrito por Kent Beck.
DynamicBindings	1.2	Paquete requerido por KomHttpServer.
KomServices	1.1.1	Paquete requerido por KomHttpServer.
KomHttpServer	7.0.2	Paquete requerido por Seaside. Servidor web íntegramente desarrollado en Smalltalk.
Seaside	2.5	Framework de desarrollo de aplicaciones web basado en continuations. Para más información ver

Nombre	Versión	Descripción
		el sitio www.seaside.st

La imagen oficial se puede descargar desde el sitio www.squeak.org específicamente:

http://ftp.squeak.org/current_stable/Squeak3.8-6665-full.zip.

Para construir una imagen, como la usada en este libro, lo más fácil es instalar el paquete “ProgramacionConSmalltalk”, usando el SqueakMap, y este instalará las herramientas necesarias descargándolas desde Internet.