Tesis de Licenciatura

“Conjuntos con Semántica”

Un framework de conjuntos por comprensión en ambientes de objetos

Tesistas: Daniel Altman - Hernán Tylim

Director: Dan Rozenfarb

Marzo / 2007

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

daniel.altman@gmail.com - htylim@ciudad.com.ar
drozenfa@yahoo.com.ar

Resumen

Al revisar los principales lenguajes y ambientes de programación orientada a objetos, llegamos a la conclusión de que los modelos de conjuntos utilizados actualmente no brindan la funcionalidad que esperamos, principalmente porque los conjuntos están modelados como simples enumeraciones de objetos.

En la presente tesis desarrollamos un framework de conjuntos al que llamamos *SSets*. SSets permite definir los conjuntos de manera declarativa, haciendo que la semántica de los conjuntos quede explícita en el modelo, lo que acarrea muchas ventajas. Un ejemplo es la posibilidad de representar conjuntos infinitos. Otro, es la simplificación del diseño de las aplicaciones, al liberar a estas de la responsabilidad de actualizar el contenido de sus conjuntos de forma manual.

SSets provee también una serie de herramientas gráficas que permiten manipular los conjuntos y operar con ellos de manera interactiva. Esto contribuye a simplificar la programación exploratoria. Las herramientas, sumadas a que la semántica de los conjuntos queda ahora representada dentro de los conjuntos mismos, permiten comprender más fácilmente un diseño al inspeccionar una aplicación.

Finalmente, utilizando el framework remodelamos las herramientas *Senders Of*, *Implementors Of* y *Users Of* utilizadas en Smalltalk desde hace más de 30 años. El resultado fueron herramientas con mayor funcionalidad sin costo de implementación adicional. Esto gracias a las funcionalidades de conjuntos provistas en el framework.

Agradecimientos

Antes de comenzar con la tesis en sí misma, queremos tomarnos unas líneas para agradecer a alguna gente que estuvo apoyándonos de distintas maneras a lo largo de estos años, en nuestra carrera en general y en este trabajo en particular.

Dany

A mi esposa Cynthia, que llenó mi vida de proyectos, me incentivó constantemente para que avanzara con la tesis y se bancó innumerables horas conmigo sentado frente a la compu. ¡Te amo!

A mis padres, Alicia y Carlos. A mis hermanos y cuñados. A mi sobrina Cande. A mi suegro Roberto. A La Bobe. ¡Los quiero mucho!

A Dan Rozenfarb, que empezó como mi docente y terminó siendo mi director y amigo. Sin sus ideas y su paciencia a lo largo de estos años, esta tesis no hubiera sido posible.

A Hernán Tylim, con quien recorrimos este camino, apoyándonos mutuamente, compartiendo, criticando y aprendiendo juntos constantemente.

A Máximo Prieto, Dan Rozenfarb, Hernán Wilkinson y Carlos Ferro, que me iniciaron en el paradigma de la POO que tanto aportó a mi formación académica y profesional. Disfruté de tenerlos tanto de docentes como de compañeros.

A los amigos que hicieron de mi carrera algo más placentero, compartiendo conmigo desde noches de trabajos prácticos y exámenes, hasta asados y partidos de fútbol. Sé que son muchos, pero en representación voy a nombrar a Eze, a Chala, a Quique, y por supuesto, a Diego es Inocente, que tantas alegrías y campeonatos de fútbol me dio (¡y cuántos goles le di yo…!).

A mis otros amigos, esos que están siempre ahí: Fede, Sebas, Gusty, Ale W., Cebo, y todos los demás.

Hernán

A mi familia, amigos, compañeros y profesores, quienes sin su apoyo o guía este trabajo no hubiera sido posible.

Índice

Introducción

Falencias del modelo actual de conjuntos

En los lenguajes orientados a objetos (Smalltalk [SMA80], .Net [DOTNET], Self [UNG87], Java [JAVA], etc.) los conjuntos están planteados como simples enumeraciones de elementos. Un conjunto es sólo un contenedor de otros objetos [LIU, capítulo 6], que cumple con ciertas reglas como la no repetición de sus elementos y la carencia de un orden entre los mismos dado por el conjunto. Un ejemplo típico de cómo se crea un conjunto en estos ambientes, utilizando la sintaxis de Smalltalk, es:

```
setA := Set new.
setA add: 'a'.
setA add: 'b'.
setA add: 'c'.
```

Ejemplo: creación de un conjunto que contiene los elementos ‘a’, ‘b’ y ‘c’.

A pesar de que este modelo de conjuntos/contenedores ha sido y es muy utilizado en la actualidad, en muchos casos no es suficiente para cubrir las verdaderas necesidades de un diseño que requiere de conjuntos. Esto obliga a buscar otras alternativas de diseño, lo cual redunda en diseños más complicados. A continuación presentamos una lista no exhaustiva de problemas que el modelo actual trae aparejados.

Conjuntos Infinitos

La primera limitación del diseño actual es la imposibilidad de modelar conjuntos infinitos. Esto representa un escollo a la hora de necesitar una abstracción de este tipo y obliga a buscar alternativas que complican los diseños.

Por ejemplo, el conjunto de los números pares, a pesar de ser una abstracción sumamente utilizada en la matemática, no puede representarse como una instancia de la clase Set (o su equivalente en cada ambiente) y deben buscarse alternativas. Podría tal vez utilizarse un método que resuelva la pertenencia o no de un número a dicho conjunto (Number>>even en Smalltalk), modelar el conjunto como un Singleton [GAMMA] cuyo método #includes: contenga el código para resolver la pertenencia, u otras alternativas. Cada uno de estos modelos trae aparejados sus propios problemas: en el primer caso, por ejemplo, el conjunto en sí no estará modelado con un objeto, mientras que en el segundo caso el problema será la explosión de clases que se producirá al querer representar otros conjuntos.

Semántica de los Conjuntos

Pensamos que una falencia importante del modelo actual es que, al ser los conjuntos simples enumeraciones de elementos, se pierde su semántica. Es decir, la semántica del conjunto no queda representada más que dentro del código, en el nombre de las variables que se utilicen para referenciarlo, y tal vez en algún comentario en el código fuente, pero no en el objeto en sí. Si uno manipula directamente el objeto que representa el conjunto, como es habitual hacer a través de los inspectores en los ambientes basados en objetos, sólo verá los elementos que pertenecen al conjunto, y en el mejor de los casos, podrá intentar adivinar el criterio que los agrupa. Observe como ejemplo un inspector sobre un conjunto tal como se ve en Squeak:

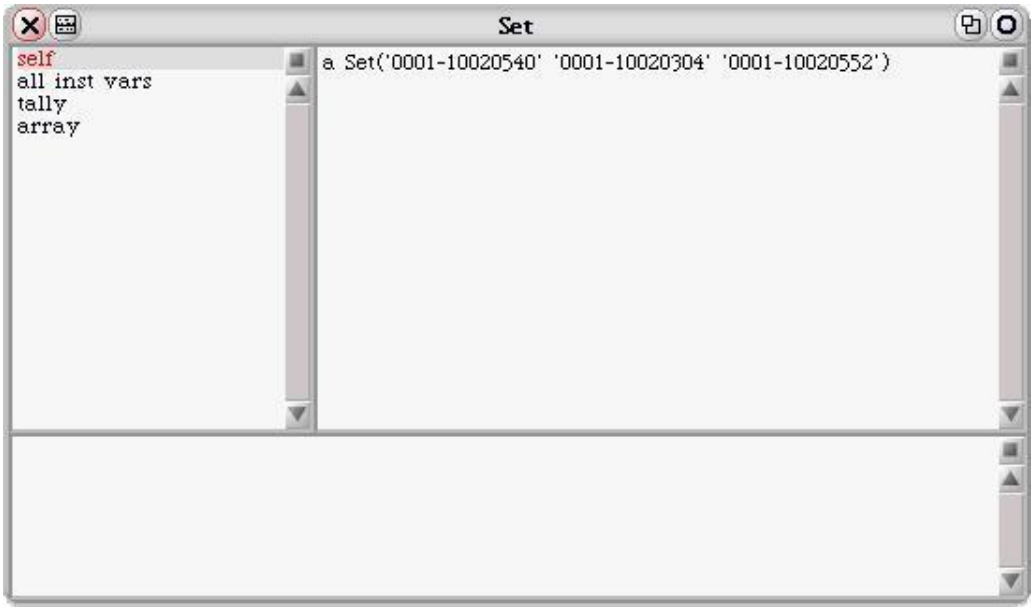


Figura: Un Inspector de Squeak

El conjunto del ejemplo contiene los números de las facturas impagas de una empresa. Es imposible deducir eso al inspeccionar el conjunto, y no por una falencia del inspector, sino porque dicha semántica no es accesible desde el conjunto. Nuestra aspiración es que al inspeccionar el conjunto, pueda entenderse lo que el mismo representa. En este ejemplo, las facturas que cumplen que:

- No fueron pagadas
- El tiempo transcurrido desde su fecha de emisión es mayor al plazo de pago del cliente

Esto es imposible si el conjunto es sólo una enumeración de elementos.

Creemos entonces que sería sumamente útil contar con la definición del conjunto expresada de manera declarativa, es decir, poder definir el conjunto por comprensión. Más aún, aspiramos a que dicha definición del conjunto sea accesible. Siguiendo con el ejemplo anterior, pensemos en que si las condiciones financieras cambiasen, debería poderse redefinir que las facturas impagas sean ahora las que:

- No fueron pagadas
- Transcurrieron 30 días o más desde su fecha de emisión

Estando el conjunto definido por comprensión, se puede visualizar la vieja definición, revisarla y experimentar con definiciones nuevas.

Además, el concepto de Facturas Impagas queda explícito en el modelo, independientemente de los elementos que contenga en un momento dado. Queda así representado en el programa un concepto fundamental del dominio y por lo tanto el mismo pasa a ser un mejor modelo de la realidad.

La naturaleza de los ambientes basados en objetos, unida a nuevas herramientas que permitan manipular de forma simple las definiciones de los conjuntos, deberían contribuir a achicar el *gap semántico* que presentan los sistemas tradicionales, acercando el sistema a su usuario y permitiendo la manipulación más directa de los datos y conceptos del negocio.

Actualización de la Pertenencia al Conjunto

Un tema que puede haberse pasado por alto en el ejemplo anterior, pero que según nuestro punto de vista es importante, es que contar con definiciones por comprensión hace posible que el conjunto actualice automáticamente qué elementos pertenecen al mismo, aliviando al resto del modelo de esta responsabilidad. Esto podría ocurrir porque un cambio en el ambiente determine que un elemento pertenezca o deje de pertenecer al conjunto.

Para ejemplificar supongamos que se tiene un conjunto de clientes morosos. Con conjuntos como simples contenedores de datos, cuando un cliente se convierte en moroso, en alguna parte del modelo se deberá incorporar manualmente al objeto cliente dentro del conjunto:

```
clientesMorosos add: unCliente.
```

Y removerlo cuando deja de serlo:

```
clientesMorosos remove: unCliente.
```

Por el contrario, con un conjunto por comprensión la manipulación manual del conjunto ya no es necesaria. Si definimos al conjunto de clientes morosos con la siguiente expresión:

```
clientesMorosos :=
      '{[cliente saldoDeudor > 5000]}' toSSetInUniverse: clientes.
```

Cuando el cliente adeude un saldo mayor a 5000 pesos automáticamente el conjunto lo reconocerá como un cliente moroso.

Con este enfoque obtuvimos un modelo más sencillo que no debe preocuparse por sincronizar el estado de los conjuntos manualmente.

Además, los conjuntos avisan cuando un elemento empieza o deja de pertenecer al mismo, lo que permite que la aplicación realice los cambios necesarios. Siguiendo con el ejemplo anterior, la aplicación podría enviar un mail al cliente recordándole los pagos pendientes.

Por último queremos decir que la actualización automática posibilita lo que nosotros llamamos *anidamiento de conjuntos*. Es decir, si uno tiene por ejemplo el conjunto C = A U B y cambian los elementos de A, cambiarán también los de C. Se hace posible así modelar conceptos complejos de la realidad sin agregar complejidad innecesaria al modelo, anidando conocimiento de forma tal que se respete la relación real de los conceptos. Este modelo se mantendrá coherente de manera automática.

Programación Exploratoria

La Programación Exploratoria es una técnica que plantea la actividad de programación como una herramienta fundamental para el análisis y la obtención de la especificación correcta de un sistema. En contraste con las metodologías tradicionales, en donde la programación es utilizada para implementar una especificación, en Programación Exploratoria la programación *es* escribir la especificación [SAN88].

Aquí la programación se inicia con el primer bosquejo de especificación que se produce. La actividad de programación nos fuerza a responder preguntas que no habíamos pensado antes. Esto nos da nuevas perspectivas sobre el problema a resolver y una comprensión más real del dominio del problema [ROZ01]. El nuevo conocimiento se vuelca entonces en una nueva especificación del sistema y en una nueva implementación. Este proceso iterativo continúa hasta que la especificación y el software cumplan con todos los requerimientos del sistema.

Para que la Programación Exploratoria sea exitosa, el costo de experimentación debe ser bajo. El tiempo en implementar una idea debe ser lo suficientemente breve como para poder descartarlas cuando estas fracasan. El costo de experimentación se encuentra directamente influenciado por dos factores: el costo de realización de cambios en el código, de manera de poder modificar la iteración anterior y adaptarla a los nuevos conocimientos, y la disponibilidad de abstracciones que puedan ser reutilizadas [SAN88]. En este aspecto creemos que los conjuntos por comprensión brindan un aporte importante a la Programación Exploratoria.

Como hemos mencionado, el contar con conjuntos por comprensión posibilita al programador concentrarse únicamente en lo que importa, la semántica de los conjuntos. De esta manera se ahorra trabajo en implementaciones que utilicen contenedores de datos y sincronicen éstos con su semántica en el modelo. El hecho de que los conjuntos sean una abstracción bastante utilizada en muchos dominios, hace de este un aporte relevante.

Los ambientes de Smalltalk [SMA80] y Self [UNG87], han sido señalados por muchos autores (ver por ejemplo [SHE83] y [SAN88]) como ambientes ideales para la práctica de Programación Exploratoria tanto por su nivel de interactividad como por su riqueza de herramientas (Inspectores, Workspaces, Debugger, etc.). Siguiendo esta filosofía hemos construido para nuestros conjuntos por comprensión herramientas para incentivar y facilitar la Programación Exploratoria. Estas son `SSetInspector` y `SSetBrowser`.

La herramienta `SSetInspector` (figura siguiente) permite analizar un conjunto, obtener su definición, inspeccionarlo, ver si el conjunto es enumerable y si lo es acceder a sus valores, inspeccionarlos, etc. Permite también realizar operaciones de conjuntos entre este y otros, verificar la pertenencia de elementos al conjunto, etc.

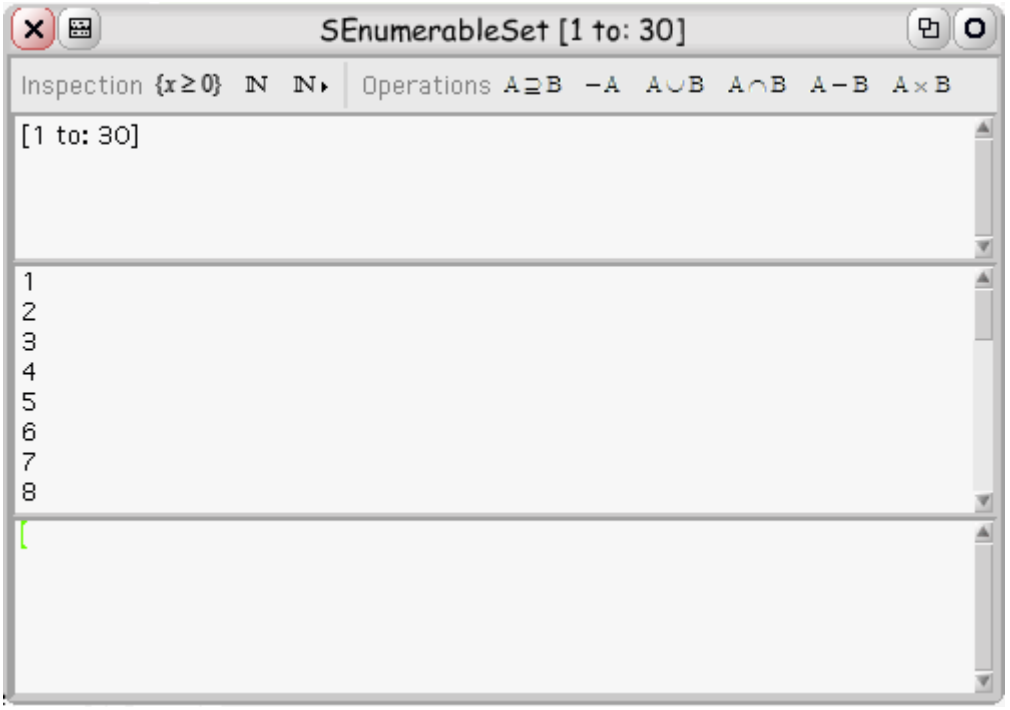


Figura: Un `SSetInspector` inspeccionando el conjunto {[1 to: 30]}

La herramienta `SSetBrowser` (figura siguiente) fue construida con el objetivo de proveer al programador una herramienta en donde este pueda sentirse libre y jugar con el poder declarativo de las definiciones por comprensión y las operaciones entre conjuntos. Es por medio de este "juego" de interacción que creemos posibilitar un mejor aprendizaje del dominio y las semánticas de los conjuntos por comprensión en el modelo.

Figura: `SSetBrowser`

Las herramientas `SSetInspector` y `SSetBrowser` se explican en detalle en la sección *Herramientas Desarrolladas*.

Nuevas Alternativas de Diseño

Finalmente, nuestra motivación radica también en que pensamos que teniendo un nuevo modelo de conjuntos que incluya todas estas cosas, se abrirán alternativas de diseño que no pueden pensarse a priori. Creemos que este modelo puede representar un salto cualitativo que permita al diseñador evaluar nuevas alternativas al momento de diseñar.

La matemática como inspiración

El punto de partida para el presente trabajo fue sin lugar a dudas la matemática y los conjuntos definidos por comprensión. Al ver una definición por comprensión, uno puede entender de qué se trata el conjunto, su semántica. Veamos como ejemplo la siguiente definición por comprensión:

$$P = \{ x \in N / x \bmod 2 = 0 \}$$

El conjunto P representa los números naturales pares. Las definiciones por comprensión exponen la importancia de poder definir un conjunto declarativamente: permiten hacerlo sin la necesidad de conocer de antemano sus elementos y, como dijimos, facilitan comprender la semántica de un conjunto ya dado.

Cabe aclarar que el enfoque de la tesis NO es el de modelar la matemática de manera estricta, sino el de tomarla como inspiración para llegar a un modelo de conjuntos que permita enriquecer el mundo de los objetos.

Si bien en un principio comenzamos modelando los conjuntos lo más parecido posible a la matemática, pronto encontramos diferencias entre ésta y el mundo de los objetos que hicieron que abandonásemos esa idea en función de un framework que resultase más útil a los desarrolladores de software en general.

Una diferencia importante entre la matemática y los ambientes de objetos está dada porque la matemática vive en un mundo estático, mientras que los ambientes de objetos son dinámicos. Un conjunto matemático que contiene ciertos elementos siempre va a contenerlos. En un ambiente de objetos, la pertenencia o no de un elemento a un conjunto puede ir cambiando a lo largo del tiempo.

Esto nos obligó a considerar en el framework aspectos como la actualización automática de los elementos, que mencionamos anteriormente, y que no tienen que ver con la matemática sino con el mundo de los objetos y con cómo será utilizado el framework para el desarrollo de aplicaciones. Un ejemplo interesante de esto último es que, como explicaremos más adelante, nos vimos en la necesidad de crear un nuevo tipo de conjunto, que fuese capaz no sólo de discriminar qué elementos pertenecen o no al mismo, sino también de enumerar sus elementos. Nótese la diferencia con la matemática, donde un conjunto puede *ser* enumerable, pero nunca va a realizar la acción de enumerar sus elementos.

Organización de la tesis

La siguiente es una lista de los capítulos que componen la tesis, con una descripción breve de cada uno:

1. Introducción: se divide en tres secciones. *Falencias del modelo actual de conjuntos* explica los problemas del modelo de conjuntos utilizado actualmente en muchos lenguajes de programación, los cuales intentamos resolver con el framework. *La matemática como inspiración* cuenta que si bien el framework se inspiró en los conjuntos matemáticos por comprensión, tuvo que alejarse de la matemática para tratar con aspectos propios de los ambientes de objetos. Finalmente, *Organización de la tesis* es esta sección.

2. Definiciones básicas: explica algunos conceptos necesarios para una mejor comprensión del trabajo.

3. Estado del Arte: muestra la implementación de conjuntos en muchos lenguajes del paradigma, para permitir comparar y ver las mejoras que propone este trabajo.

4. El Framework SSets: explica el diseño y funcionalidad del framework, qué se puede hacer con él y cómo.

5. Herramientas Desarrolladas: describe las herramientas provistas por el framework para trabajar con los conjuntos desde una interfase gráfica.

6. Prueba de Concepto: muestra una implementación de las herramientas *Senders Of*, *Implementors Of* y *Users Of* utilizando conjuntos por comprensión. Presenta la comparación de una implementación con y sin *SSets.*

7. Conclusiones: conclusiones obtenidas del trabajo.

8. Trabajo Futuro: enumera algunas mejoras al framework e ideas que pensamos sería interesante investigar para continuar con el trabajo.

9. Anexo I: Experiencia de uso: describe una experiencia de uso de los conjuntos SSet en una aplicación de mediana complejidad.

10. Anexo II: Notación de los Diagramas Utilizados: explica la sintaxis utilizada en los diagramas incluidos en el trabajo.

11. Bibliografía.

Definiciones Básicas

Programación Orientada a Objetos

En el paradigma de orientación a objetos sólo existen objetos que se comunican entre sí enviándose mensajes. Al recibir un mensaje, lo único que un objeto puede hacer es interactuar con otros objetos enviándoles más mensajes y luego devolver un resultado, el cual por supuesto no es ni más ni menos que un objeto. Por ejemplo, para obtener la cantidad de letras de la palabra “set” basta con enviarle a esa cadena de caracteres el mensaje *size*, lo cual retornará como resultado un objeto que representa el número tres.

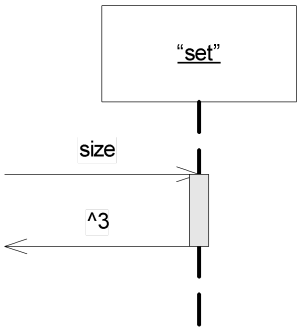


Figura: envío del mensaje *size* a la cadena de caracteres “set”

Pero... ¿qué es un objeto?

Un *objeto* es una abstracción de un ente de la realidad. Una representación computacional de algo, tangible o no, que se utiliza para modelar el dominio que nos interesa y construir programas. Los *programas* son modelos de la realidad llevados a la computadora. Recortes de la misma formados por objetos que interactúan para llevar a cabo una cierta tarea.

Una parte fundamental de la creación de un programa es entonces analizar el dominio sobre el cual se trabaja e identificar las abstracciones que se necesitan. Al realizar esta actividad, surge de inmediato lo compleja que la realidad es. Nuestro dominio de interés se cruzará con otros dominios y muchas veces las abstracciones que necesitamos pertenecerán a aquellos. Por ejemplo, si queremos construir un programa de facturación, seguramente surgirán las abstracciones Factura y Cliente, pero también necesitaremos los números, que pertenecen al dominio matemático. De esta manera se produce una *reutilización* de los objetos creados al estudiar otros dominios y ciertos modelos se constituyen en piezas básicas para la creación de otros.

Cuanto mejor modelado esté un dominio, más sencillo será que sean reutilizadas sus abstracciones y mejores diseños se obtendrán en los modelos que las reutilicen. Esto será una consecuencia natural de una mejor distribución de responsabilidades. Al quedar encapsulada la complejidad de cada dominio donde corresponde, aumentará la cohesión del modelo y disminuirá el acoplamiento.

Frameworks

Definición

Existen variadas definiciones de framework en la literatura de objetos ([Deu98], [RJ97], [GAMMA], etc.). Basándonos en la definición de [ROZ01], vamos a decir que un framework es un conjunto de objetos cooperantes conformando un diseño que encarna la teoría de un dominio. Este diseño debe ser además reusable, o en otras palabras, un framework debe estar preparado para facilitar su aprovechamiento en nuevas aplicaciones que utilicen dicho dominio.

Potencialmente podría construirse al menos un framework para cada dominio. Existen, por ejemplo, frameworks que permiten construir compiladores de manera sencilla [JML92], otros que permiten construir aplicaciones que gestionan modelos financieros [BE93], frameworks de unidades de medida [Wil06], etc.

Construir un framework es una tarea compleja. En la definición misma de framework, se nos revela por qué es difícil: no basta con construir una aplicación que permita resolver un problema particular de un dominio dado, sino que hay que modelar el dominio. Hay que estructurar y representar todo el conocimiento del dominio que las aplicaciones que utilicen el framework vayan a requerir. Al construir un framework se toman muchas decisiones de diseño que las aplicaciones reutilizarán. El framework define muchas de las clases y objetos que compondrán las aplicaciones y su distribución de responsabilidades. Esto permite que el diseñador de la aplicación se concentre en los problemas específicos de la misma, reutilizando el modelo desarrollado en el framework.

Instanciación de un framework

Al conjunto de actividades necesarias para poder utilizar un framework en una aplicación se lo llama *instanciación del framework* [ROZ01].

Muchos frameworks no proveen clases concretas que una aplicación pueda instanciar, sino clases abstractas de las que el desarrollador deberá derivar las suyas. A estos frameworks se los conoce como *frameworks de caja blanca*, ya que el desarrollador deberá conocer la implementación de las clases del framework para poder derivar sus subclases.

Los frameworks más maduros tienden a ser de *caja negra*, es decir que el reuso de los objetos del framework se realiza mediante mecanismos de composición de objetos en lugar de herencia de clases [SYN96]. Se les dice caja negra porque no será necesario para el desarrollador de aplicaciones conocer los aspectos internos del framework, sólo sus interfases. Esto hace que sean en general más sencillos de instanciar y la curva de aprendizaje sea mucho menor que para los frameworks de caja blanca.

Los frameworks de caja negra tienen también la ventaja de que las aplicaciones son mucho menos susceptibles a los cambios en el mismo, ya que no se verán prácticamente afectadas mientras se respeten las interfases.

Como veremos más adelante, SSets está desarrollado como un framework de caja negra. Suponemos que esto facilitará a los desarrolladores de aplicaciones la decisión de adoptarlo.

Smalltalk y Squeak

Smalltalk

Smalltalk es el ambiente de programación orientada a objetos que elegimos para implementar las ideas de la tesis. Hay muchas implementaciones de Smalltalk, para distintas plataformas, comerciales, open source, etc. La que utilizamos nosotros se llama Squeak.

Smalltalk nació en los años ’70 en los laboratorios del Xerox Palo Alto Research Center (Xerox PARC), al cual debemos también invenciones tales como la impresora láser, la red Ethernet, el disco óptico, la arquitectura cliente/servidor, etc. Fue creado por un equipo liderado por Alan Kay e integrado por Dan Ingalls, Ted Kaehler y Adele Goldberg.

En ese momento el enfoque más popular era ver a las computadoras como bienes de uso exclusivo de técnicos y científicos, debido mayormente a su elevado costo y complejidad. Alan Kay había concebido en 1968 la idea del *Dynabook*, una computadora parecida a las modernas tablet-PC, idea sumamente novedosa para la época. Él pensaba que las computadoras debían ser accesibles para el público en general, y que para eso debían adaptarse a las necesidades de la gente y ser más amigables.

Si bien Alan Kay y su equipo no pudieron construir el Dynabook, mayormente debido a las limitaciones tecnológicas de la época, se acercaron lo más que pudieron, llegando a construir una computadora personal bastante similar a las actuales. Para desarrollar en dicha computadora, y siguiendo su idea de hacer a la computadora más amigable, crearon Smalltalk, un ambiente de programación basado en los conceptos de objeto y mensaje, al que consideraron más cercano al pensamiento humano. Smalltalk incorporó también otras ideas novedosas, como el sistema de ventanas para la interfase de usuario.

Smalltalk es una implementación *pura* del paradigma de orientación a objetos. Es decir, todo lo que hay que en el ambiente son objetos y mensajes. Esto lo convierte en una excelente opción para investigar dentro del paradigma.

Squeak

En 1995, Apple reunió un equipo basado en el viejo grupo de Xerox PARC con el objetivo de experimentar en software educativo y construir el Dynabook. El equipo quedó conformado por Alan Kay, Dan Ingalls, Ted Kaehler, John Maloney y Scott Wallace.

Como parte del proyecto crearon Squeak, una versión de Smalltalk moderna y open-source, a partir del cual se encolumnó una gran comunidad de desarrolladores muy activa actualmente, que continúa mejorándolo y aportando ideas y desarrollos, lo cual hace de Squeak un ambiente en continua evolución. Esto representa para nosotros un incentivo adicional para elegir Squeak como ambiente de programación, ya que nuestro desarrollo será puesto a disposición de la comunidad, evaluado, criticado y eventualmente enriquecido con sus aportes.

Squeak es uno de los ambientes de desarrollo más portables que hay en la actualidad ya que sus autores tuvieron la idea de construir una máquina virtual muy simple y pequeña, escrita en Smalltalk mismo, que se traduce a C y puede ser compilada y ejecutada en casi cualquier plataforma. Esto contribuye por un lado a ampliar la comunidad de usuarios y por el otro, hace accesible la propia máquina virtual al programador (se puede crear y debuggear una instancia de la máquina virtual desde dentro de Squeak mismo). No hay en Squeak elementos que funcionen gracias a mecanismos internos inaccesibles, pudiéndose modificar absolutamente cualquier parte del sistema, lo cual lo hace ideal para investigar y experimentar.

Estado del Arte

Como mencionamos anteriormente, el concepto de conjunto como enumeración o contenedor de elementos existe en todos o casi todos los lenguajes del paradigma o lenguajes híbridos (lenguajes imperativos con orientación a objetos). Sin embargo no encontramos ninguna implementación de conjuntos definidos por comprensión.

La siguiente es una reseña de cómo manejan conjuntos algunos de los principales lenguajes de programación que revisamos:

Smalltalk

La librería estándar de clases de Smalltalk [SMA80, SMAANSI] incluye la jerarquía *Collection*, que representa las colecciones de objetos. No se provee ningún mecanismo fuera de la enumeración de elementos para la creación de colecciones.

Ejemplo:

```
aSet := Set withAll: { factura1. factura2. factura3 }.
```

C++

La biblioteca Standard Template Library [STR97] provee al programador de C++ *templates* para la representación de colecciones de objetos. Sin embargo ninguno de los provistos permite la creación de colecciones por comprensión.

Ejemplo:

```
std::set<Factura> aSet;
aSet.insert( factura1 );
aSet.insert( factura2 );
aSet.insert( factura3 );
```

Java

Al igual que Smalltalk, Java provee una jerarquía de clases llamada *Collection* para la representación de colecciones [ARN96, JAVAAPI]. Como en Smalltalk, ninguna de las representaciones provistas permite la creación de colecciones por comprensión.

Ejemplo:

```
HashSet aSet = new HashSet();
aSet.add( factura1 );
aSet.add( factura2 );
aSet.add( factura3 );
```

Encontramos sin embargo una clase llamada *PredicatedSet* [SETUTILS], en la implementación de Apache, que extiende la implementación default de Java. Básicamente se trata de un *Decorator* [GAMMA] que asocia un predicado a un conjunto, de manera tal de que sólo se puedan agregar al conjunto elementos que hagan verdadero al predicado.

Si bien esto está lejos aún de la implementación de conjuntos por comprensión de SSets, representa un acercamiento con respecto a los conjuntos por enumeración, que permite tener la semántica del conjunto explícita en el modelo de objetos.

.NET

En la plataforma .Net [DOTNET] hay diversas interfases e implementaciones de colecciones, pero ninguna de ellas permite definir colecciones por comprensión.

```
ArrayList col = new ArrayList();
      col.Add( factura1 );
      col.Add( factura2 );
      col.Add( factura3 );
```

Python

Python es otro lenguaje de programación orientado a objetos que se ha vuelto muy popular en estos últimos años de la mano de la popularidad creciente de los llamados “Scripting Languages”.

Uno de los aspectos destacados en Python es su importante librería de clases [PYTHON]. En la misma hemos buscado el soporte de colecciones de objetos y hemos encontrado que no provee colecciones de objetos por comprensión.

Ejemplo:

```
      aSet = set()
      aSet.add( factura1 )
      aSet.add( factura2 )
      aSet.add( factura3 )
```

Self

Self [UNG87] es un ambiente de objetos inspirado en gran parte en Smalltalk, pero a diferencia de este último provee un ambiente de objetos sin clases, basado en prototipos y acceso uniforme a comportamiento y estado.

Entre el trabajo de investigación realizado por el equipo de Self se encuentra un rediseño de la jerarquía Collection del Smalltalk-80. Hemos estudiado la misma y tampoco provee ningún mecanismo fuera del de enumeración para la creación de colecciones de objetos.

Strongtalk

El ambiente de objetos Strongtalk es un rediseño de Smalltalk-80. Mientras mantiene la sintaxis y semántica de Smalltalk, Strongtalk lo extiende con un sistema opcional de tipado estático y un rediseño de su librería de clases [BRA96].

Si bien Strongtalk provee una nueva implementación de la jerarquía Collection antes citada, no innova en los mecanismos de definición de las mismas.

Otros lenguajes

Además de los antes mencionados hemos estudiado Ruby [RUBY] y Eiffel [EIFFELI]. Ambos son lenguajes orientados a objetos que cuentan con librerías de clases importantes. Sin embargo no hemos encontrado soporte o mención de colecciones de objetos por comprensión en ninguno de ellos.

El Framework SSets

Al ver la falta de conjuntos por comprensión en los lenguajes de programación orientada a objetos, nos preguntamos por qué no estarían modelados. La respuesta a la cual llegamos fue simplemente que… ¡no sabemos por qué no están, pero sería muy bueno contar con ellos! Nos pusimos entonces manos a la obra en la creación del framework de conjuntos al que llamamos *SSets*.

Tal como lo pensamos y desarrollamos, y como explicaremos en detalle en las secciones que siguen, estas son algunas de las características que ofrece SSets:

- Permite instanciar conjuntos infinitos y operar con ellos
- Visualización de la definición de un conjunto al inspeccionarlo
- Actualización automática de la pertenencia de elementos al conjunto
- Notificación de cambios en los elementos de un conjunto
- Permite definir conjuntos sin necesidad de conocer previamente sus elementos
- Operar con conjuntos mediante herramientas diseñadas a tal efecto

Desde nuestro punto de vista, lo más importante que logra el framework es ampliar el abanico de posibilidades del diseñador, ofreciendo una alternativa a considerar en casos donde la necesidad de conjuntos se cubría mediante soluciones *ad-hoc* que muchas veces tenían un alto costo asociado (ver *Falencias del modelo actual de conjuntos*) .

Modos de utilizar SSets

SSets permite crear y manipular los conjuntos de dos maneras:

- Programáticamente: permite utilizar los conjuntos desde las aplicaciones
- Interactivamente: permite operar con ellos utilizando herramientas gráficas

En este capítulo expondremos la funcionalidad de SSets explicando cómo se usa programáticamente. Mostraremos también el diseño del framework y cómo permite superar las limitaciones del modelo actual de conjuntos expuestas en la sección *Falencias*…. En secciones posteriores explicaremos las herramientas gráficas.

Sobre el diseño del framework

Para exponer el diseño del framework nos valemos de los diagramas de objetos, de secuencias y de clases, tal como se explican en el *Anexo II: Notación de los Diagramas Utilizados*.

El capítulo no intenta hacer un recorrido exhaustivo de las clases y métodos, sino mostrar lo suficiente como para que el diseño se comprenda. Ante cualquier duda, recomendamos ir a Squeak mismo y ver las clases y métodos.

Queremos resaltar que llegar al diseño que se muestra a continuación fue un arduo proceso que nos llevó al menos 3 iteraciones principales (donde el diseño sufrió grandes cambios), e infinidad de pequeños retoques en cada iteración. El diseño fue simplificándose a medida que avanzamos con las iteraciones y finalmente obtuvimos un modelo sencillo y fácil de usar.

Aprovechamos la mención de las muchas modificaciones que sufrió el modelo para resaltar lo beneficioso que nos resultó utilizar SUnit [SUNIT, SUNIT2]. SUnit es un framework que permite construir tests de regresión y ejecutarlos de manera muy simple cuantas veces se desee. Utilizando SUnit construimos una batería importante de tests, que pueden encontrarse en la categoría *SSets-Tests*. Esto nos permitió encontrar muchos bugs introducidos accidentalmente al modificar el modelo y es también una muy buena fuente de aprendizaje para ver cómo se utilizan las distintas funcionalidades de SSets.

Tipos de conjunto

Antes de meternos de lleno en la creación y uso de conjuntos, explicaremos los diferentes tipos de conjunto que maneja el framework y las características de cada uno. Para ello vamos a trazar un paralelo con la matemática.

En matemática existen los conjuntos definidos por enumeración y los conjuntos definidos por comprensión. Recordemos viendo el siguiente ejemplo:

Por enumeración: $A = \{ 0, 2, 4, 6, 8, 10 \}$

Por comprensión: $A = \{ x \in N / x \leq 10 \ \wedge \ x \bmod 2 = 0 \}$

Ejemplo: definición del conjunto A de números pares entre 0 y 10

Los **conjuntos por enumeración** se definen diciendo explícitamente qué elementos pertenecen al mismo.

Los **conjuntos por comprensión** se definen mediante un predicado que es verdadero para los elementos del conjunto y falso para los que no pertenecen al mismo.

Conjuntos definidos por enumeración

Los conjuntos definidos por enumeración, tal cual se usan en matemática, ya están modelados en la mayoría de los lenguajes de programación orientada a objetos, como mencionamos en la sección *Estado del Arte*. En Smalltalk en particular están representados por la clase `Set`.

Veamos nuevamente un ejemplo de cómo se definen, creando el conjunto A del ejemplo anterior:

```
setA := Set withAll:{0. 2. 4. 6. 8. 10}.
```

La clase `Set` de Smalltalk soporta el protocolo `{#add:, #remove:}`. Esto quiere decir que uno puede modificar el conjunto evaluando por ejemplo:

```
setA add: 12.              A = { 0, 2, 4, 6, 8, 10, 12 }
setA remove: 4.            A = { 0, 2, 6, 8, 10, 12 }
```

Si bien esto otorga dinamismo a los conjuntos, realizar los `#add:` y `#remove:` es muy costoso en ciertos casos, sobre todo si se tiene una aplicación con gran cantidad de conjuntos que deben mantenerse sincronizados.

Ilustremos este punto con un ejemplo: tengo el conjunto de clientes de una empresa en la variable global `Clients`:

```
Clients    { client1 (edad 30),
             client2 (edad 33),
             client3 (edad 40) }
```

Quiero tener el conjunto de clientes cuya edad es menor de 35 años en la variable `youngClients`. Utilizando la clase Set podría hacer:

```
youngClients := Set new.
youngClients add: client1.
youngClients add: client2.
```

O mejor aún, seleccionar de `Clients` aquellos clientes menores a 35 años:

```
youngClients := Clients select:[:client | client edad < 35]
```

El resultado será:

```
youngClients    {  client1 (edad 30), client2 (edad 33) }
```

El ejemplo funciona tal como esperábamos, pero el problema es que en `youngClients` tengo la lista de clientes menores a 35 años de un momento dado, el momento en el cual se evaluó la expresión con la cuál lo creé. ¿Qué pasa cuando se agrega un nuevo cliente a `Clients`? La respuesta: nuestro conjunto `youngClients` queda desactualizado. Esto obliga a nuestra hipotética aplicación a ocuparse de mantener la sincronía, haciendo los `#add:` y `#remove:` de `youngClients` que sean necesarios.

La raíz del problema está en que la clase `Set` no sabe nada acerca de la semántica de los conjuntos.

En SSets creamos un tipo de conjunto que es muy parecido a los conjuntos definidos por enumeración, pero que conoce su semántica y por lo tanto puede mantenerse actualizado. Lo llamamos ***Enumerable Sets***. Veamos un ejemplo:

```
youngClients :=
      '{[Clients select:[:client | client edad < 35]]}' toSSet
```

El conjunto `youngClients` se ocupará de mantenerse sincronizado con `Clients`. Y no sólo eso, si cambia por ejemplo la edad de algún cliente, también se sincronizará automáticamente.

Conjuntos definidos por comprensión

Los conjuntos por comprensión se definen en SSets, al igual que en matemática, mediante un predicado al que llamamos *predicado discriminador*, porque discrimina entre los elementos que pertenecen al conjunto y los que no. A este tipo de conjuntos los llamamos en el framework ***Intentional Sets***.

Volviendo con el ejemplo del conjunto A, podemos definirlo como:

```
setA := '{[(x >= 0) & (x <= 10) & (x even)]}' toSSet
```

A diferencia de los Enumerable Sets, la definición en estos conjuntos consiste en un predicado que evalúa a verdadero para los elementos que pertenecen al conjunto, y sólo para ellos. Esto hace que este tipo de conjuntos sean no enumerables (a menos que puedan enumerarse los elementos de su universo, como veremos más adelante) pero tiene como ventaja que puede predicarse sobre conjuntos infinitos y conjuntos que no sabemos cómo enumerar. Además, las definiciones por comprensión suelen ser más fáciles de entender.

Nota: a pesar de que a lo largo del informe utilizamos x como nombre de variable en la mayoría de las definiciones por comprensión, esto no es necesario. SSets distingue y utiliza la variable libre de las definiciones.

La jerarquía SSet

Comenzaremos la explicación de framework mostrando la que podría decirse que es la jerarquía con las clases principales del modelo. Son las clases que representan a los distintos tipos de conjunto.

Como puede verse en el diagrama que sigue, la jerarquía se compone de las siguientes clases:

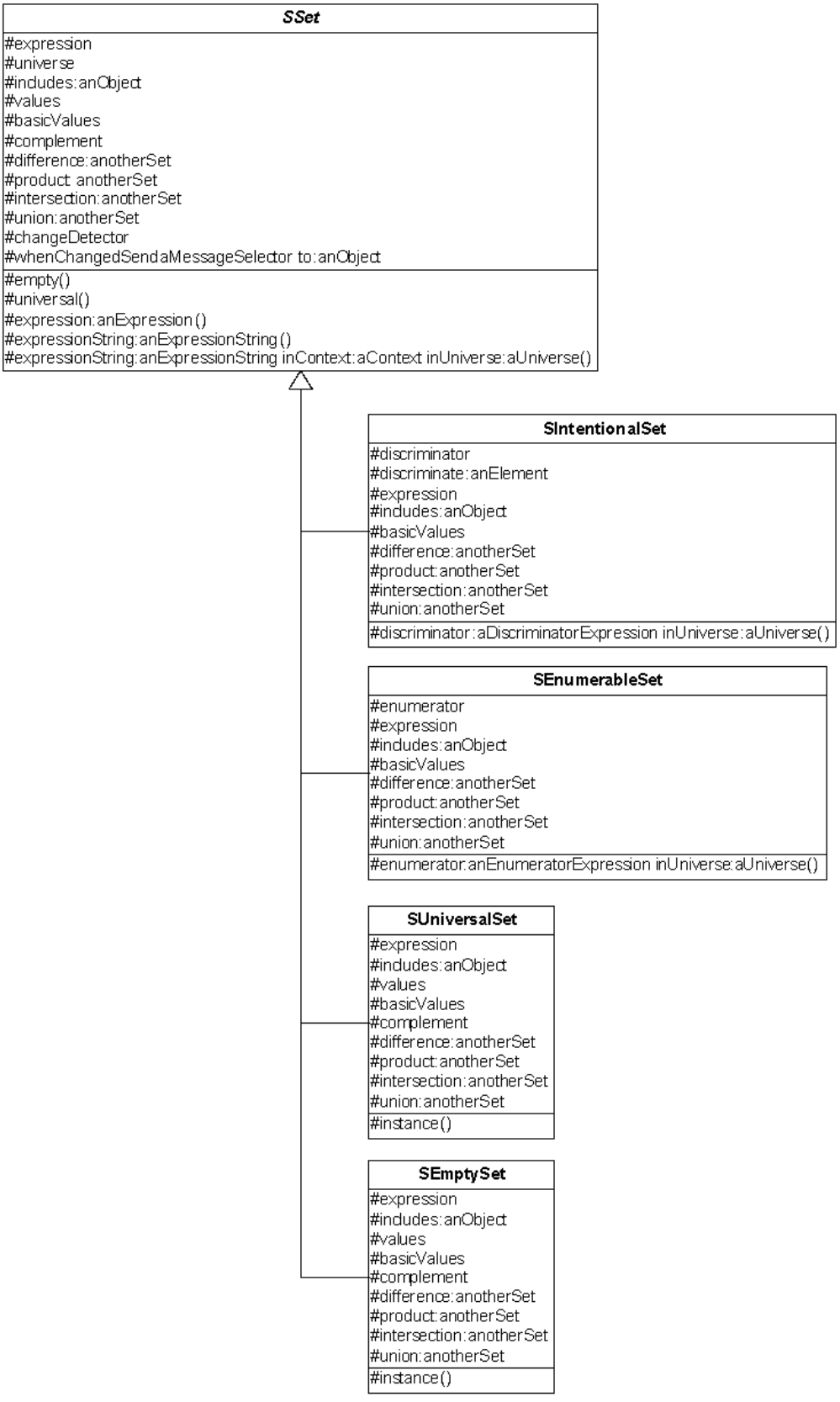
- *SSet*: es una clase abstracta que representa el concepto de conjunto, y contiene el protocolo que todo conjunto debe satisfacer. Tiene métodos de clase que permiten crear conjuntos de todos los tipos, actuando como Façade [GAMMA] para evitar que el usuario necesite conocer la jerarquía SSet.

- *SIntentionalSet:* representa los conjuntos por comprensión. Tienen un *discriminador* que indica cuáles elementos pertenecen al conjunto y cuáles no.

- *SEnumerableSet:* representa los conjuntos por enumeración. Tienen un *enumerador* que al evaluarse retorna los elementos que pertenecen al conjunto.

- *SUniversalSet:* es un Singleton [GAMMA] que representa el conjunto universal, que contiene todos los elementos.

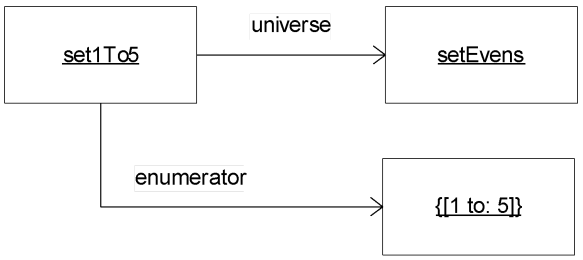
- *SEmptySet:* es un Singleton que representa el conjunto vacío. Tiene al conjunto universal por universo.


SSet
#expression
#universe
#includes:anObject
#values
#basicValues
#complement
#difference:anotherSet
#product anotherSet
#intersection:anotherSet
#union:anotherSet
#changeDetector
#whenChangedSendaMessageSelector to:anObject
#empty()
#universal()
#expression:anExpression()
#expressionString:anExpressionString()
#expressionString:anExpressionString inContext:aContext inUniverse:aUniverse()
SIntentionalSet
#discriminator
#discriminate:anElement
#expression
#includes:anObject
#basicValues
#difference:anotherSet
#product:anotherSet
#intersection:anotherSet
#union:anotherSet
#discriminator:aDiscriminatorExpression inUniverse:aUniverse()
SEnumerableSet
#enumerator
#expression
#includes:anObject
#basicValues
#difference:anotherSet
#product:anotherSet
#intersection:anotherSet
#union:anotherSet
#enumerator:anEnumeratorExpression inUniverse:aUniverse()
SUniversalSet
#expression
#includes:anObject
#values
#basicValues
#complement
#difference:anotherSet
#product:anotherSet
#intersection:anotherSet
#union:anotherSet
#instance()
SEmptySet
#expression
#includes:anObject
#values
#basicValues
#complement
#difference:anotherSet
#product:anotherSet
#intersection:anotherSet
#union:anotherSet
#instance()

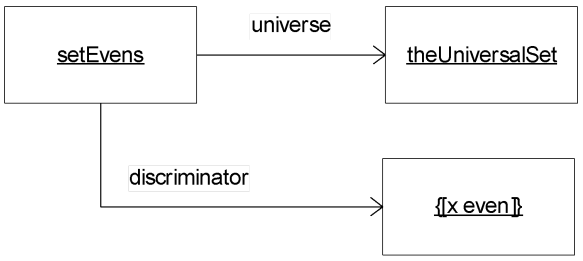
Diagramas de objetos

Los siguientes diagramas muestran ejemplos de cómo se relacionan los conjuntos con otros objetos del modelo.

- **El conjunto enumerable de números del 1 al 5, en el universo de los números pares:**



- **El conjunto por comprensión de números pares:**



Definición de un conjunto

Existen varias maneras de obtener un conjunto en SSets:

1. Dando su definición (como en los ejemplos anteriores)
2. Utilizando los métodos `#empty` o `#universal` de la clase `SSet`
3. Operando con conjuntos preexistentes

Veremos ahora (1) y explicaremos las demás opciones en las secciones que siguen.

La definición de un conjunto

Como dijimos anteriormente, un conjunto tiene una definición que especifica qué elementos pertenecen al mismo. Para crear un conjunto tenemos que pasar dicha definición como parámetro a algún método de la clase `SSet` encargado de crear el conjunto.

La definición será de distinto tipo según el tipo de conjunto que queremos crear:

Intentional Sets**:** la definición debe ser un predicado con una variable libre, tal que reemplazando cada elemento en dicha variable, el predicado sea verdadero sólo para los elementos que pertenecen al conjunto.

Ejemplos:

```
{[(x >= 0) & (x <= 10)]}
Define el conjunto de números entre 0 y 10.

{[x even]}
Define el conjunto de los números pares
Number>>#even es un método de Smalltalk
```

Enumerable Sets: la definición al evaluarse debe dar como resultado una colección con los elementos del conjunto.

Ejemplos:

```
{[Boolean subclasses]}
Define el conjunto de las subclasses de Boolean

{[1 to:5]}
Define el conjunto de números enteros del 1 al 5
```

El framework *Logic Expressions*

Para poder escribir las definiciones de manera sencilla, y a la vez para que SSets pueda utilizarlas y operar con ellas, desarrollamos en el transcurso de la tesis un segundo framework al que llamamos Logic Expressions (LE). El objetivo de LE es representar expresiones de lógica de primer orden como objetos, permitiendo evaluar dichas expresiones y operar con ellas.

Así, por ejemplo, si quiero obtener la representación de la expresión "$p \Rightarrow q$" en la variable de Smalltalk `myExpression`, puedo ejecutar:

```
p := #p asLVariable.
q := #q asLVariable.
myExpression := (p => q).
```

Si quisiera evaluar si "verdadero implica verdadero", "falso implica verdadero", etc., voy a tener que pasar al método `#value:` las valuaciones correspondientes. Ejemplo:

```
myExpression value: {#p -> true. #q -> true}        true
myExpression value: {#p -> false. #q -> true}       true
myExpression value: {#p -> true. #q -> false}       false
myExpression value: {#p -> false. #q -> false}      true
```

También se puede operar con las expresiones, combinándolas de manera sencilla. Esta característica es muy utilizada por SSets al operar con los conjuntos, para combinar las definiciones de los conjuntos y obtener el conjunto resultante.

Ejemplos:

- `myExpression | (#r asLVariable)` $(p \Rightarrow q) \vee r$
- `myExpression not` $\neg(p \Rightarrow q)$
- `myExpression & (LConstant false)` $(p \Rightarrow q) \wedge false$

LE trae además incorporado un parser, lo cual permite crear las expresiones escribiéndolas en un lenguaje parecido al de la matemática. Esto facilita escribir expresiones complejas. Por ejemplo, para obtener la expresión "$(p \Rightarrow q) \equiv (\neg p \vee q)$", puedo hacer:

```
LExpression fromString: '{p => q = not p or q}'
```

Otra característica interesante de LE son las *Block Expressions*, que permiten escribir código Smalltalk como parte de una expresión. Esto aumenta la potencia de LE. Para definir este tipo de expresiones, basta incluir un bloque como parte de la expresión.

Ejemplos:

- La expresión expIsEven retorna verdadero si y sólo si x es par:

```
expIsEven := '{[x even]}' asLExpression.
expIsEven value: {#x -> 1}.            false
expIsEven value: {#x -> 2}.            true
```

- La expresión expA evalúa a verdadero sólo para las cadenas de caracteres que empiezan con la letra *a*:

```
expA := '{[str beginsWith: ''a'' ]}' asLExpression.
expA value: {#str -> 'abcd'}.          true
expA value: {#str -> 'zabcd'}.   false
```

- La expresión exp evalúa a verdadero sólo cuando *x* es par e *y* es mayor a cinco:

```
exp := '{[x even] and [y > 5]}' asLExpression.
exp value: {#x->2. #y->6}.             true
exp value: {#x->2. #y->4}.             false
exp value: {#x->1. #y->8}.             false
```

Lo último que mencionaremos de LE es el concepto de contexto de una expresión: el contexto de una expresión es el conjunto de nombres utilizados en la expresión que no corresponden a variables, y los objetos a los que dichos nombres deben asociarse al evaluar la expresión. Esto nos da la posibilidad de separar conceptualmente las variables de la expresión de los otros objetos que la misma debe conocer.

Por ejemplo, intentemos obtener el conjunto de los clientes de mi empresa, la cual tengo referenciada en la variable `myCompany`, definiéndolo con la expresión `'{[company clients]}'`:

```
exp := '{[company clients]}' asLExpression.
exp value.

  Error: "Variable 'company' not valuated"
```

El error se debe a que la expresión no conoce el nombre *company*, ya que no es una variable y no está en su contexto. Puedo cambiar esto agregando al contexto el binding entre los nombres de la expresión y los objetos a los que esos nombres refieren:

```
exp := '{[company clients]}' asLExpression.
exp context at:#company put:myCompany.
exp value.                clientes de la empresa
```

Por default, las expresiones tienen en su contexto a los objetos globales de Smalltalk, lo cual permite que los mismos no tengan que ser explícitamente agregados y permite predicar más fácilmente sobre objetos tales como las clases, etc.

Los métodos de creación de conjuntos

En la clase `SSet` existe toda una familia de métodos dedicada a la creación de conjuntos, la cual detallamos a continuación.

#expression: recibe como parámetro una expresión y devuelve el conjunto definido por la misma. El conjunto será un Intentional Set o un Enumerable Set dependiendo de la expresión pasada como parámetro. Si la expresión evalúa a una colección, intentará definirse un Enumerable Set. En caso contrario, un Intentional Set.

Ejemplos:

- Definimos el conjunto por comprensión setEvens, de los números pares:

```
expIsEven := LExpression fromString: '{[x even]}'.
setEvens := SSet expression: expIsEven.
```

- Definimos el conjunto enumerable de los números enteros del 1 al 5:

```
exp1to5:= LExpression fromString: '{[1 to:5]}'.
set1to5 := SSet expression: exp1to5.
```

#expressionString: recibe el string de la expresión como parámetro e intenta parsearlo y crear el conjunto. Esto ahorra el paso de crear la expresión.

Ejemplos:

- Definimos los conjuntos del ejemplo anterior como:

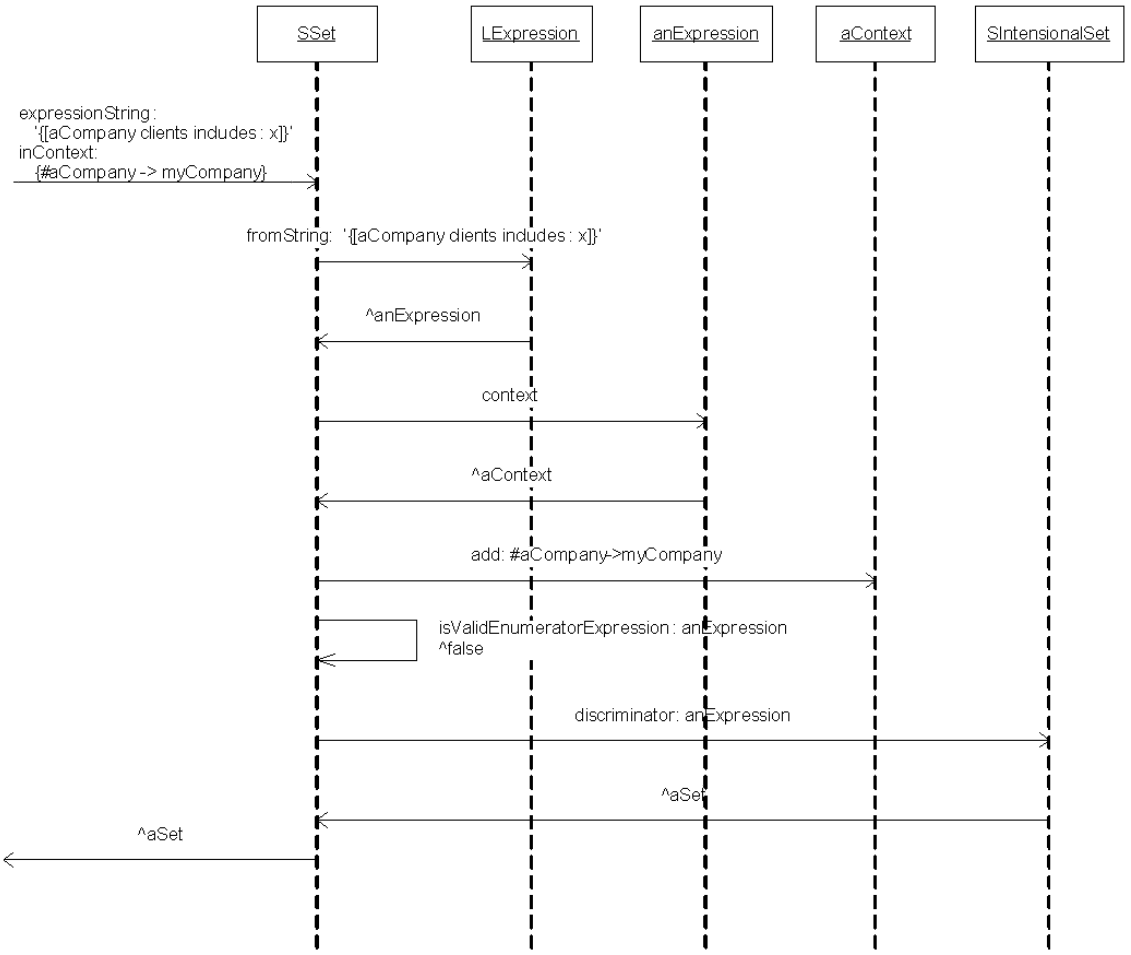
```
setEvens := SSet expressionString: '{[x even]}'.
set1to5  := SSet expressionString: '{[1 to:5]}'.
```

#expressionString:inContext: permite especificar el contexto de la expresión que define el conjunto.

Ejemplo:

- Definimos por comprensión el conjunto de clientes de la empresa referenciada por la variable *MyCompany*.

```
setClients :=
      SSet expressionString:
            '{[aCompany clients includes: x]}'
      inContext: {#aCompany -> myCompany}
```



#enumerator: el protocolo de métodos que comienza con #enumerator: es similar al de #expression:, sólo que directamente intenta crear un Enumerable Set. Esto posibilita atrapar algunos errores que de lo contrario pasarían desapercibidos. Veamos un ejemplo de este fenómeno:

- Creamos un Intentional Set sin querer, al olvidar incluir la colección myNumbers en el contexto de la expresión.

```
myNumbers := [1 to:5] value.
myEvenNumbers :=
      SSet expressionString: '{[myNumbers select:[:n|n even]]}'

  Resultado: un Intentional Set
```

La explicación de este error es que al no estar definido el nombre *myNumbers* en el contexto, el framework toma dicho nombre como la variable libre de la expresión y crea un Intentional Set con una definición equivalente a: {`[x select:[:n|n even]]}`.

Veamos ahora el mismo ejemplo, pero utilizando el método #enumeratorString:

```
myNumbers := [1 to:5] value.
myEvenNumbers :=
      SSet enumeratorString: '{[myNumbers select:[:n|n even]]}'

  Error: The enumerator in an enumerable set cannot have free variables.
```

El framework utiliza excepciones para notificar de los errores ocurridos.

#discriminator: ídem, pero estos métodos intentan crear directamente Intentional Sets.

String>>toSSet: pensando en brindar una forma más cómoda de crear conjuntos, agregamos a la clase String el método toSSet.
Ejemplos:

- Definimos los mismos conjuntos del ejemplo de *expressionString* anterior:

```
setEvens := '{[x even]}' toSSet.
set1to5  := '{[1 to:5]}' toSSet.
```

Los métodos #empty y #universal

Existen dos conjuntos en particular que son muy utilizados, incluso dentro del framework mismo, y hay en la clase `SSet` métodos que permiten obtenerlos directamente, sin tener que definirlos cada vez.

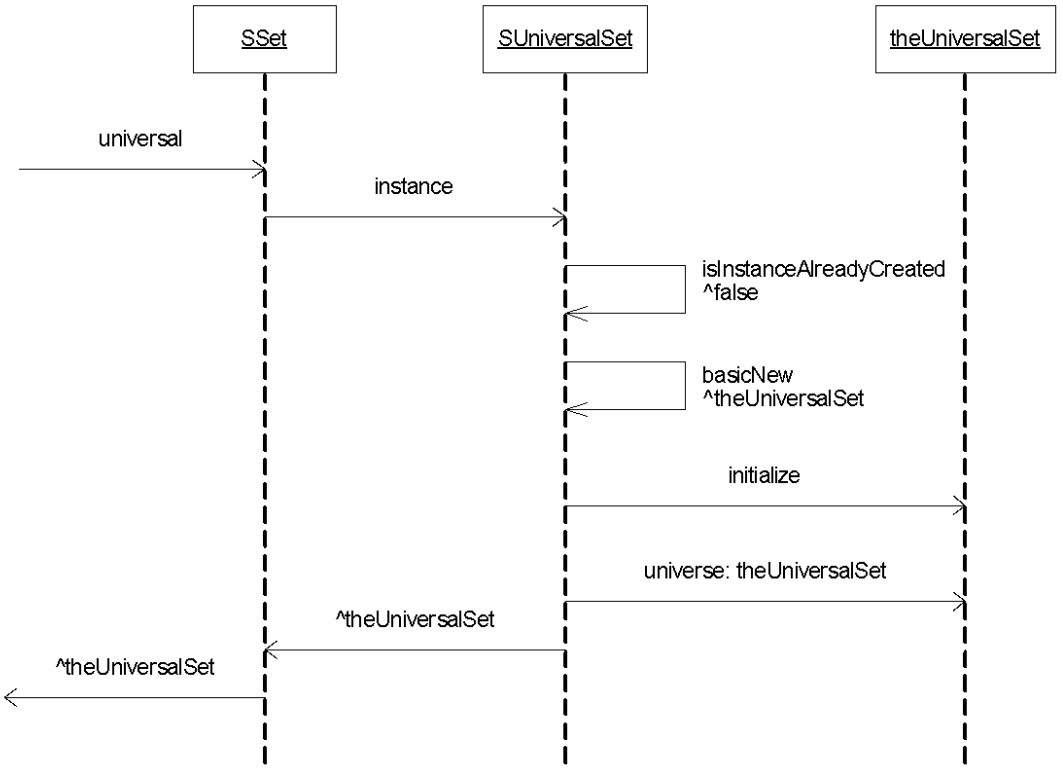
#empty: retorna el conjunto vacío.

#universal: retorna el conjunto universal. Lo definimos como un conjunto que incluye todos los elementos.

Nota: El método `#new` de la clase `SSet` fue además redefinido para que retorne el conjunto vacío.

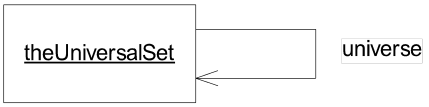
Estos conjuntos están implementados en el framework como Singletons [GAMMA]. Esto quiere decir que a pesar de que hay una clase que representa cada uno de estos tipos de conjunto, existirá una sola instancia de los mismos.

El diagrama que sigue muestra la obtención del conjunto universal. Corresponde al caso cuando es la primera vez que se lo utiliza y por lo tanto se crea el objeto. El conjunto vacío es creado de manera análoga.



Los siguientes diagramas de objetos muestran la relación entre los conjuntos vacío y universal con su propio universo:

- **El conjunto universal:**



- **El conjunto vacío:**

El universo de un conjunto

En SSets modelamos el universo de un conjunto de forma naïve, siguiendo el mismo concepto que se enseña en la escuela primaria mediante los diagramas de Venn. En estos diagramas, los conjuntos se representan con círculos y todos ellos están circunscriptos en un rectángulo, que es el conjunto universal.

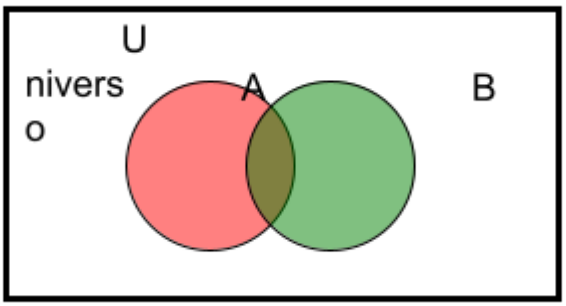


Figura: Diagrama de Venn

El conjunto universal no es más que un conjunto que se elige arbitrariamente bajo la sola condición de que debe contener a todos los elementos de interés con los cuales se está trabajando, de manera que siempre se opere dentro de ese universo. Así, los conjuntos que se definen son siempre subconjuntos del conjunto universal. Esta característica es sumamente útil, elegir un buen universo de trabajo puede simplificar mucho las definiciones de los conjuntos.

La operación Complemento, que devuelve otro conjunto con los elementos que no pertenecen al conjunto sobre el cual se aplica, también se limita a los elementos del universo con el cual se trabaja, y es entonces un buen test para ver si el resultado que se obtiene es el esperado, o en otras palabras, si el universo de trabajo está correctamente definido.

El universo de un conjunto puede obtenerse en SSets enviandole el mensaje `#universe` a un conjunto.

Creación de conjuntos con universo

Para cada método de creación de conjuntos que vimos anteriormente, existe uno equivalente que recibe también como parámetro el universo del conjunto. Así, contamos también en la clase `SSet` con los siguientes métodos de creación:

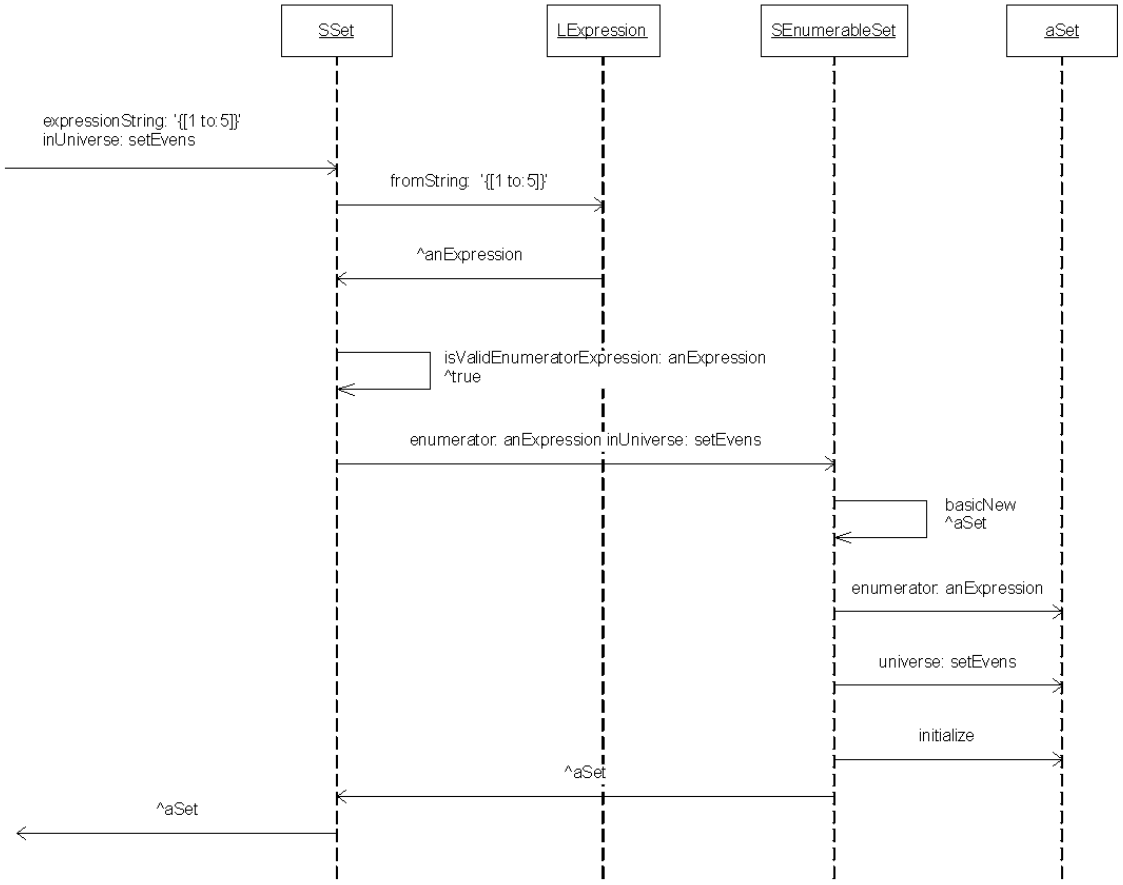
- `#expression:inUniverse:`
- `#expressionString:inUniverse:`
- `#expressionString:inContext:inUniverse:`
- `#enumerator:inUniverse:`
- `#enumeratorString:inUniverse:`
- `#enumeratorString:inContext:inUniverse:`
- `#discriminator:inUniverse:`
- `#discriminatorString:inUniverse:`
- `#discriminatorString:inContext:inUniverse:`
- `String>>toSSetInUniverse:`

Ejemplos:

- Teniendo en *theCompany* la representación de una empresa, definimos el conjunto de los mejores clientes como aquellos que compraron más de 1000.

```
setClients := SSet enumeratorString: '{[company clients]}'
                   inContext: {#company -> theCompany}.

setBestClients := SSet discriminatorString:
                            '{[client purchasedAmount > 1000]}'
                       inUniverse: setClients.
```

- Creación de un conjunto enumerable de números del 1 al 5, en el universo de los números pares.



Cuando se define un nuevo conjunto sin especificar su universo, su universo será el conjunto universal. Esto hace que el único factor que determine los elementos del conjunto sea la definición del conjunto misma.

El método #copyToUniverse

Este método permite “copiar” un conjunto a otro universo, o dicho de otra manera, crear un conjunto en otro universo, que tenga la misma definición que un conjunto que ya tenemos.

Ejemplo:

- Definimos el conjunto de los números del 1 al 10 y luego creamos uno con la misma definición, pero en el universo de los números pares.

```
set1to10 := '{[1 to:10]}' toSSet.
set1to10 values.                              {1, 2, 3, …,  10}

setEvens := '{[x even]}' toSSet.

set1to10evens := set1to10 copyToUniverse: setEvens.
set1to10evens values.                         {2, 4, 6, 8, 10}
```

Operaciones de conjuntos

Definimos en el framework cinco operaciones que pueden realizarse sobre los conjuntos: unión, intersección, diferencia, producto y complemento. A continuación hacemos un breve repaso de cada una de ellas.

Una de las cuestiones interesantes que encontramos al implementar las operaciones fue decidir en cada caso cuál debía ser el universo del conjunto resultante en caso de que se operase con conjuntos con distinto universo. Los criterios adoptados se explican en cada operación.

Como puede verse en los diagramas de secuencia, para el modelado de las operaciones entre conjuntos, se utiliza la técnica de double-dispatch, debido a que las operaciones se realizan de distinta manera dependiendo del tipo de conjunto con el cual se opere.

Unión

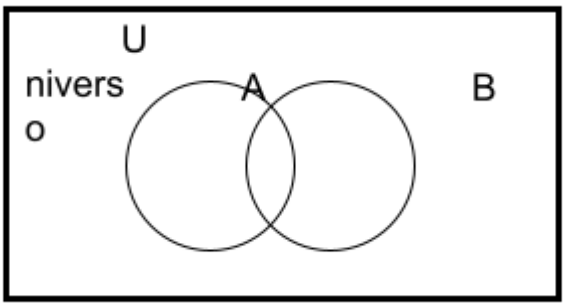


Figura: el área rayada representa AUB

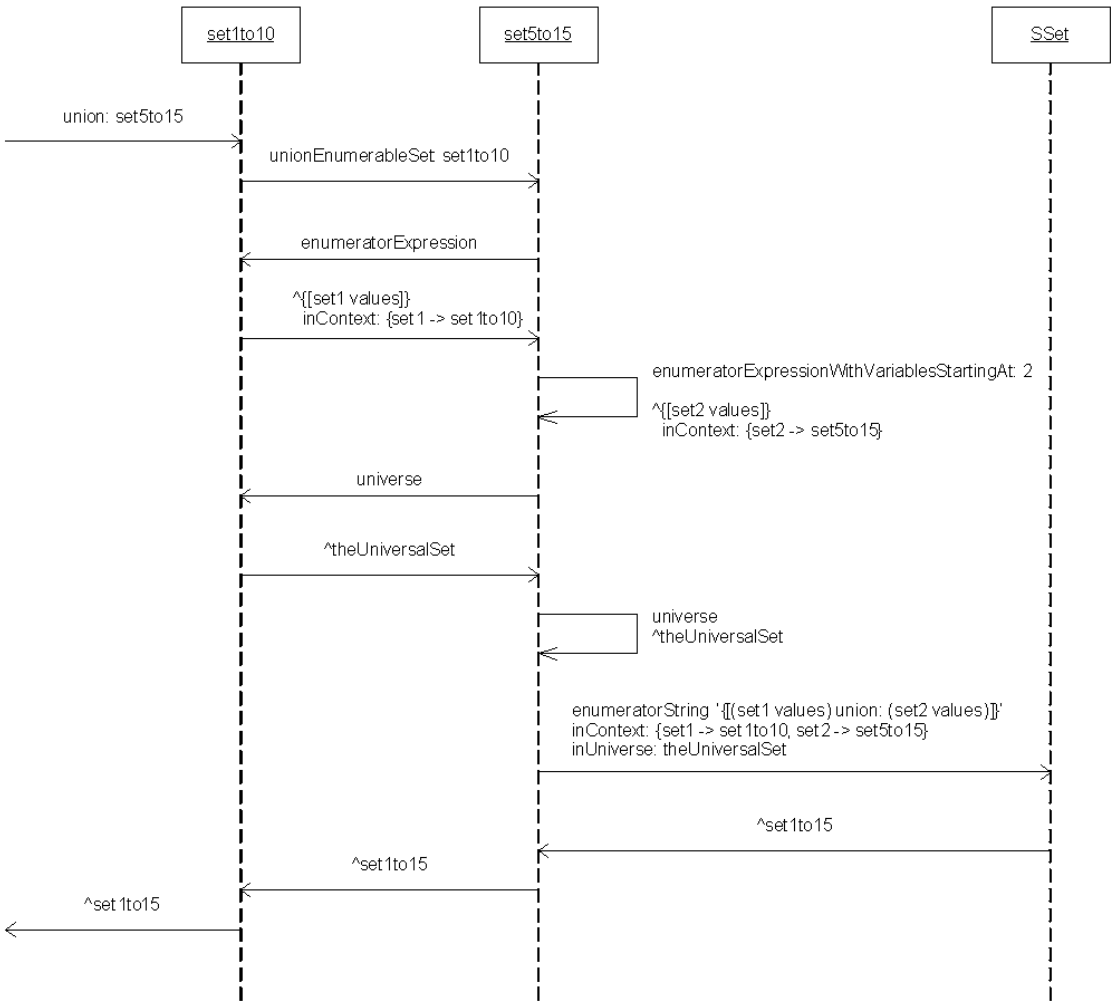
AUB: la unión de los conjuntos A y B devuelve un conjunto que contiene todo elemento que pertenecía a alguno de los dos conjuntos. El universo de AUB es la unión de los universos de A y B.

Ejemplo:

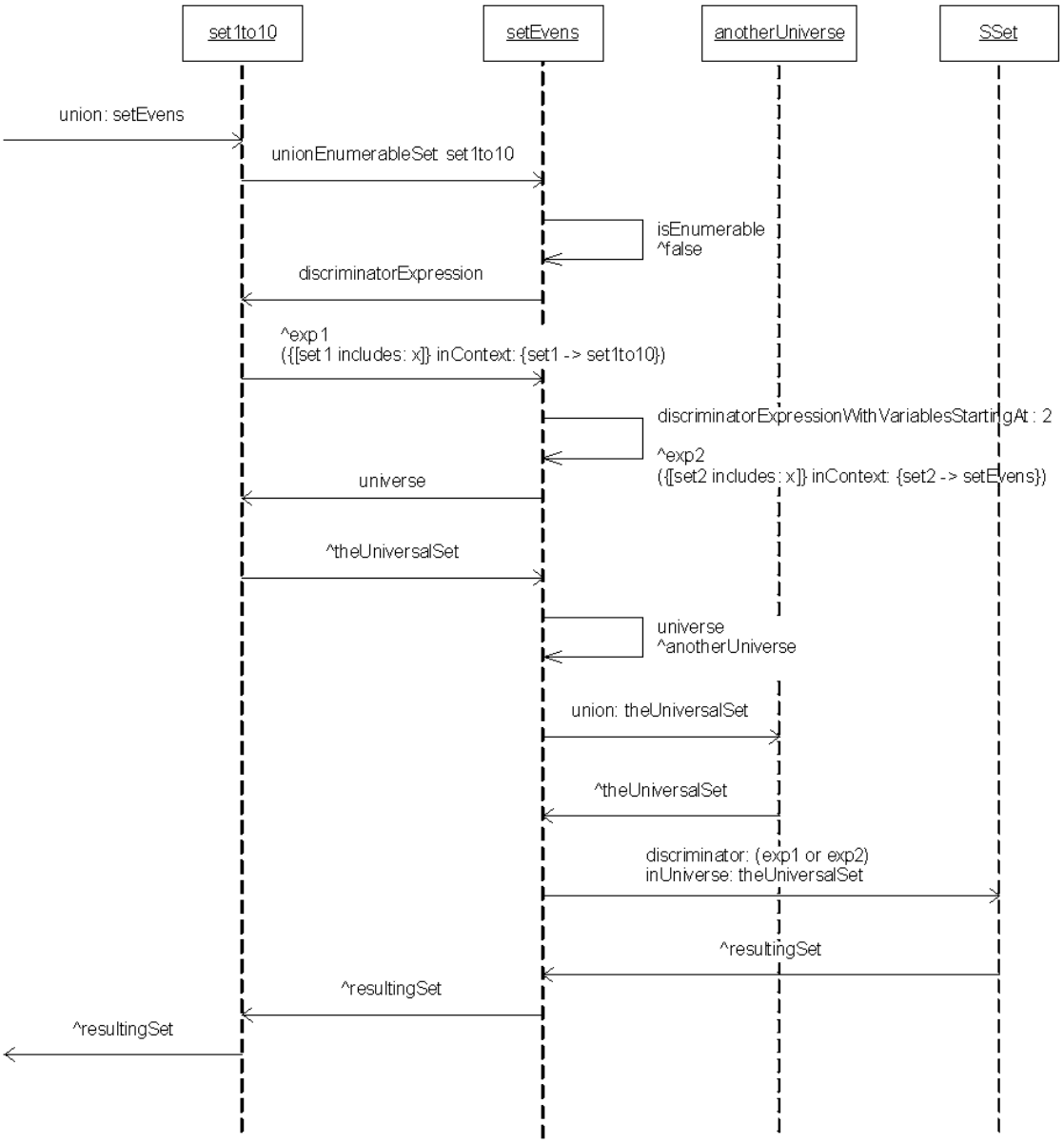
```
setA := '{[1 to:3]}' toSSet.                {1, 2, 3}
setB := '{[3 to:4]}' toSSet.                {3, 4}
setC := setA union: setB.                         {1, 2, 3, 4}
```

Los siguientes diagramas muestran otros casos de Union y cómo los resuelve el framework internamente:

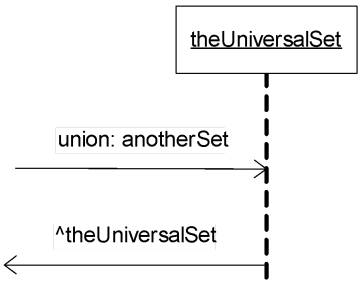
- Unión de los conjuntos enumerables de números del 1 al 10 y 5 a 15.
- En este caso el universo de ambos conjuntos es el conjunto universal, que queda como universo del conjunto resultante.

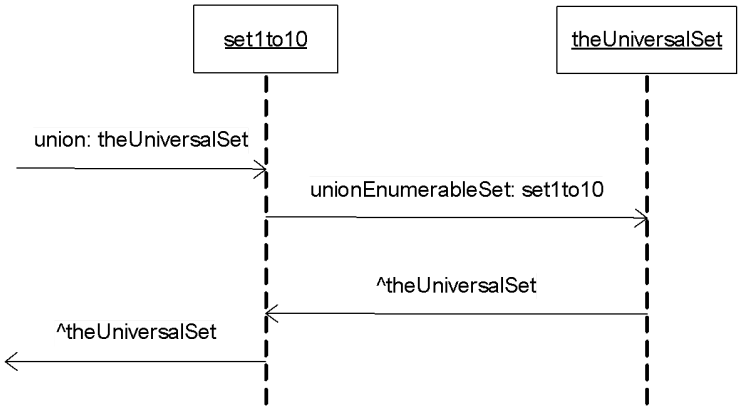


- Unión del conjunto de números del 1 al 10 y el conjunto de números pares que tiene como universo otro conjunto de números no enumerable.
- El universo del conjunto resultante es la unión de los universos de los conjuntos con los que se opera, que en este caso da como resultado el conjunto universal.

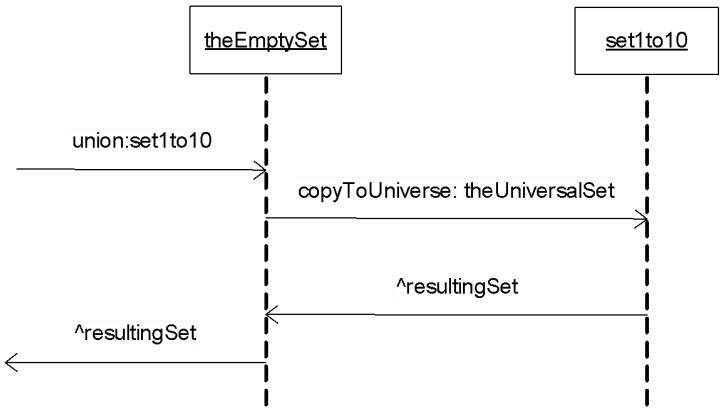


- Unión con el conjunto universal, que dará como resultado el conjunto universal.





- Unión con el conjunto vacío, el double-dispatch funciona de manera análoga al conjunto universal.
- El resultado es el mismo conjunto con el cuál se une al conjunto vacío, pero con el conjunto universal como universo. Esto se debe a que el universo del resultado debe ser la unión de los universos de los dos operandos.



Intersección

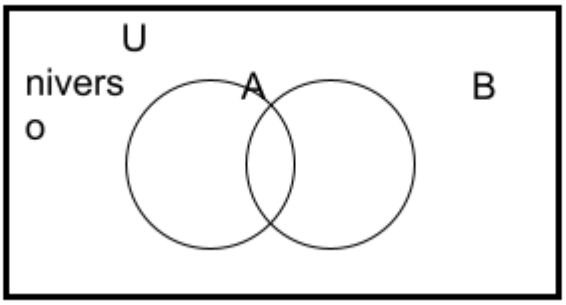
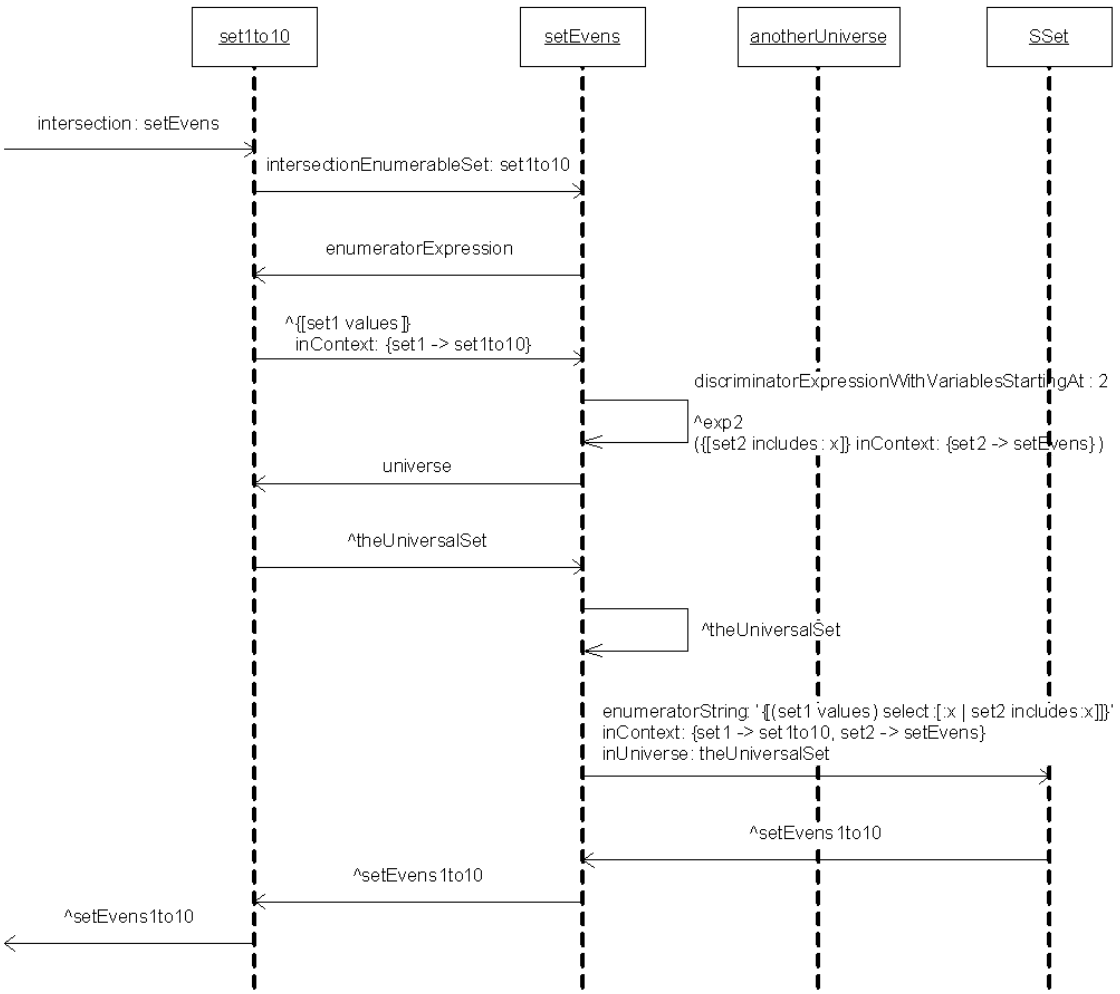
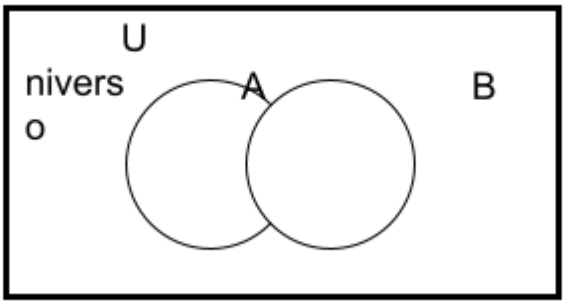


Figura: el área cuadriculada representa A∩B

A∩B: la intersección de los conjuntos A y B devuelve un conjunto que contiene todo elemento que pertenecía a ambos conjuntos. El universo de A∩B es la unión de los universos de A y B dado que deseamos que este contenga todos los elementos de interés. Si fuese la intersección, lo que uno podría intuir como primera opción, podrían quedar elementos de A o de B fuera del universo de trabajo.

Ejemplo:

- ```
  setA := '{[1 to:3]}' toSSet.                    {1, 2, 3}
  setB := '{[3 to:4]}' toSSet.                    {3, 4}
  setC := setA intersection: setB.                {3}
  ```

La resolución de la operación de intersección es muy parecida a la unión, por lo que sólo mostraremos un caso:
  ```

- Intersección del conjunto de números de 1 a 10 con el conjunto de números pares.



Diferencia



Figura: el área rayada representa A-B

A-B: la diferencia entre los conjuntos A y B devuelve un conjunto que contiene todo elemento que pertenecía a A pero no a B. El universo de A-B es el mismo que el de A.

Ejemplo:

- ```
  setA := '{[1 to:3]}' toSSet.                {1, 2, 3}
  setB := '{[3 to:4]}' toSSet.                   {3, 4}
  setC := setA difference: setB.              {1, 2}
  ```

Los diagramas de secuencia para la diferencia entre conjuntos son muy parecidos a las operaciones anteriores, con las siguientes diferencias:

- Si se sustrae de un conjunto no enumerable, entonces el resultado será un Intentional Set cuya definición responderá a la formula:

```
{([set1 includes: x] and not [set2 includes: x])}
```

- Si se sustrae de un conjunto enumerable, el resultado será un Enumerable Set cuya definición tendrá la forma:

```
{[(set1 values) reject: [:x | set2 includes: x]]}
```
```

Complemento

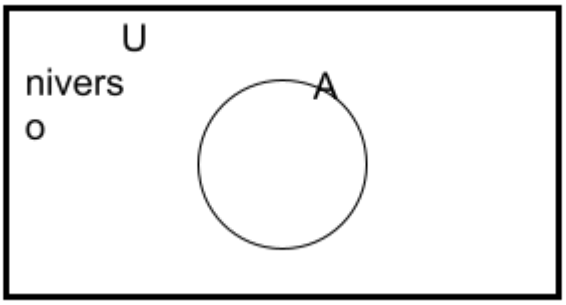


Figura: el área rayada representa Ac

A^c: el complemento de A es el conjunto formado por todos los elementos del universo que no pertenecen a A. El universo de A^c es el mismo que el de A.

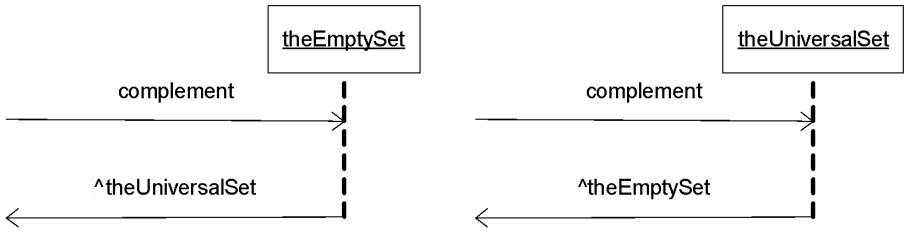
Ejemplo:

```
setU := '{[1 to:5]}' toSSet.                    {1, 2, 3, 4, 5}
setA := '{[1 to:3]}' toSSetInUniverse: setU.       {1, 2, 3}
setC := setA complement.                        {4, 5}
```

La implementación de la operación Complemento es un poco diferente a las demás, ya que se basa en la Diferencia. Esto permite que esté implementada a nivel de la clase `SSet`.

- Complemento del conjunto de números del 1 al 3 que tiene como universo a los números del 1 al 5.

- Complemento de los conjuntos vacío y universal.



Producto

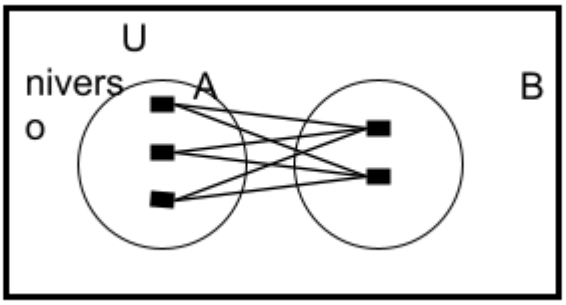


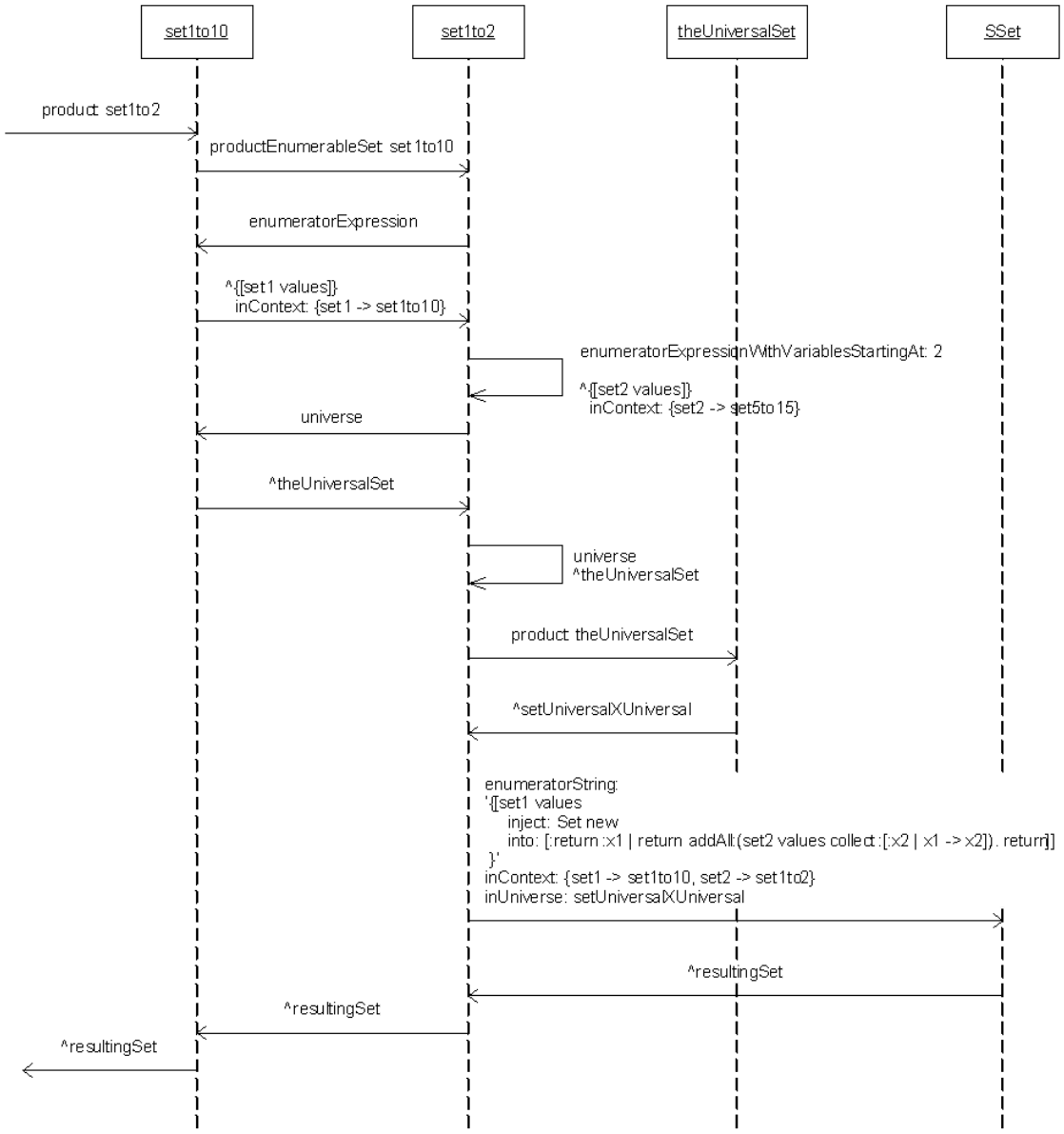
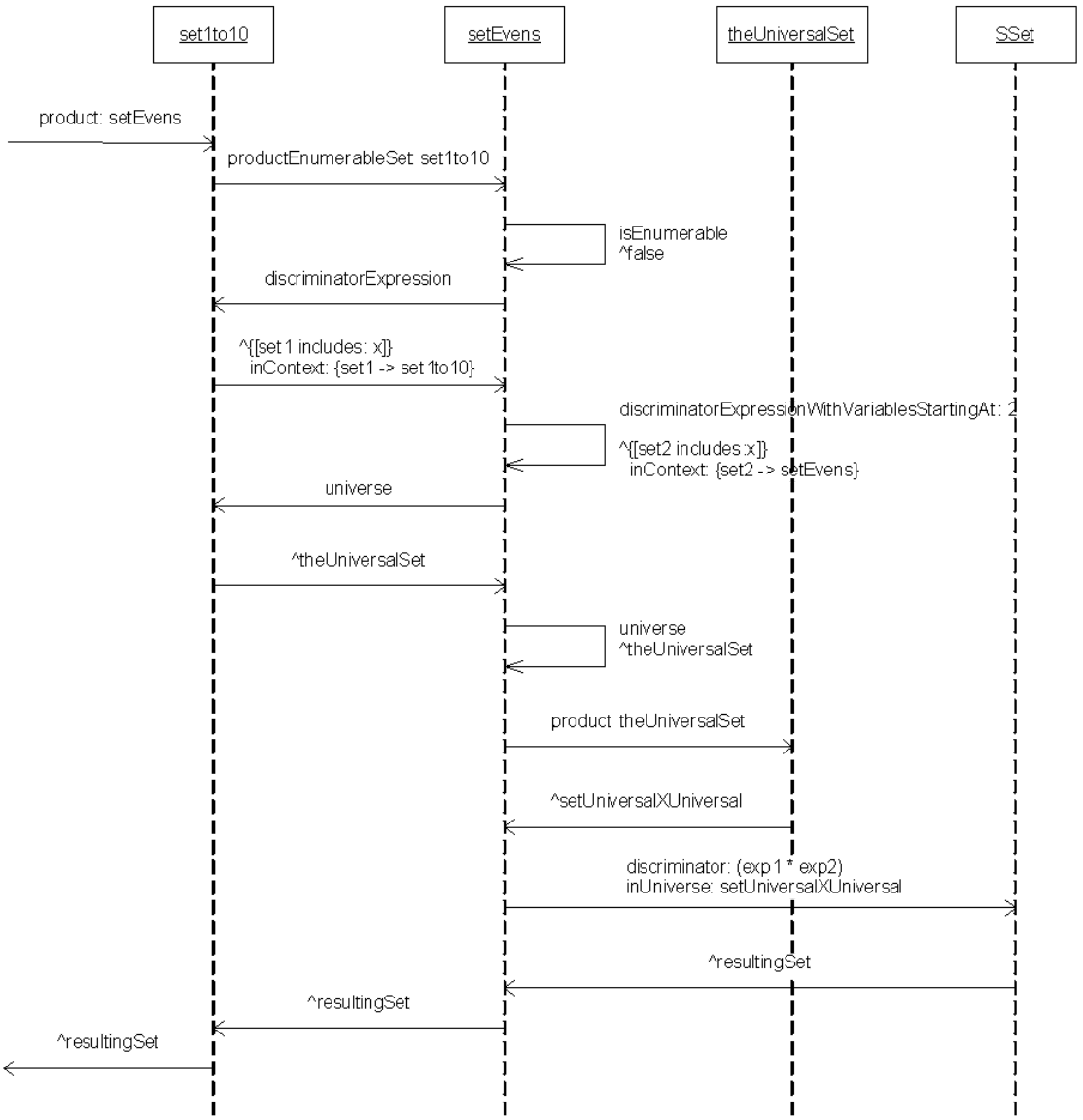
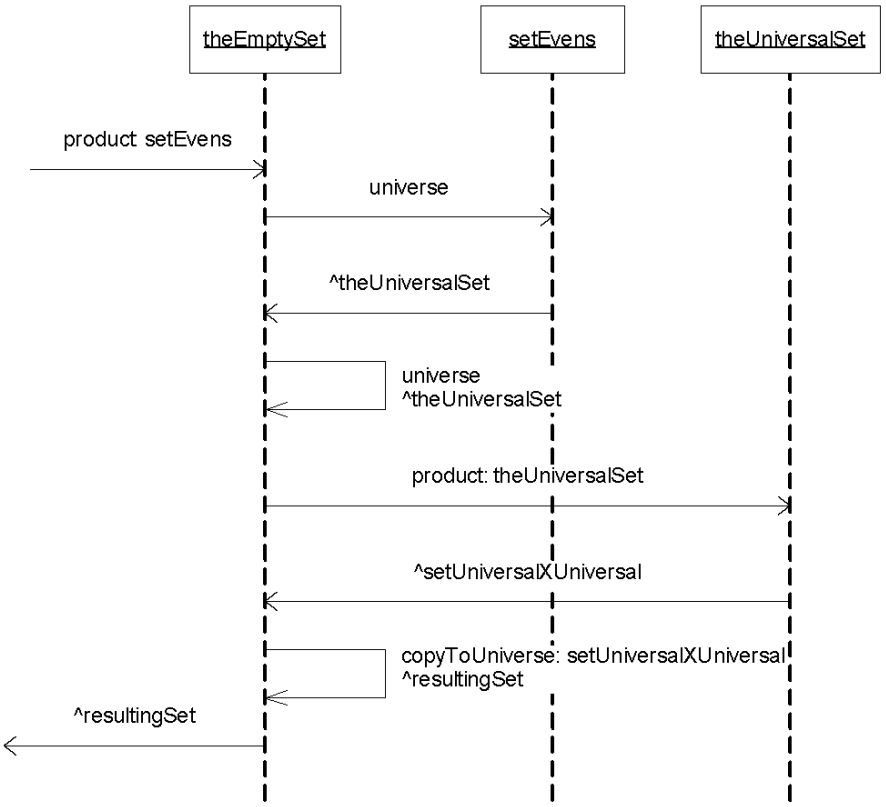
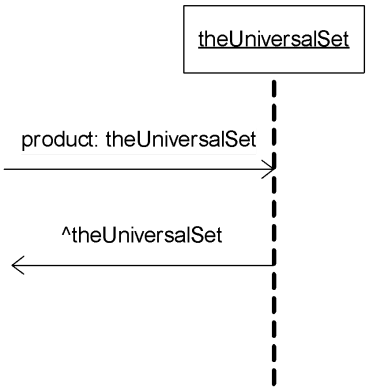
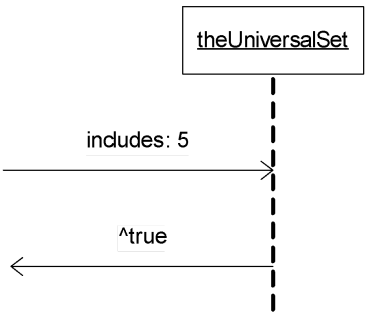
Figura: las líneas representan los pares ordenados de AxB

AxB: el producto de A y B devuelve un conjunto que contiene como elementos a todos los pares ordenados cuya primera coordenada pertenece a A y la segunda a B. El universo de AxB es el producto de los universos de A y B.

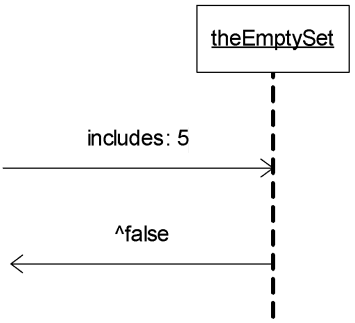
Ejemplo:

- ```
  setA := '{[1 to:2]}' toSSet.                    {1, 2}
  setB := '{[3 to:4]}' toSSet.                    {3, 4}
  setC := setA product: setB.
                        {(1, 3), (1, 4), (2, 3), (2, 4)}
  ```

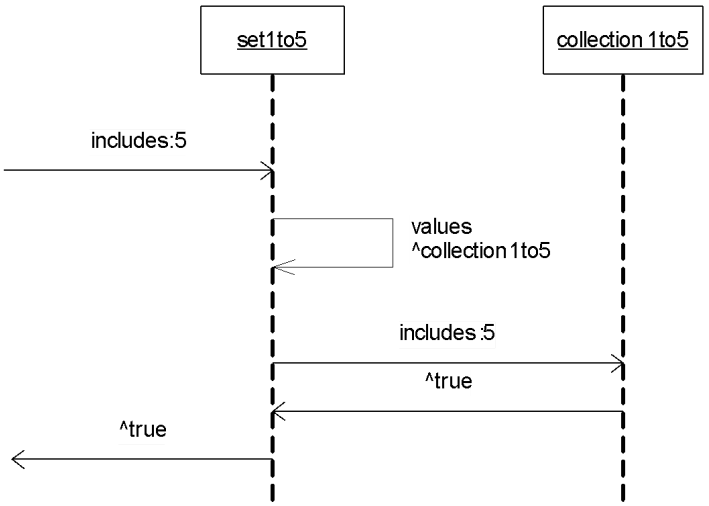
La implementación del producto implica cambiar la cardinalidad del conjunto resultante, que queda formado por duplas con un elemento de cada uno de los conjuntos con los que se opera.

Si los dos conjuntos son enumerables, el resultado será un Enumerable Set. En caso contrario será un Intentional Set.
  ```

- Producto de los conjuntos enumerables de números del 1 al 10 y del 1 al 2.



- Producto del conjunto no enumerable de números pares, por el conjunto de números del 1 al 10.



- Producto del conjunto de los pares con el conjunto vacío.



- Producto del conjunto universal consigo mismo.



El producto del conjunto universal con otro conjunto funciona exactamente igual que cuando se opera con un conjunto no enumerable.

Pertenencia de elementos al conjunto

El protocolo {#includes:, …}

Este protocolo hace posible testear la pertenencia de elementos a un conjunto. Los métodos que lo conforman son:

#includes: recibe un elemento como parámetro y retorna si pertenece o no al conjunto.

#includesAllOf: recibe una colección de elementos como parámetro y retorna si todos pertenecen al conjunto o no.

#includesAnyOf: recibe una colección de elementos como parámetro y retorna si al menos uno de ellos pertenece al conjunto o no.

Ejemplos:

- ```
  setEvens := '{[x even]}' toSSet.
  setEvens includes: 1.                  false
  setEvens includes: 6.                  true
  setEvens includes: 'a'.                false
  setEvens includesAllOf: #(2 3).        false
  setEvens includesAnyOf: #(2 3).        true
  ```

Los siguientes diagramas explican la implementación de #includes:. Este método testea la pertenencia de un elemento al conjunto. Es abstracto en la clase `SSet` y cada tipo de conjunto provee su implementación:

- `Testeando si el número 5 pertenece al conjunto universal`


  ```

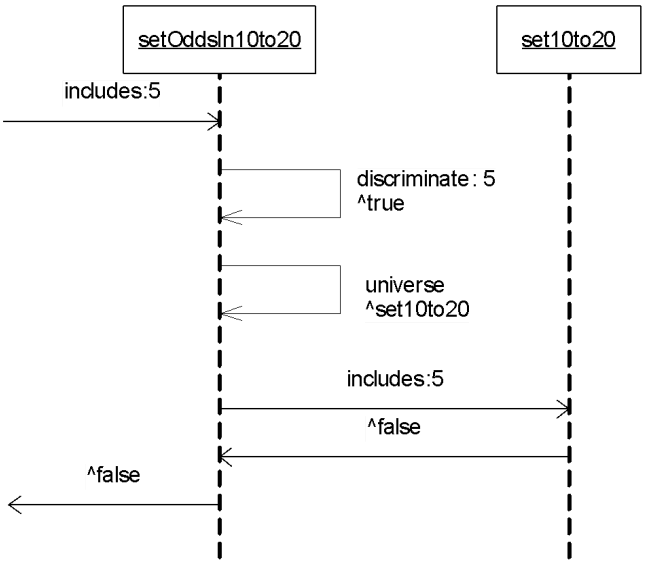
- Testeando si el número 5 pertenece al conjunto vacío

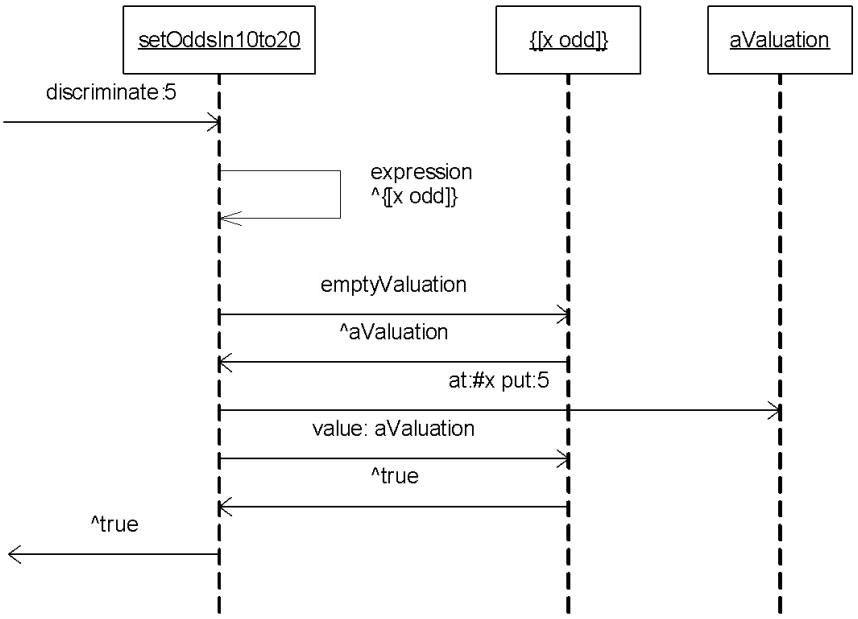


- Testeando si el número 5 pertenece al conjunto de números del 1 al 10



- Testeando si el número 5 pertenece al conjunto de números impares con universo los números del 10 al 20.





Enumeración de los elementos de un conjunto

El método `#isEnumerable` permite testear si un conjunto es o no enumerable. La enumerabilidad de un conjunto está dada exclusivamente por cómo se lo definió. Un conjunto será enumerable si se cumple alguna de las siguientes condiciones:

- Su universo es enumerable
- Es un Enumerable Set, o sea, se lo definió a partir de una expresión que retorna una colección de elementos.
- Es el resultado de alguna operación con conjuntos y la enumerabilidad de los operandos hace posible que el conjunto resultante sea enumerable.

Para los conjuntos enumerables, pueden utilizarse los siguientes métodos:

#values: retorna una colección con los elementos del conjunto
#isEmpty: indica si el conjunto contiene o no elementos
#size: devuelve la cantidad de elementos del conjunto

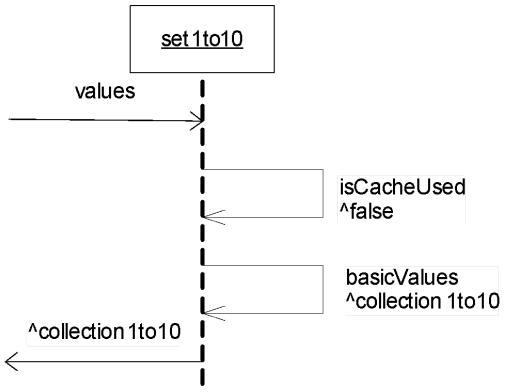
Si se intenta envía cualquiera de estos mensajes a un conjunto no enumerable, el framework disparará una excepción de tipo `CannotEnumerateValuesError`.

Implementación

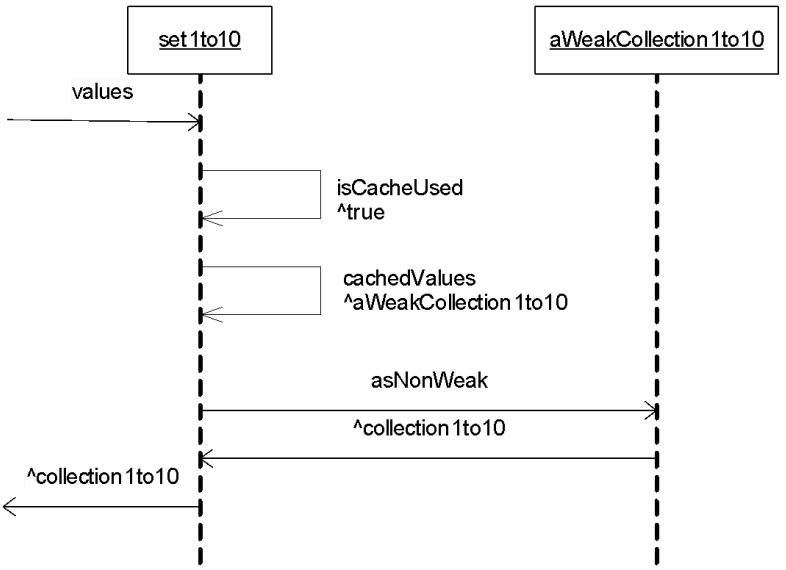
El método que se utiliza para obtener los elementos de un conjunto es `#values`. Este método está en `SSet` y lo único que hace en realidad es el manejo del cache de elementos del conjunto.

Un conjunto puede trabajar con cache o no, dependiendo de si el conjunto tiene un *change detector* instalado. La idea es que si hay un change detector, entonces los elementos del conjunto no cambiar a menos que este avise, por lo que es seguro mantener el cache. Si no se usa change detector, entonces los elementos del conjunto son computados cada vez que se los necesita. La tarea de obtener los elementos del conjunto propiamente dicha es delegada a `#basicValues`, que es reimplementado por los distintos tipos de conjunto.

- `Se piden los valores del conjunto de números del 1 al 10, sin haber instalado un change detector en el conjunto.`



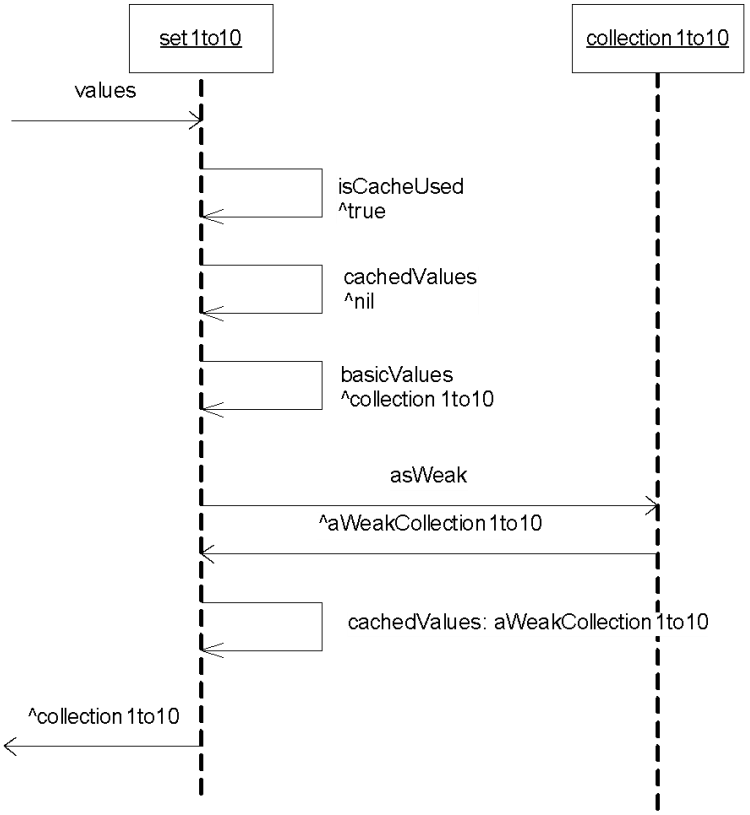
- Se piden los valores del conjunto de números del 1 al 10, teniendo instalado un change detector en el conjunto. En este caso, los elementos del conjunto estaban previamente cacheados.



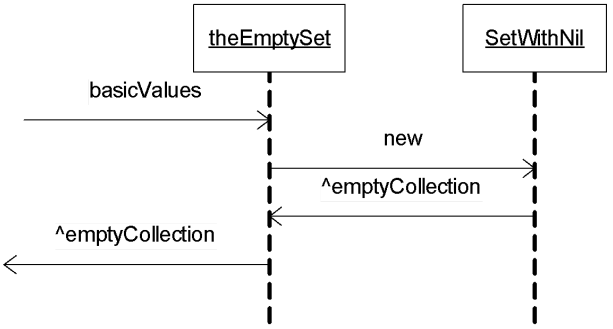
Nota: los elementos se cachean utilizando weak collections, que es un tipo de colección que no mantiene referencias a los elementos que contiene, por lo que estos aparecen como no referenciados para Smalltalk y entonces el garbage collector puede recolectarlos si es necesario (recordemos que los conjuntos en SSets no son contenedores de elementos). Si esto no fuese así, los elementos incluidos en los conjuntos permanecerían siempre en memoria, en uso por el cache del conjunto, aún si nadie más los utilizase.

Para devolver el resultado de `#values`, dicha colección se tranforma en una colección normal y quien la utilice mantendrá referencias a los objetos devueltos, evitando así que el garbage collector los borre mientras están en uso.

- Se piden los valores del conjunto de números del 1 al 10, teniendo instalado un change detector en el conjunto. En este caso, los elementos del conjunto NO estaban previamente cacheados.



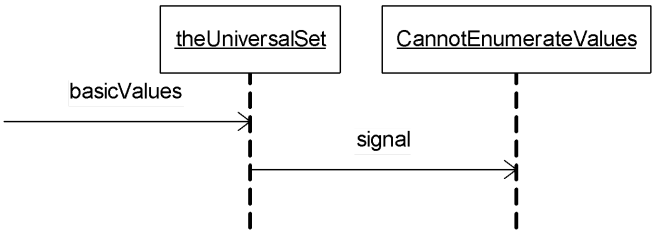
- #basicValues en el conjunto vacío



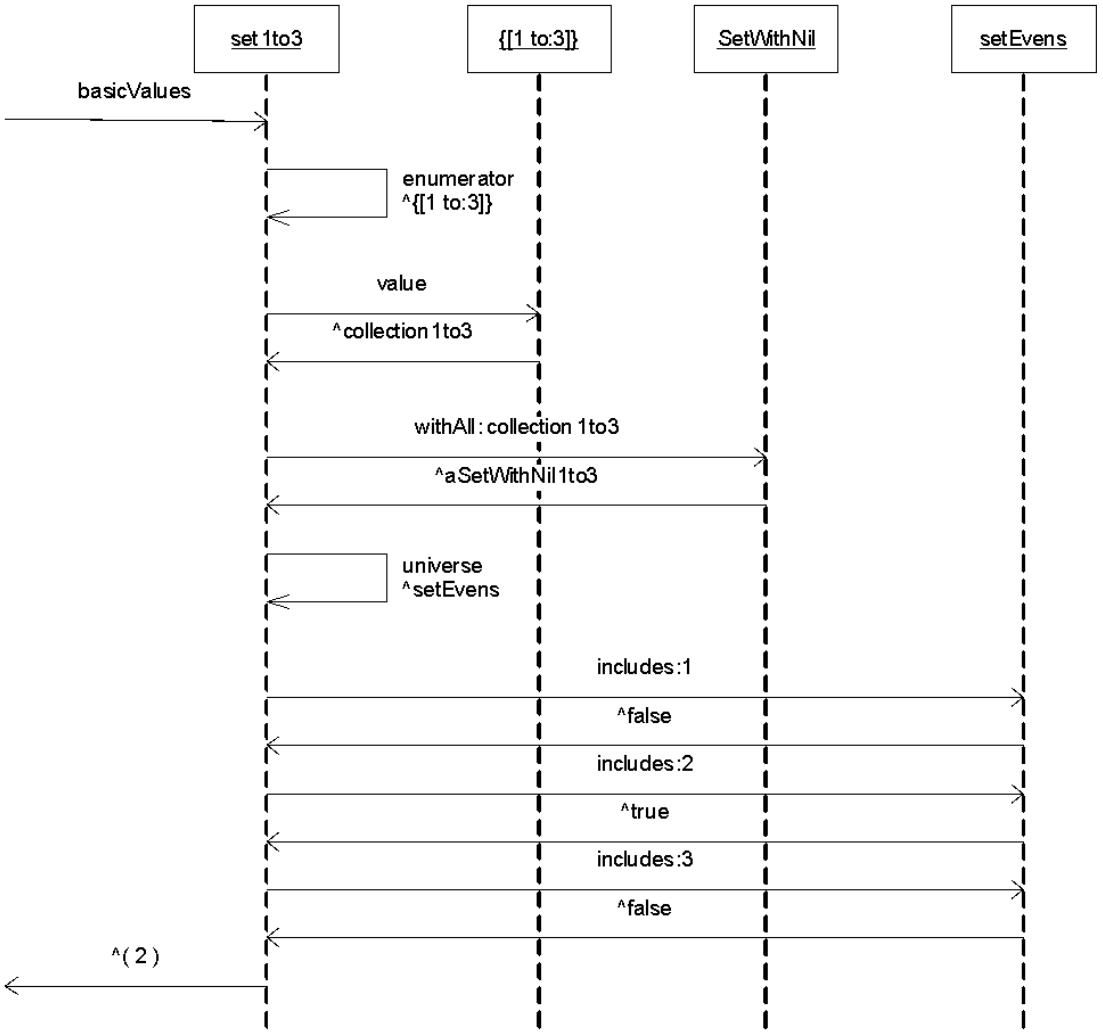
Nota: Para los conjuntos vacío y universal, no se utiliza cache de elementos, por lo que `#values` invoca directamente a `#basicValues`.

Nota: *`SetWithNil`* es una clase que tuvimos que implementar imitando la clase `Set` standard de Smalltalk, pero que soporta contener a nil como elemento.

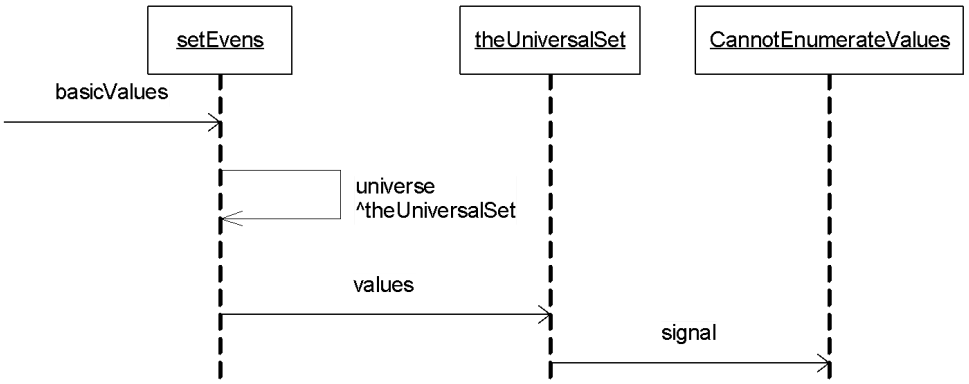
- #basicValues en el conjunto universal
- El conjunto universal es no enumerable, por lo que dispara una excepción si se trata de enumerar sus elementos.



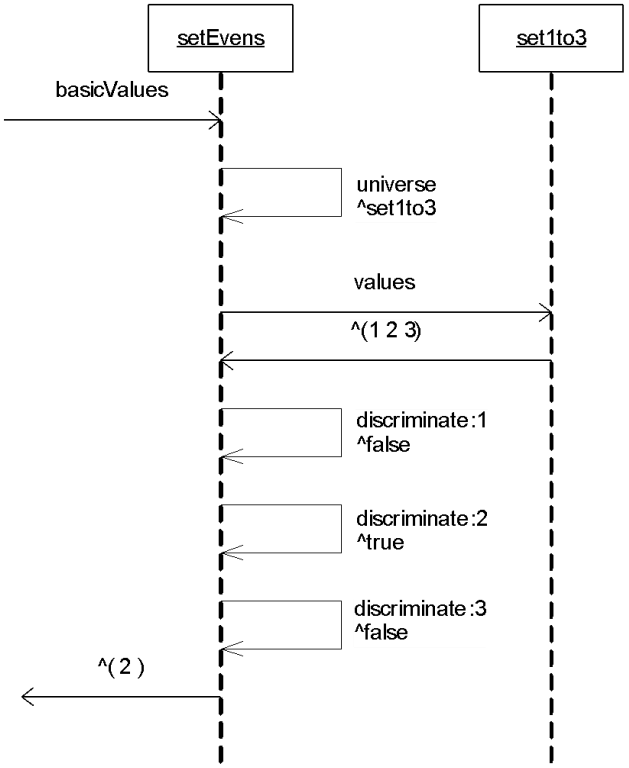
- #basicValues en el Enumerable Set de números del 1 al 3, con universo en el conjunto de números pares.



- #basicValues en el Intensional Set de números pares.
- El conjunto no es enumerable, por lo que dispara una excepción.



- #basicValues en el Intensional Set de números pares, pero esta vez el conjunto tiene de universo al conjunto de números del 1 al 3.



Compatibilidad con Collection

Sin duda las colecciones son uno de los objetos más útiles y utilizados en Smalltalk. Todo programador con algo de experiencia en Smalltalk conoce al menos su protocolo principal y hay muchas clases ya construidas que están diseñadas para colaborar con objetos polimórficos a las colecciones.

Decidimos entonces que los conjuntos implementen el protocolo de `Collection`, y de esta manera puedan ser utilizados en muchos más lugares.

Lo métodos de `Collection` son muy útiles para trabajar con los elementos de los conjuntos enumerables, pero carecen de sentido cuando el conjunto en cuestión no lo es. En estos casos se disparará la excepción `CannotEnumerateValuesError` si se los intenta utilizar.

Los métodos de `Collection` que implementamos por ahora son:

- ***#do***
- ***#collect:***
- ***#select:***
- ***#reject:***

Iremos implementando los demás métodos a medida que se vayan necesitando.

Resta decir que no queremos entrar en este trabajo en una discusión conceptual acerca de qué es una `Collection`, ni de cómo debería estar implementada la jerarquía `Collection` en Smalltalk. Actualmente hay consenso en la comunidad de Smalltalkers acerca de que dicha implementación no es óptima y hay incluso publicados varios trabajos acerca de cómo debería ser rediseñada y mejorada (ver Strongtalk por ejemplo, [BRA96]). Si en algún momento se acepta alguno de estos rediseños como el estándar de Smalltalk, sería interesante estudiar cómo encajan los conjuntos por comprensión en este nuevo esquema.

Detección de cambios y actualización de la pertenencia al conjunto

Como mencionamos anteriormente, SSets implementa un mecanismo por el cual los conjuntos detectan y avisan cuando cambian sus elementos. El mecanismo se compone de dos partes:

- Suscripción al conjunto de los objetos a los cuales se debe notificar
- Detección de los cambios en el conjunto

Suscripción de objetos a los cuales notificar

Squeak trae incorporada una implementación del pattern Observer [GAMMA], mecanismo mediante el cual un objeto puede observar a otro y ser notificado cuando ocurre un cierto evento. El concepto es que el observador se suscribe a un evento del objeto observado y cuando este evento ocurre, el observador recibe un mensaje.

En SSets utilizamos este mecanismo para que otros objetos puedan enterarse cuando hay un cambio en los elementos de un conjunto. Cuando uno de estos cambios se detecta, el evento que se dispara en el conjunto es el evento `#changed`.

El mensaje `#whenChangedSend:to:` puede enviarse a un conjunto para que un objeto sea notificado de sus cambios. El método `#removeChangeNotifications` hace que el conjunto pare de notificar de los cambios.

Ejemplo:

- `mySet whenChangedSend: #foo to: theObserverObject`

Para más detalles de cómo suscribir objetos a eventos, enviar parámetros en las suscripciones, etc., ver los protocolos `event-registering` y `event-removing` en la clase `Object`.

Detección de los cambios en el conjunto

Ya dijimos que cuando un conjunto detecta un cambio en sus elementos, va a notificarlo a los observadores. Ahora bien, ¿cómo detecta un conjunto los cambios?

Esto es dependiente del conjunto. Por eso el modelo trae incorporada la posibilidad de crear lo que llamamos *Change Detectors*, que son básicamente objetos encargados de detectar los cambios y avisar al conjunto. De esta manera, puede decidirse e implementarse en cada caso la mejor estrategia para la detección de cambios.

Nota: SSets trae implementado un change detector que será utilizado si no se especifica lo contrario, que funciona mediante polling y permite detectar cambios sólo en conjuntos enumerables.

Los siguientes métodos de un conjunto posibilitan el manejo de los change detectors:

#whenChangedSend:to: al enviar este mensaje a un conjunto, además de suscribir al objeto, se instala en dicho conjunto el change detector default si es que no había otro instalado.

#changeDetector: permite instalar un change detector en el conjunto

Para implementar un change detector, el único requisito que hay es respetar la interfase que exigen los conjuntos, implementando lo siguientes métodos:

#set: este mensaje se enviará al change detector con el conjunto como parámetro cuando se lo instale, y con `nil` como parámetro cuando se lo desinstale del conjunto.

#contentsChanged / contentsChangedTo: debe enviarse alguno de estos mensajes al conjunto para notificarlo cuando hubo un cambio en sus elementos. El conjunto se ocupará entonces de avisar a los observadores.

Nota: si quiere implementar un changeDetector, puede tomar como ejemplo las siguientes clases:

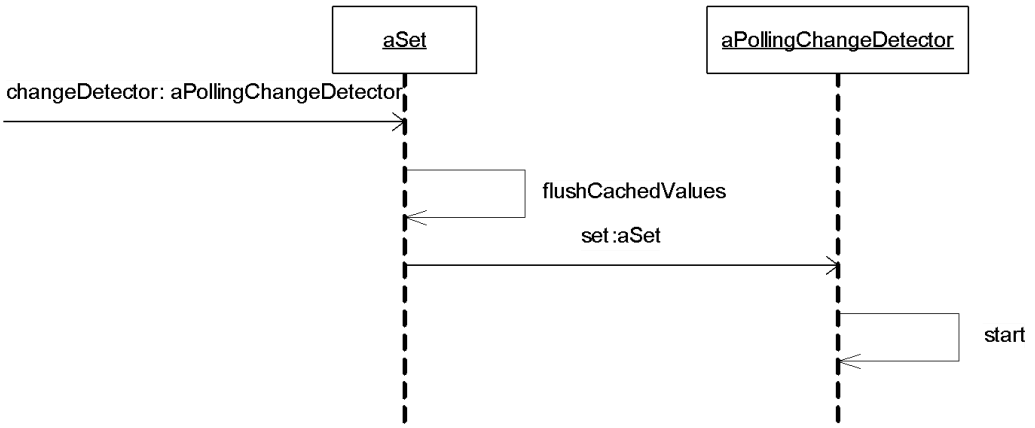
- *SSetPollingChangeDetector* change detector default del framework.
- *MClassChangeDetector* change detector implementado en Mustache (ver más adelante) que funciona mediante pushing.

Implementación

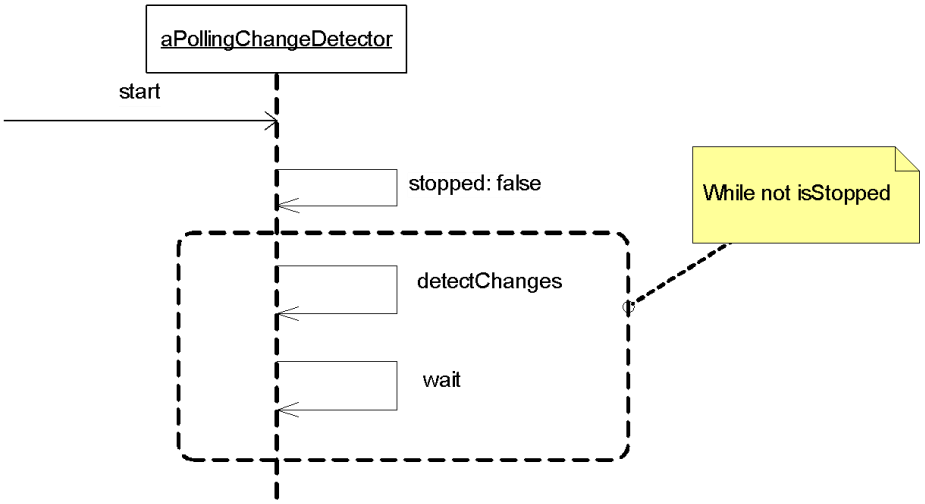
A continuación mostramos el diseño del *Polling Change Detector*, el change detector que se utiliza por default en los conjuntos y que viene implementado en el framework.

Está diseñado para detectar cambios en los conjuntos enumerables solamente. Para los conjuntos no enumerables, deberán implementarse change detectors específicos a cada caso.

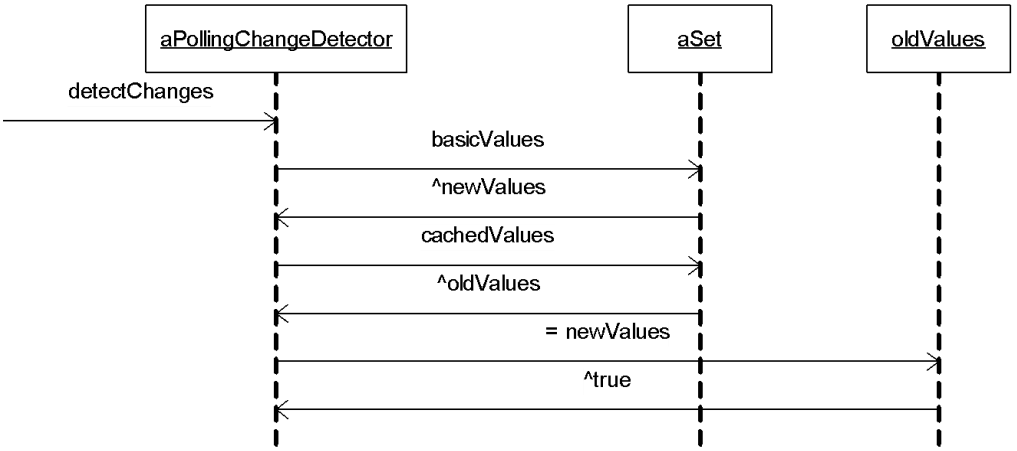
- `Un polling change detector es asignado al conjunto aSet.`



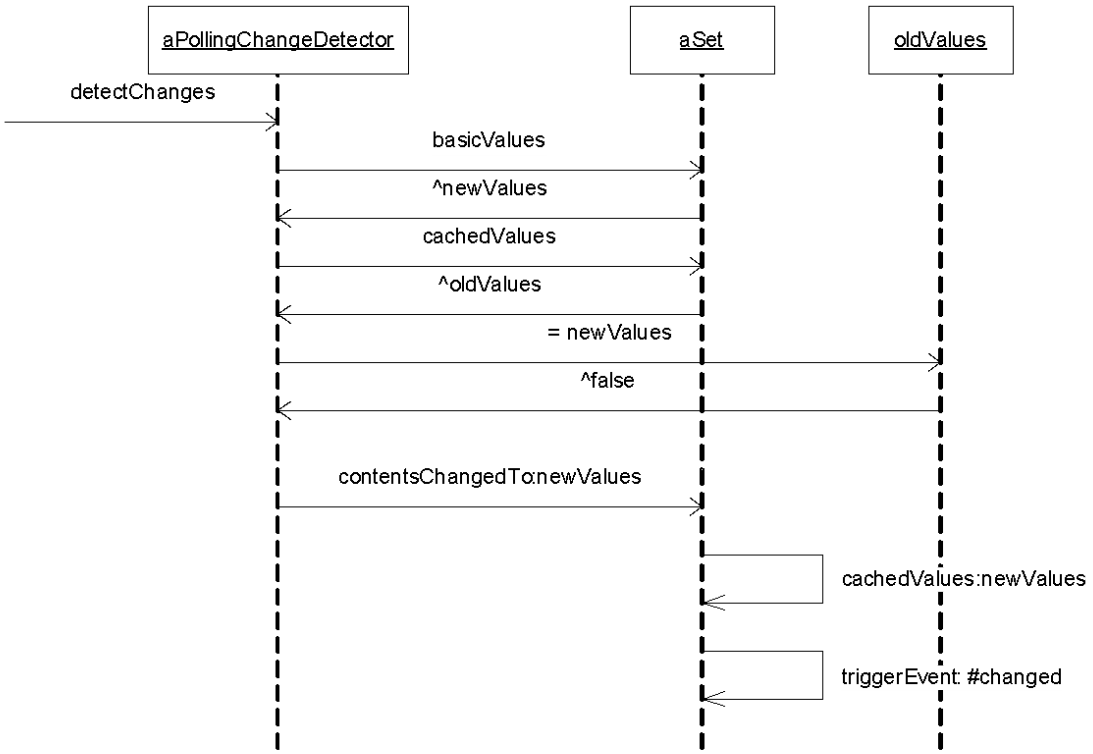
- El mensaje #start inicia el polling



- El change detector se fija si el contenido del conjunto cambió, y no cambió



- El change detector se fija si el contenido del conjunto cambió, y sí cambió, por lo que el conjunto dispara el evento #changed



Herramientas Desarrolladas

Como parte de SSets, desarrollamos una serie de herramientas gráficas que permiten al usuario interactuar directamente con los conjuntos, eliminando en muchos casos la necesidad de escribir código para ello y favoreciendo la manipulación directa de los objetos del ambiente.

Las herramientas están pensadas para simplificar la programación exploratoria, ya que permiten operar con los conjuntos fácilmente, posibilitando explorar un dominio y jugar al "what-if" de manera sencilla.

Pensamos además, que pueden ayudar a facilitar la comunicación entre los programadores y los expertos de negocios, dado que estos últimos podrán ahora inspeccionar de manera sencilla los conjuntos del dominio y revisar sus definiciones directamente en las aplicaciones, sin necesitar conocimientos de programación.

SSetInspector

En los ambientes de Smalltalk una de las herramientas más utilizadas durante el desarrollo y debugging de aplicaciones son los inspectores. Por medio de ellos un desarrollador puede tomar cualquier instancia de una clase y consultar el estado de sus variables, como así también modificarlas. Otra funcionalidad es el envío directo de mensajes a la instancia inspeccionada, para, por ejemplo, ejecutar consultas u otra acción sobre la misma.

La figura siguiente muestra un inspector sobre una instancia de la clase `Color`. El panel de la izquierda muestra las variables y el panel de la derecha sus valores. Las entradas `self` y `all inst vars`, no son variables sino entradas especiales que evalúan en el primer caso a la instancia inspeccionada, y en el segundo en un resumen del estado de la misma.

El panel inferior permite ingresar código Smalltalk e interactuar de forma directa con la instancia. La porción de código Smalltalk que muestra la figura toma como contexto de evaluación la instancia inspeccionada. Por eso el keyword `self` en `self isTranslucentColor` resuelve a la instancia misma.

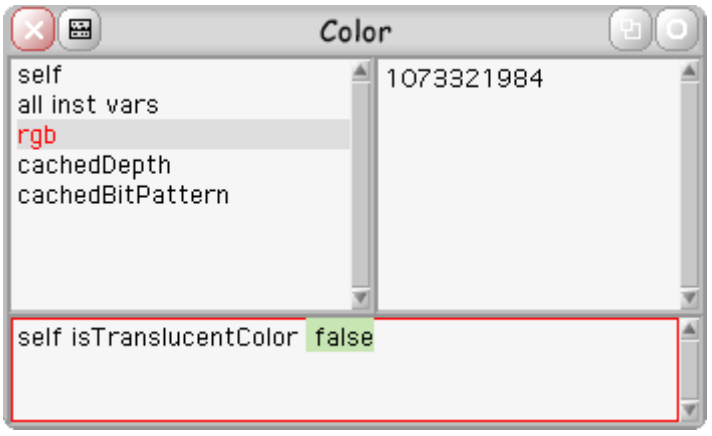


Figura: Un inspector sobre una instancia de la clase `Color`

Inspectores como el anterior proveen una visión de caja blanca sobre los objetos que inspeccionan. Claramente no ocultan su implementación, están pensados para trabajar a nivel meta, es decir para lidiar con objetos en general y su implementación, y no para facilitar la utilización de tipos de objeto específicos por parte del usuario. La figura muestra como `Color` utiliza un `SmallInteger` para codificar la terna RGB con la que internamente la clase representa un color.

En el ambiente de Squeak existen además otros inspectores. La figura siguiente muestra al inspector especializado de la clase `Dictionary`. Este, a diferencia del anterior provee una visión de caja negra. Como se puede observar en la figura la implementación de la clase no se encuentra visible y sólo se provee acceso a las claves del diccionario y a sus valores.

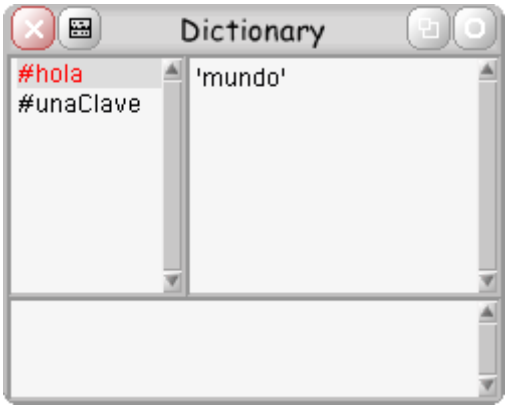


Figura: Un inspector sobre una instancia de la clase `Dictionary`

Para la clase `SSet` hemos desarrollado un inspector del tipo caja negra que provee una interfaz focalizada en semántica de conjuntos y en programación exploratoria.

La figura siguiente muestra una ventana de `SSetInspector` abierta sobre el conjunto `{[1 to: 30]}`.

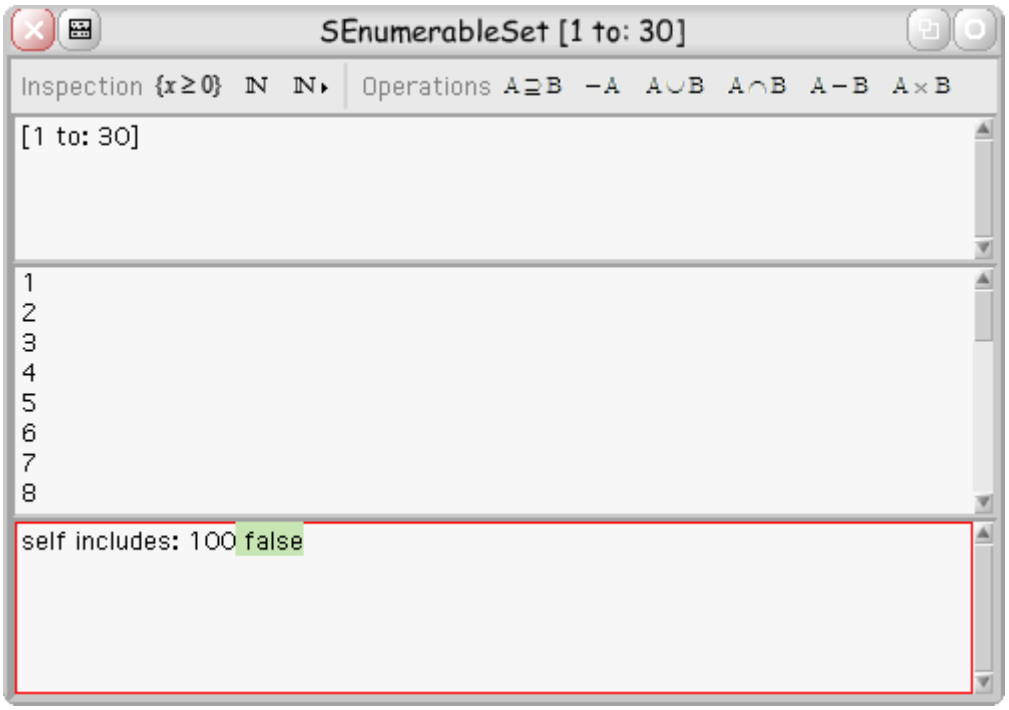


Figura: Un `SSetInspector` sobre el conjunto {`[1 to: 30]`}

De arriba hacia abajo la interfaz de `SSetInspector` se compone de:

- Barra de herramientas con operaciones de conjuntos y de inspección.
- Panel con la definición del conjunto.

- Panel con los elementos del conjunto (cuando este es enumerable).
- Panel de evaluación y de envío de mensajes al conjunto inspeccionado.

Barra de herramientas

La barra de herramientas provee las siguientes operaciones de conjuntos y de inspección:

$\{x \geq 0\}$	Abre un inspector sobre la definición del conjunto.
$\mathbb{N}$	Abre un inspector sobre el universo del conjunto inspeccionado.
$\mathbb{N}_{\blacktriangleright}$	Muestra la cadena de universos del conjunto inspeccionado.
$A \supseteq B$	Abre un inspector sobre un subconjunto del conjunto inspeccionado.
$-A$	Abre un inspector sobre el complemento del conjunto inspeccionado.
$A \cup B$	Abre un inspector sobre el resultado de la unión entre el conjunto inspeccionado y otro.
$A \cap B$	Abre un inspector sobre el resultado de la intersección entre el conjunto inspeccionado y otro.
$A - B$	Abre un inspector sobre el resultado de la diferencia entre el conjunto inspeccionado y otro.
$A \times B$	Abre un inspector sobre el resultado del producto entre el conjunto inspeccionado y otro.

Panel de definición

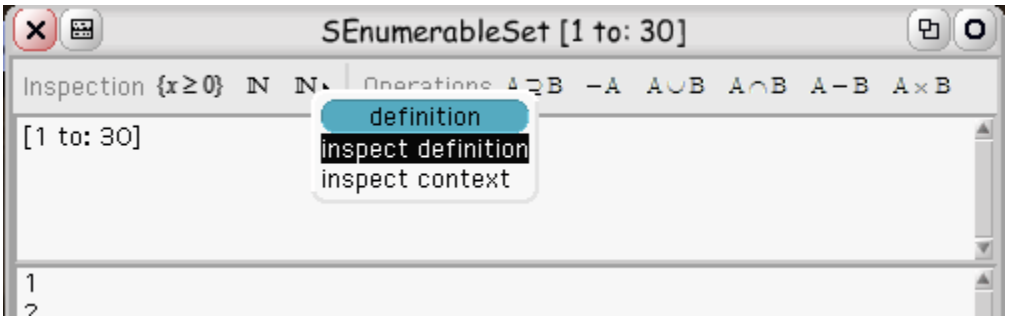


Figura: Panel de definición y su menú contextual

El panel superior en `SSetInspector` muestra la definición del conjunto. La expresión `[1 to: 30]` corresponde a la definición de un Enumerable Set.

El menú contextual en este panel ofrece una opción para abrir un inspector sobre el objeto mismo de definición (en este caso una instancia de `LBlockExpression`) y una opción para inspeccionar el contexto de la expresión.

Panel de elementos del conjunto

Figura: Panel de elementos del conjunto. Izq., con elementos. Der., un conjunto no enumerable.

El panel medio del inspector muestra el contenido del conjunto inspeccionado. Cuando el conjunto no es enumerable en su lugar se muestra el cartel de la derecha.

El widget de lista utilizado es un widget especializado en instancias de `SSet`. Este se describe en la sección `SSetPluggableList`.

Panel de evaluación

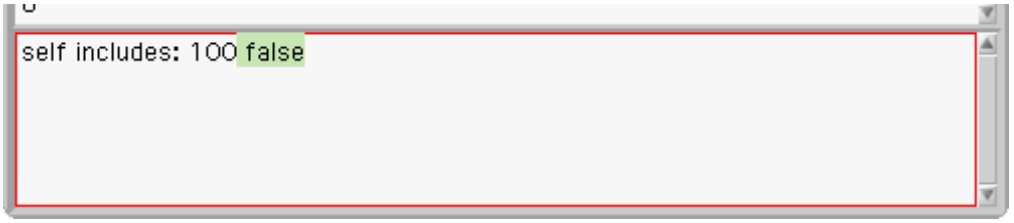


Figura: Panel de evaluación

El panel inferior es el panel de evaluación que se puede encontrar en todos los inspectores del ambiente de Squeak. El código Smalltalk que se ingresa aquí es evaluado con la instancia inspeccionada como contexto de evaluación.

SSetBrowser

Como mencionamos en la sección Programación Exploratoria, creemos que los conjuntos por comprensión son un excelente soporte para esta. El Set Browser fue construido para facilitar la experimentación con conjuntos. Su objetivo es proveer un entorno donde uno pueda jugar, experimentando con diferentes definiciones y operaciones entre conjuntos.

Figura: Una ventaja de SSetBrowser

La ventana de `SSetBrowser` se compone de:

- Una barra de herramientas con operaciones de conjunto.
- Un panel de selección (Panel izquierdo)
- Un panel de inspección (Panel derecho)

El usuario utiliza la barra de herramientas para elegir la operación a realizar. El panel de la izquierda lista los conjuntos con los que se ha operado. El panel de la derecha visualiza la selección.

Operando con SSetBrowser

Seleccionando cualquiera de las operaciones de complemento, unión, intersección, diferencia o producto se abre una ventana de `SSetBrowser` especializada en dicha operación.

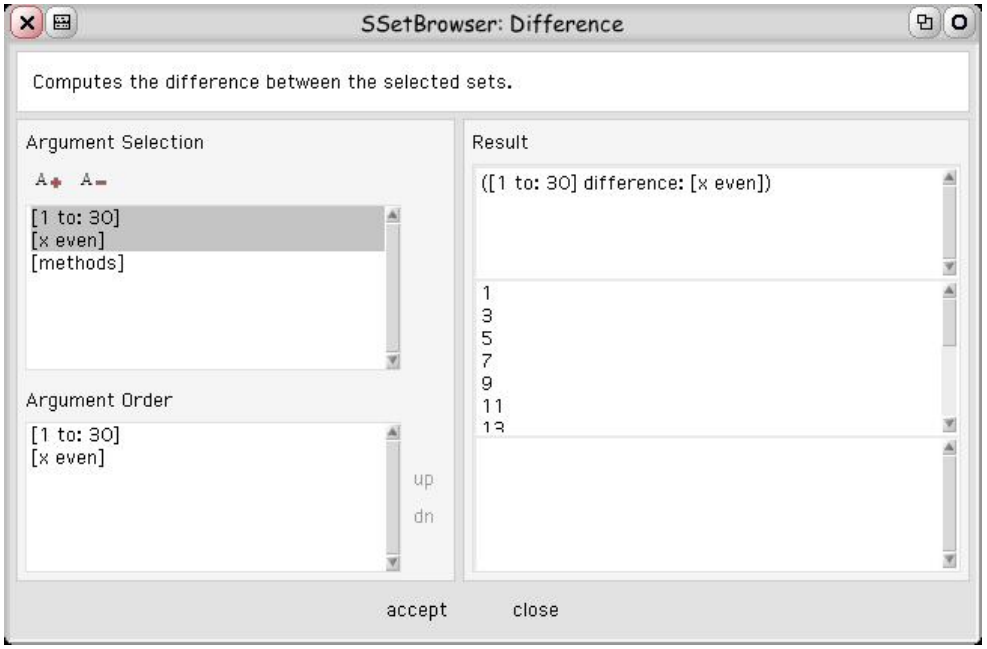


Figura: Ventana de operación de SSetBrowser.

Esta ventana permite al usuario operar con conjuntos de forma interactiva en vez de programáticamente. El panel superior permite seleccionar los argumentos de la operación, mientras que el panel inferior cambiar el orden de estos. El panel derecho previsualiza el resultado de la operación.

SSetPicker

La herramienta `SSetPicker` permite al usuario seleccionar de forma interactiva en vez de programáticamente una instancia de `SSet`.

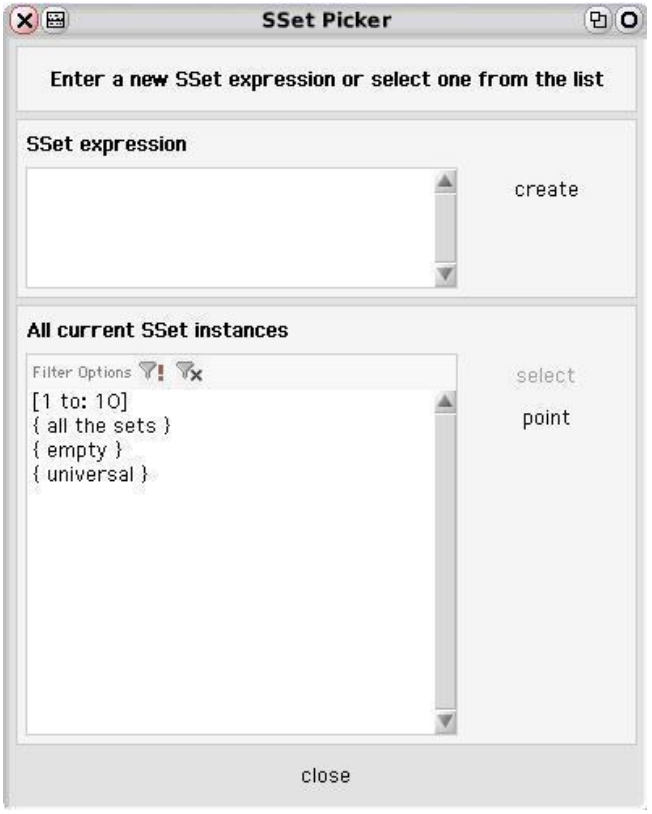


Figura: Una ventana de `SSetPicker`

El panel superior de `SSetPicker` permite crear un conjunto mediante el ingreso de una definición nueva, mientras que el panel inferior lista todas las instancias de `SSet` presentes en el ambiente.

El botón *point* permite al usuario seleccionar la instancia de `SSet` seleccionando con el mouse la aplicación que la usa. Presionando *point* y apuntando a un `SSetInspector` por ejemplo, se seleccionará la instancia inspeccionada. Esto es útil cuando se quiere trabajar con un conjunto que uno está viendo en una herramienta gráfica, porque permite “llevarlo” desde ahí al Set Browser por ejemplo, mediante dos clicks.

Ejemplo:

- `Se selecciona el conjunto {[1 to:30]}, apuntando a un inspector donde se inspeccionaba dicho conjunto`

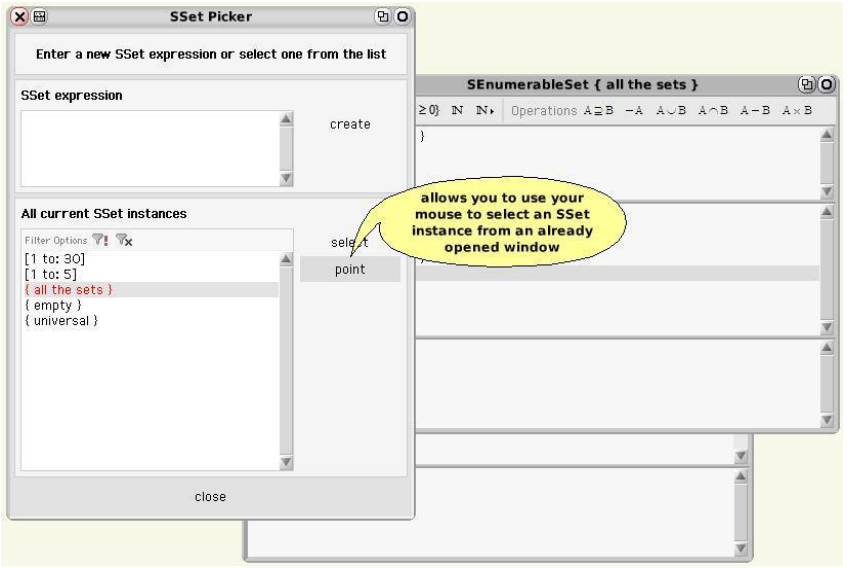


Figura 1: presionando el botón Point

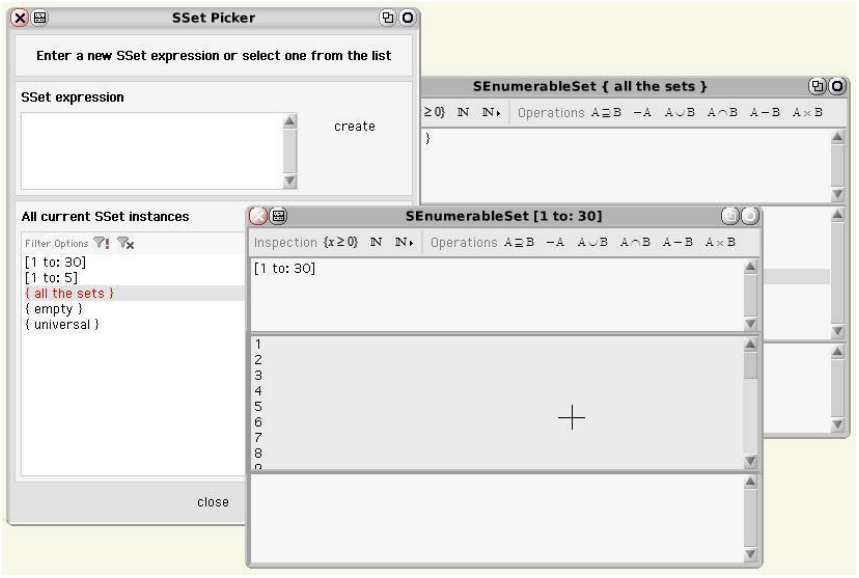


Figura 2: seleccionando la ventana donde se inspeccionaba el conjunto `{[1 to:30]}`

Creemos importante hacer notar que el panel inferior no hace mas que visualizar al conjunto `SSet` definido con la expresión `{[SSet allSubInstances]}`.

Nota: el widget de lista utilizado es una instancia de `SSetPluggableList`, que se describe a continuación.

SSetPluggableList

La clase `SSetPluggableList` provee una implementación de widget de lista especializada en conjuntos.

La misma fue diseñada para soportar programación exploratoria. Con `SSetPluggableList` no sólo es posible visualizar un conjunto sino también explorarlo mediante el poder declarativo de las definiciones por comprensión.

La figura siguiente muestra una instancia de `SSetPluggableList` perteneciente a una ventana de `SSetPicker`. La clase de `SSetPluggableList` es utilizada aquí para visualizar al conjunto definido por enumeración `{[ SSet allSubInstances ]}`.

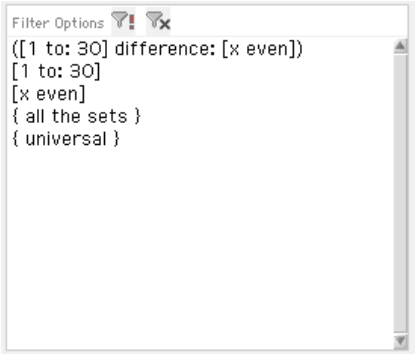


Figura: un `SSetPluggableList` visualizando el conjunto `{[ SSet allSubInstances ]}`

Como cualquier widget de lista convencional `SSetPluggableList` muestra los elementos a visualizar en un área scrolleable con selección simple. `SSetPluggableList` adicionalmente provee operaciones de inspección de los elementos del conjunto, como así también opciones para abrir un `SSetInspector` o un `SSetBrowser`.

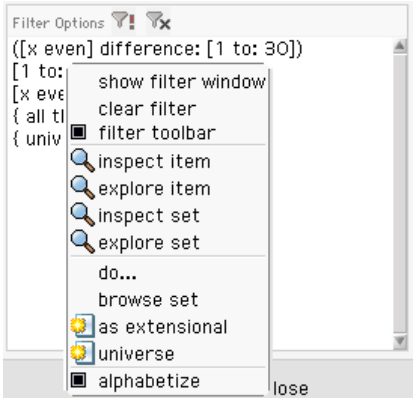


Figura: menú contextual de un `SSetPluggableList`

Filtros

Una de las funcionalidades especiales de `SSetPluggableList` con relación a la Programación Exploratoria es el uso de sus filtros.

Un filtro es una expresión por comprensión que se aplica al conjunto original para visualizar el subconjunto especificado.

Para ilustrar mejor el caso veamos el siguiente ejemplo. La figura siguiente muestra una ventana de Mustache Browser. Mustache es una implementación de un browser de clases como los que se encuentran en cualquier implementación de Smalltalk. Mustache se describirá en detalle en *Anexo I: Experiencia de Uso*.

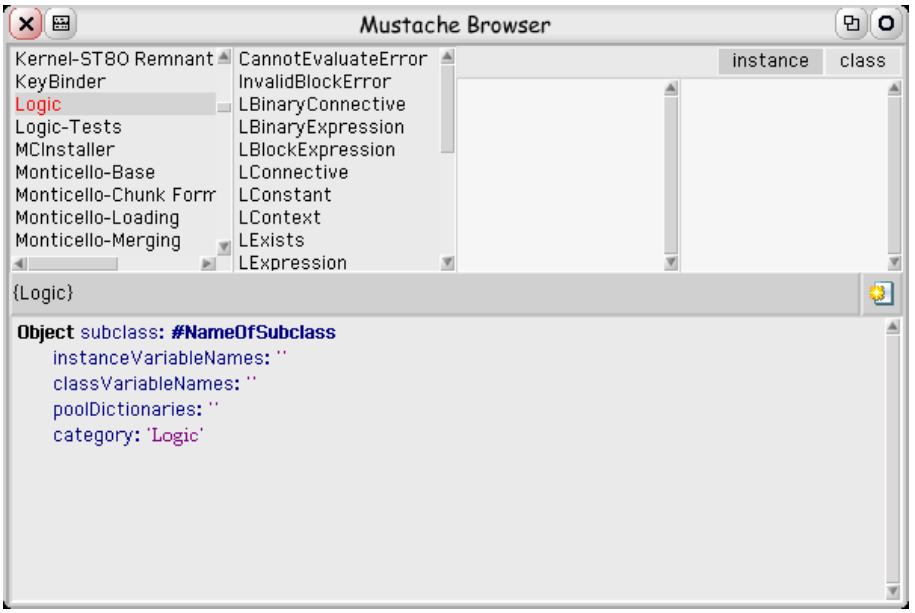


Figura: Una ventana de Mustache Browser abierta sobre la categoría de clases Logic

Cada panel superior de Mustache es una instancia de `SSetPluggableList` visualizando un conjunto de `SSet` específico. La primera columna representa el conjunto de categorías de clase del ambiente de Smalltalk mientras que la segunda al conjunto de clases de la categoría seleccionada.

Supongamos que dadas las clases de la figura deseamos saber cuáles de estas implementan un mensaje en particular. En ese caso podríamos utilizar la siguiente expresión por comprensión como filtro:

```
{[ x canUnderstand: #value:variable:valuation:domains: ]}
```

Esta expresión, aplicada sobre el conjunto anterior, resulta en el subconjunto de clases de la categoría seleccionada, que pueden pueden responder al mensaje `#value:variable:valuation:domains:`. La figura siguiente muestra la instancia de `SSetPluggableList` con el filtro aplicado:

Figura: La parte superior de un Mustache browser mostrando un filtro en el segundo panel.

Vale la aclaración de que en el ambiente de Smalltalk existen herramientas específicas para la ejecución de consultas como la que hicimos. La herramienta convencional de Implementors es un ejemplo de esto. En nuestro ejemplo sin embargo, no tuvimos que valernos de una herramienta específica para dicha consulta. No es difícil imaginarse cómo de la misma manera podríamos haber tomado un conjunto de clientes deudores y haber consultado por aquellos con domicilio comercial en el exterior.

En definitiva, `SSetPluggableList` provee el puente que une la GUI de una aplicación con el poder expresivo y exploratorio de los conjuntos de SSets.

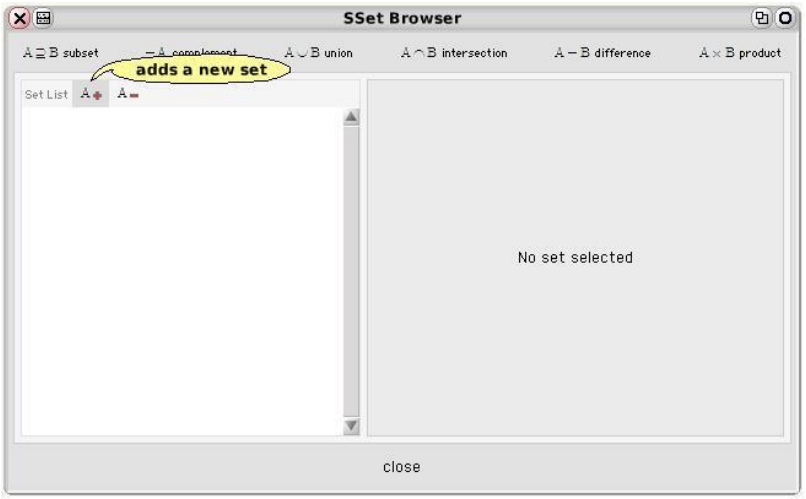
Actualización automática

`SSetPluggableList` utiliza el mecanismo de detección de cambios implementado en el framework para actualizar automáticamente la vista del conjunto.

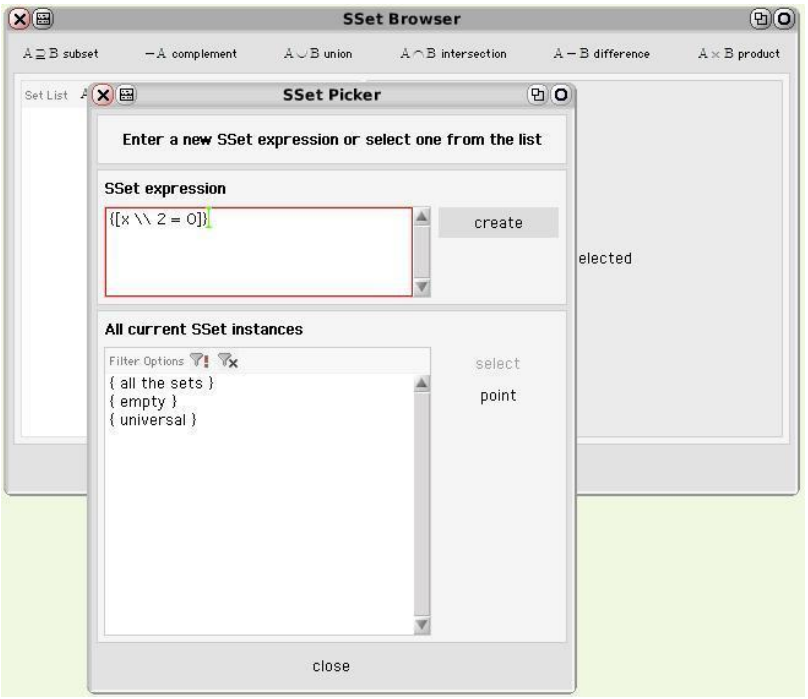
Ejemplo de Uso de las Herramientas

A continuación presentamos un ejemplo del uso de las herramientas. El mismo muestra cómo pueden utilizarse si queremos experimentar con conjuntos y ver el resultado de la intersección entre los conjuntos de números pares y los números del 1 al 10.

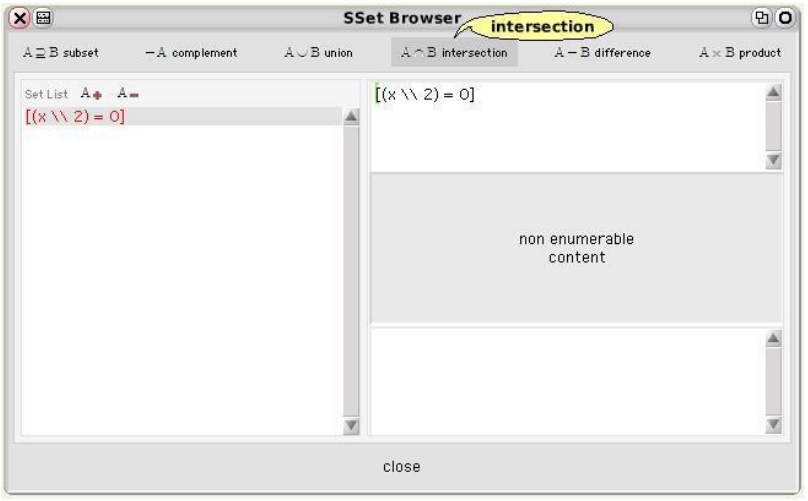
Paso 1: abrimos un `SSetBrowser`, que nos proveerá del entorno donde vamos a experimentar y seleccionamos el botón *A+* para agregar al ambiente el primero de los conjuntos.



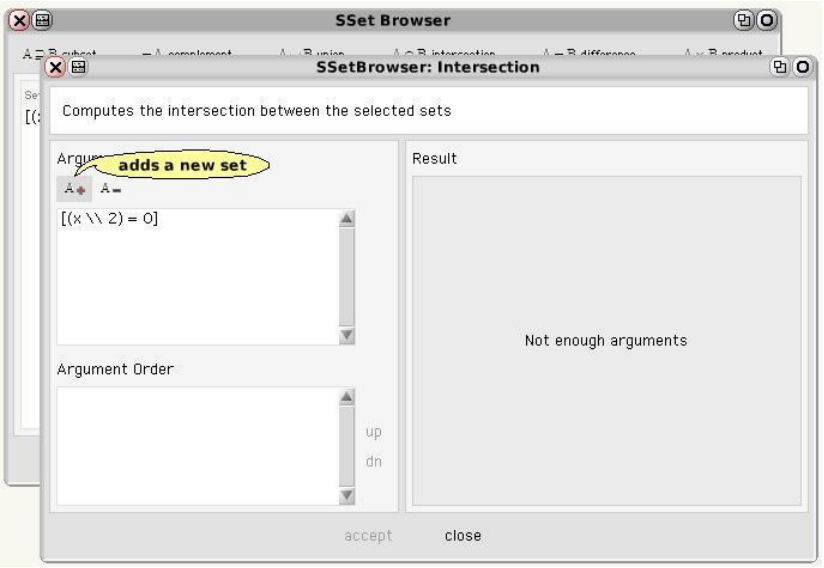
Paso 2: se abre el `SSetPicker`. Creamos el conjunto de números pares, ingresando la expresión `{[x \\ 2 = 0]}` y presionando el botón *Create* para crear este nuevo conjunto.



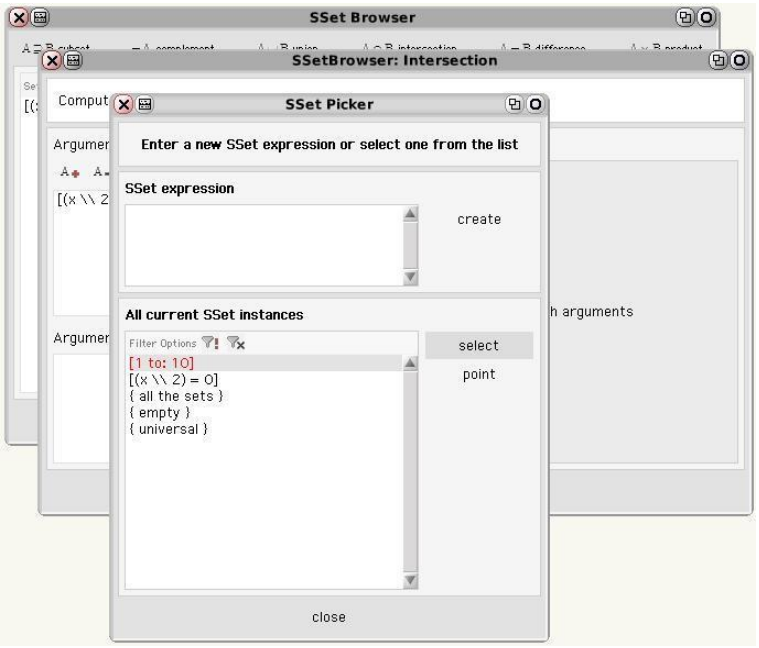
Paso 3: el conjunto de números pares fue agregado al `SSetBrowser`, vemos en el panel de la derecha que es un conjunto no enumerable. Presionamos el botón *Intersection* para realizar la operación.



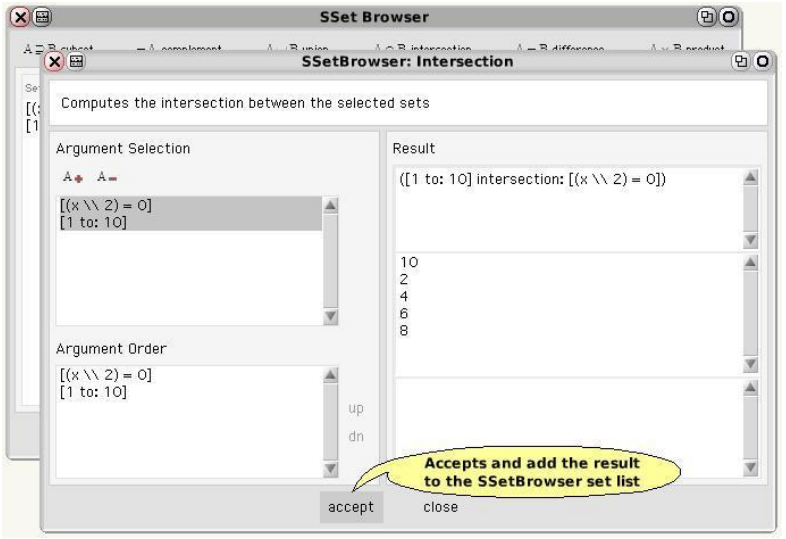
Paso 4: se abre la pantalla especializada en la operación intersección, que nos indica que falta otro conjunto para poder realizar esta operación. Presionamos entonces el botón *A+* para agregar el conjuntos de números del 1 al 10.



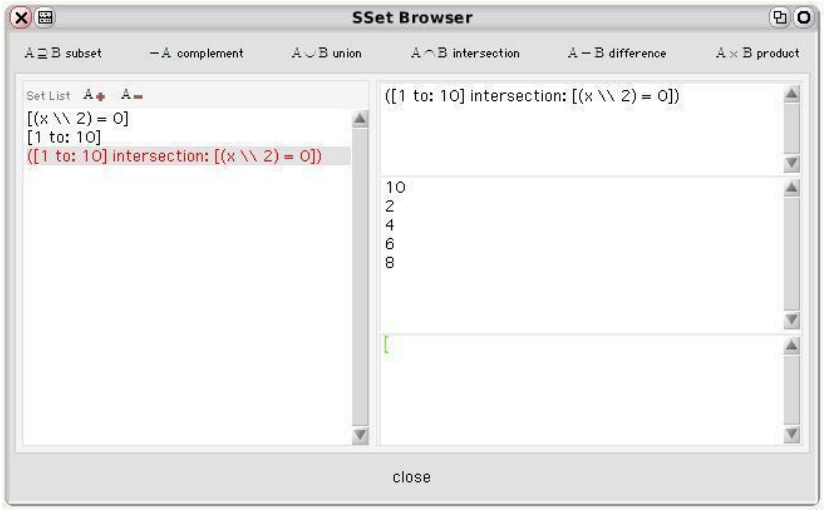
Paso 5: se abre nuevamente el `SSetPicker`. Vemos que el conjunto que buscamos ya está en la lista de conjuntos del ambiente y lo seleccionamos. Utilizando el botón *Select* confirmamos la selección.



Paso 6: el conjunto de números del 1 al 10 fue agregado a la ventana de la operación intersección, que nos muestra en el panel de la derecha el conjunto resultante de la operación. Aceptamos la operación presionando el botón *Aceptar*.



Paso 7: el conjunto resultando de la operación es agregado al `SSetBrowser`, para que podamos seguir operando con él.



Prueba de Concepto

Reimplementación de Senders Of, Implementors Of y Users Of

El ambiente de Smalltalk provee de varias herramientas al programador. Algunas de estas son simplemente para la presentación del resultado de una consulta. Por ejemplo en Squeak existen las consultas ***Senders Of***, ***Implementors Of*** y ***Users Of***, que retornan un conjunto de métodos como resultado.

Senders Of	Retorna los métodos que envían un mensaje dado
Implementors Of	Retorna los métodos que implementan un mensaje dado.
Users Of	Retorna los métodos que posean una referencia a una clase dada.

Tabla: Herramientas Senders Of, Implementors Of y Users Of

Su implementación en Squeak es bastante directa. Se realiza la consulta al sistema, se obtiene una colección con los métodos que responden a la consulta y se construye una ventana para la visualización del resultado.

No queremos dejar de notar aquí que la semántica de cada consulta en ningún momento queda vinculada con el resultado. Una vez que se obtuvo la colección de métodos ya no hay forma de saber cuál criterio los seleccionó por sobre el resto (salvo interpretando el título de la ventana). Nos preguntamos entonces cuál será el impacto de utilizar conjuntos por comprensión aquí y cuáles ventajas traería, por lo que realizamos una prueba de concepto para evaluar la idea.

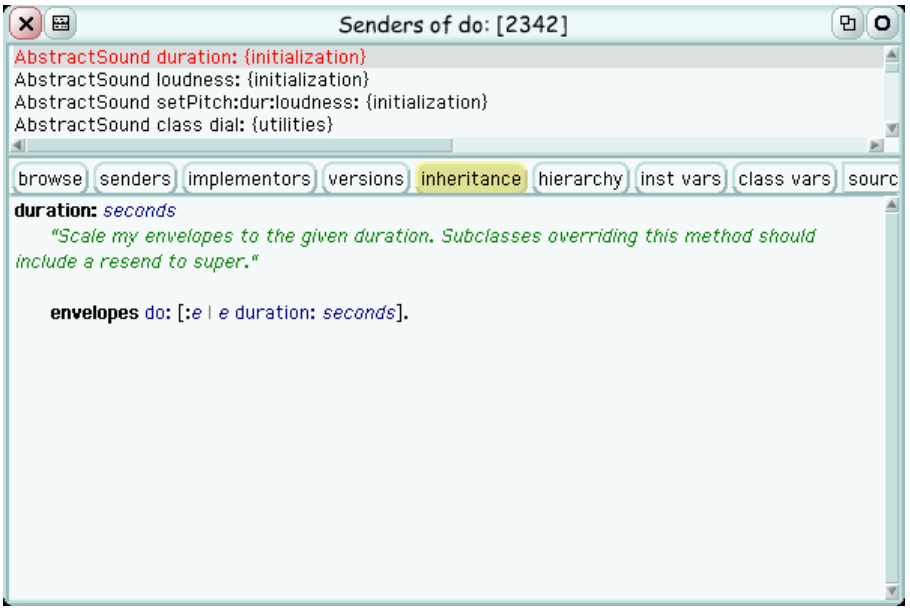


Figura: La herramienta `Senders` de Squeak

Para la prueba de concepto modelamos la semántica de cada consulta con instancias `SSet` definidas por enumeración.

La siguiente expresión retorna el conjunto *Senders Of* del mensaje `#do:`.

```
SSet enumeratorString:
       '{[ MethodReference allSendersOf: #do:]}'.
```

Análogamente la siguiente expresión retorna el conjunto *Implementors Of* del mismo mensaje:

```
SSet enumeratorString:
       '{[MethodReference allImplementorsOf: #do:]}'.
```

Por último la siguiente expresión retorna el conjunto *Users Of* de la clase `Collection`:

```
SSet enumeratorString:
      '{[MethodReference allUsersOf: Collection]}'.
```

La presentación del resultado decidimos realizarla a través de una instancia de `SSetInspector`, dado que con estos podemos no solo visualizar el resultado sino interactuar con los conjuntos y aprovechar toda la funcionalidad que brinda `SSet`. La siguiente figura muestra un inspector sobre los *Senders Of* del mensaje `#do:`.

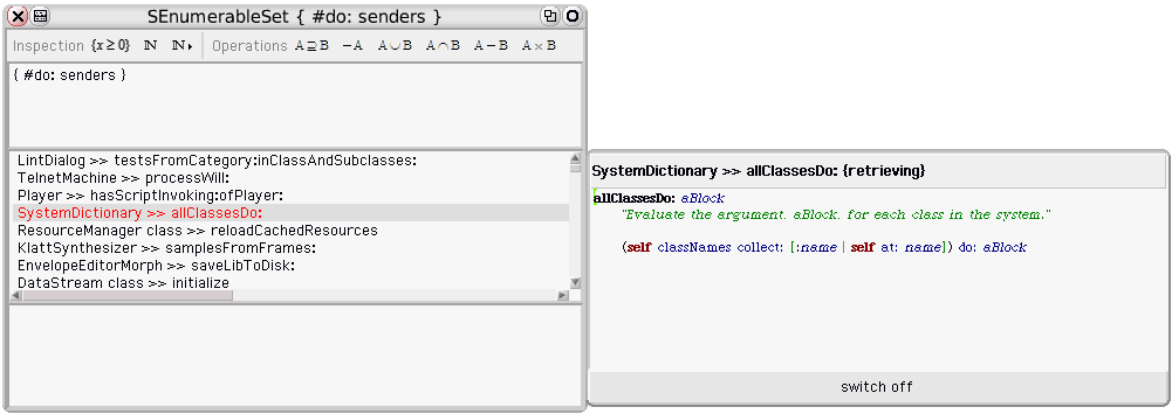


Figura: Senders del mensaje `#do:`, resuelto con SSets

El panel a la derecha es una extensión a `SSetInspector` que implementamos para la prueba de concepto. Esta extensión permite utilizar un visualizador particular sobre el elemento seleccionado del conjunto. La figura muestra el visualizador de instancias de `MethodReference`.

A continuación listamos los aportes más importantes que obtuvimos por utilizar SSets para implementar las tres herramientas:

Semántica explícita. A diferencia del modelo anterior donde se pierde la semántica de la consulta, la solución con conjuntos por comprensión la hace explícita, permitiéndole al usuario acceder a ella como un objeto más.

Consistencia del resultado a lo largo del tiempo. Los conjuntos por comprensión aseguran que la información presentada al usuario siempre será consistente con su semántica, dado que los mismos mantienen su contenido automáticamente actualizado.

Posibilidad de refinar la búsqueda. Una vez realizada la consulta, el usuario puede utilizar las operaciones de conjuntos para seguir con la búsqueda hasta obtener un resultado más útil o acotado. Por ejemplo, resulta trivial con conjuntos por comprensión y `SSetInspector` restringir la búsqueda de *Implementors Of* de `#do:` a los métodos que no provengan de una subclase de `Collection`.

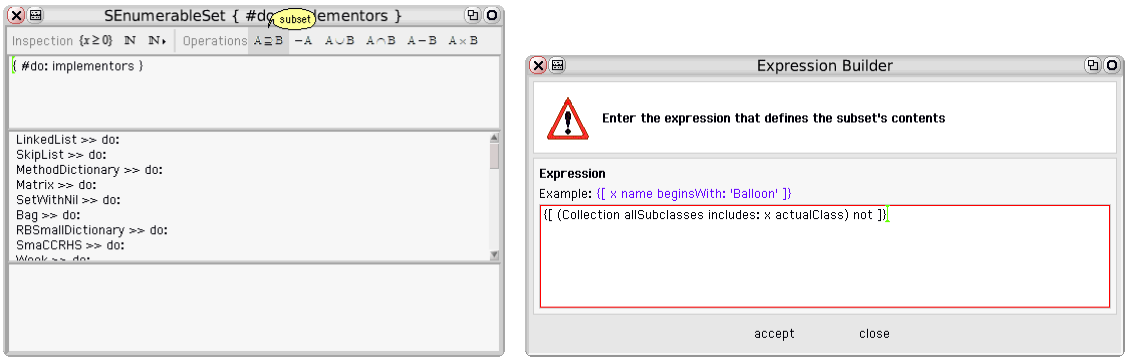


Figura: Operación de subconjunto sobre el conjunto implementors de `#do:`

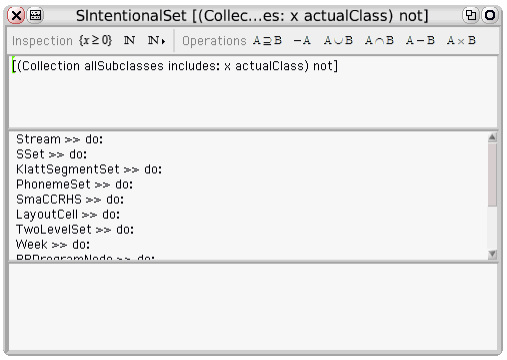


Figura: Conjunto de Implementors de `#do:` que no son descendientes de `Collection`

Posibilidad de realizar consultas más complejas que las predefinidas por el sistema. Por medio de operaciones de conjunto es posible componer consultas sencillas en consultas más complejas. Por ejemplo de esta manera es posible obtener los métodos *Senders Of* del mensaje `#do:` y también del mensaje `#do:without:`.

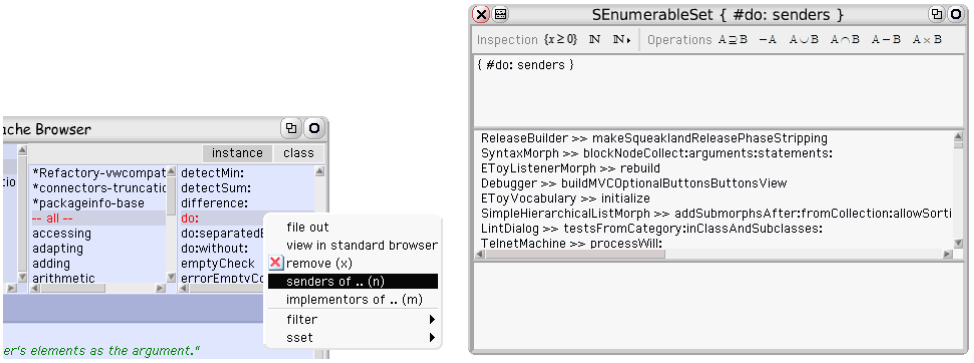


Figura: Senders de `#do:`

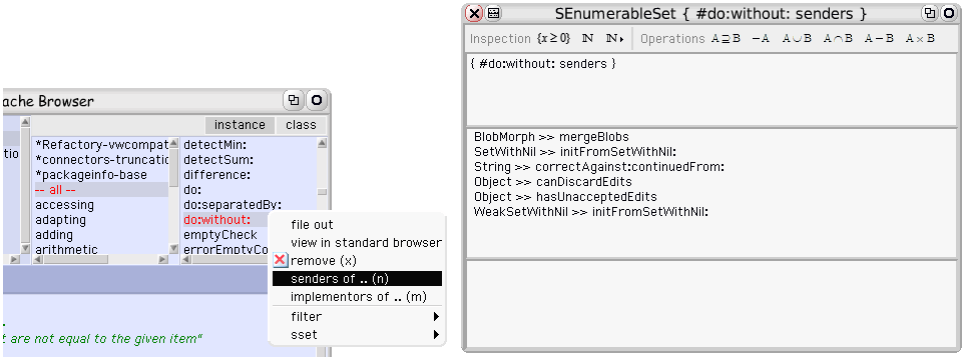


Figura: Senders de `#do:without:`

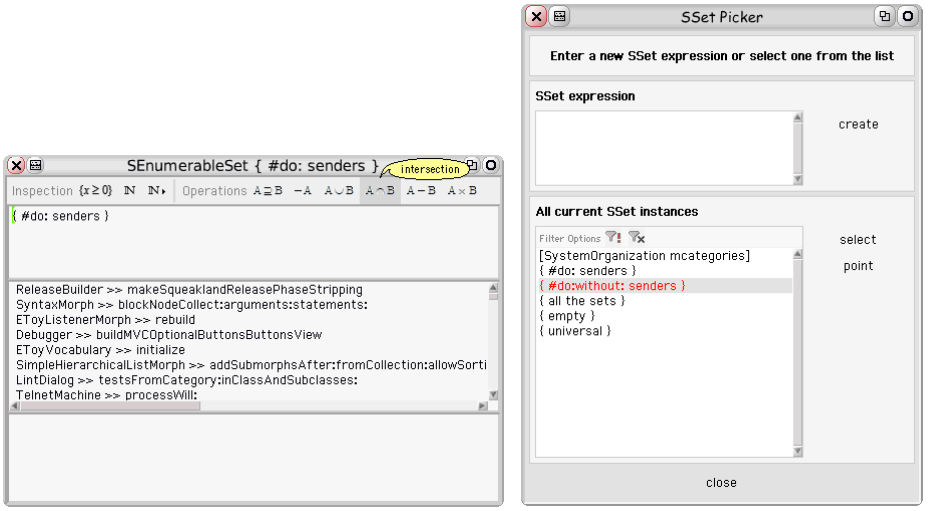


Figura: Operación de Intersección de senders de `#do:` y `#do:without:`

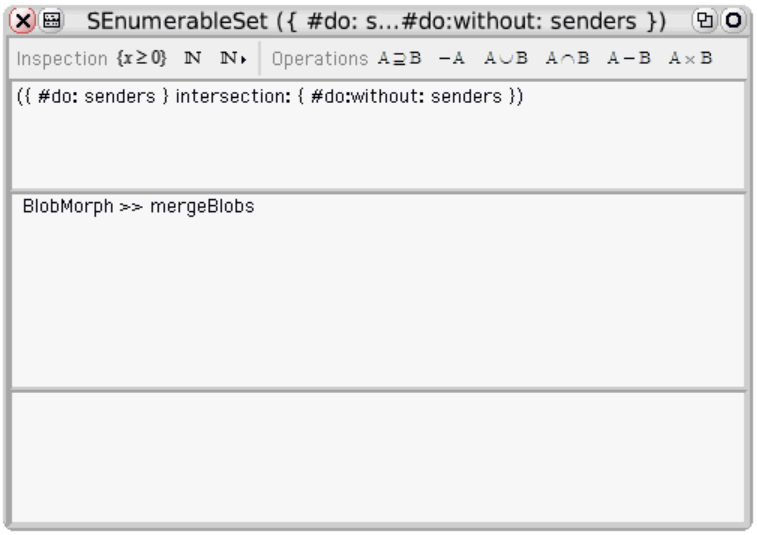


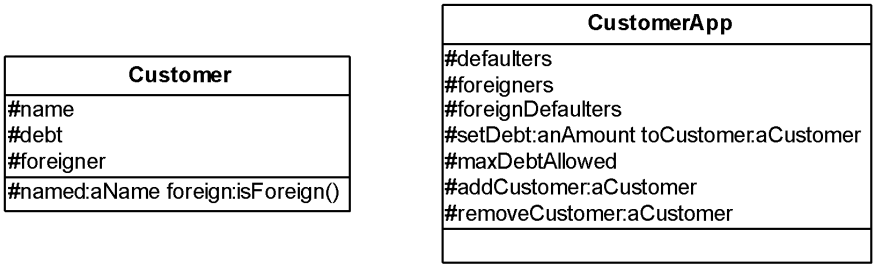
Figura: Resultado de la Intersección de los senders de `#do:` y `#do:without:`

Un ejemplo de simplificación de código

Hay un aspecto que no puede observarse en el ejemplo que usamos para la prueba de concepto y que para nosotros representa una ventaja importante de la utilización de SSets: utilizando el framework se simplifican las aplicaciones. Esto se debe a que la tarea de mantener actualizados los conjuntos de objetos de la aplicación puede ser ahora delegada al framework, liberando a la aplicación de realizar tareas "de mantenimiento" y reduciendo su tamaño y complejidad.

Para ilustrar este punto mostramos a continuación un ejemplo con y sin la utilización de SSets, para que pueda compararse. Se trata del ejemplo que mencionamos en la sección *Actualización de la Pertenencia al Conjunto* ampliado, donde se tiene un conjunto de clientes y un conjunto de clientes morosos. Mostraremos además cómo el mantenimiento que mencionamos se complica cuando se incorporan nuevos conceptos, por ejemplo los de cliente extranjero y cliente extranjero moroso.

Para modelar nuestra aplicación utilizamos las clases Customer y CustomerApp:



Como muestran los diagramas, la aplicación modela los conceptos de clientes morosos (`#defaulters`), extranjeros (`#foreigners`) y extranjeros morosos (`#foreignDefaulters`).

En la versión con colecciones dichos conceptos se encuentran modelados con instancias de `Set` y cada vez que un nuevo cliente se incorpora al sistema, es removido o su monto de deuda cambia, la aplicación debe mantener las colecciones en estado consistente. Esta es la manera habitual en la cual podría implementarse esta aplicación en Smalltalk:

```
initialize
	"Se crean los conjuntos"
	customers := Set new.
	defaulters := Set new.
	foreigners := Set new.
	foreignDefaulters := Set new.

addCustomer: aCustomer
	"Cuando un cliente es agregado al sistema, necesitamos agregarlo
	manualmente a cada uno de los conjuntos si es necesario."
	customers add: aCustomer.
	self checkIfDefaulter: aCustomer.
	self checkIfForeigner: aCustomer.
	self checkIfForeignDefaulter: aCustomer.

removeCustomer: aCustomer
	"Cuando se elimina un cliente del sistema, tenemos que eliminarlo
	manualmente de cada una de las colecciones que lo contienen."
	customers remove: aCustomer.
	defaulters remove: aCustomer ifAbsent: [].
	foreigners remove: aCustomer ifAbsent: [].
	foreignDefaulters remove: aCustomer ifAbsent: [].

setDebt: anAmount toCustomer: aCustomer
	"El sistema es informado de que la deuda de un cliente debe ser
	modificada, entonces cambia el valor de dicha deuda y agrega o elimina
	el cliente de los demás conjuntos, según sea necesario."
	aCustomer debt: anAmount.
	self checkIfDefaulter: aCustomer.
	self checkIfForeignDefaulter: aCustomer.

maxDebtAllowed
	"Deuda máxima admitida antes de considerarlo deudor"
	^500

checkIfDefaulter: aCustomer
	"Agrega/elimina un cliente de de la colección defaulters, según sea
	necesario"
	 ( aCustomer debt > self maxDebtAllowed )
		ifTrue: [defaulters add: aCustomer]
		ifFalse: [defaulters remove: aCustomer ifAbsent:[]].

checkIfForeigner: aCustomer
	"Agrega/elimina un cliente de de la colección foreigners, según sea
	necesario"
	aCustomer foreigner
		ifTrue: [foreigners add: aCustomer]
		ifFalse: [foreigners remove: aCustomer ifAbsent: []]

checkIfForeignDefaulter: aCustomer
	"Agrega/elimina un cliente de de la colección foreignDefaulters, según
	sea necesario. #checkIfDefaulter y #checkIfForeigner deben ser
	evaluados previamente."
	(foreigners includes: aCustomer) &
	(defaulters includes: aCustomer)
		ifTrue: [foreignDefaulters add: aCustomer]
		ifFalse: [foreignDefaulters remove: aCustomer ifAbsent:[]].
```

Los métodos `#checkIfDefaulter:`, `#checkIfForeigner:` y `#checkIfForeignDefaulter:` son los responsables, de mantener consistente cada una de las colecciones. El método `#removeCustomer:` debe hacer lo mismo cuando los clientes son dados de baja del sistema. Vemos

como la responsabilidad de mantener las colecciones se dispersa a lo largo de tres puntos de la aplicación, el alta de clientes, la modificación del estado de los mismos y su baja.

Por el contrario, en la versión con SSets, la aplicación no posee código dedicado a mantener sus instancias de los conjuntos. Los mismos son definidos declarativamente, explicitando su semántica, y mantenidos por el framework.

```
initialize
	"customers es la única colección. Acá pueden verse las definiciones de
	los otros conjuntos"
	customers := Set new.
	defaulters := SSet expressionString:
			'{[ aCustomer debt > app maxDebtAllowed ]}'
			  inContext: { #app -> self }
			  inUniverse: customers asSSet.
	foreigners := '{[customer foreigner]}'
			toSSetInUniverse: customers asSSet.
	foreignDefaulters := defaulters intersection: foreigners.

addCustomer: aCustomer
	"Cuando un cliente es agregado al sistema, solo hace falta agregarlo a
	la colección customers. Los SSets se mantienen actualizados
	automáticamente."
	customers add: aCustomer.

removeCustomer: aCustomer
	"Cuando un cliente es eliminado del sistema, solo hace falta sacarlo de
	la colección."
	customers remove: aCustomer.

setDebt: anAmount toCustomer: aCustomer
	"El sistema es informado de que la deuda de un cliente debe ser
	modificada, entonces cambia el valor de dicha deuda. No hace falta
	código que mantenga actualizados los SSets."
	aCustomer debt: anAmount.

maxDebtAllowed
	"Deuda máxima admitida antes de considerarlo deudor"
	^500
```

Obsérvese cuánto más simple es la versión con SSets de la aplicación.

La versión con colecciones podría haberse implementado de forma tal que estas se recomputen cada vez que se las pida, por ejemplo podrían haberse incluido los siguientes métodos:

```
defaulters
      ^customers select:
            [:aCustomer | aCustomer debt >self maxDebtAllowed]

foreigners
      ^customers select: [:aCustomer | aCustomer foreigner].

foreignDefaulter
      ^self defaulters intersection: self foreigners.
```

Si bien esta solución no parece más compleja que la de SSets, porque también evita la complejidad del mantenimiento de las colecciones, funcionalmente es inferior. Presenta las siguientes desventajas:

1. La semántica de los conjuntos queda únicamente expresada en el código fuente de los métodos y disociada de los conjuntos.

2. Se crea una nueva colección con cada evaluación del método, y por lo tanto dichas colecciones representan un snapshot del conjunto y quedan desactualizadas ni bien hay un cambio en el estado de la aplicación. El siguiente código lo ilustra:

   ```
   app setDebt: 0 to: aCustomer.
   defaulters1 := app defaulters.
   app setDebt: 5000 to: aCustomer.
   defaulters2 := app defaulters.

   defaulters1 = defaulters2   -> false
   ```

3. La performance de la aplicación podría verse afectada si existe un gran número de clientes, ya que se recalcularían las colecciones muchas más veces que las necesarias.

Obviamente el ejemplo puede refinarse y mejorarse, agregando por ejemplo un mecanismo de observación y cache, para no recalcular los conjuntos innecesariamente, mejorando así los puntos (2) y (3) mencionados anteriormente. Pensamos sin embargo, que estas ideas no hacen más que confirmar la utilidad de SSets, porque lo que se estaría haciendo es implementar en la aplicación aspectos que el framework ya trae incorporados, es decir se estaría acercando el modelo utilizado al modelo implementado en SSets.

Comentarios sobre el costo de uso de instancias SSet

En el presente trabajo nos enfocamos en los aspectos conceptuales de la solución, dejando de lado completamente el aspecto de optimización de performance del modelo.

La excepción a esto fue la inclusión del caché en el framework. Pero a pesar de las mejoras de performance que indudablemente el uso del cache trae aparejadas, tampoco nos avocamos a la tarea de optimizarlo. Debido justamente a este caché, pensamos que adicionalmente a la realizacion de un cálculo de órdenes de magnitud de la performance del modelo, lo mejor sería reimplementar una serie de aplicaciones utilizando SSets y comparar su performance contra la del modelo de colecciones tradicional. Esta puede ser una interesante rama para continuar con la investigación, tal como mencionamos en la sección Trabajo a Futuro.

Queremos mencionar también la particularidad de que el costo de las operaciones entre conjuntos es O(1), dado que las mismas se realizan combinando las definiciones de los conjuntos y postergando todo lo posible las operaciones de testeo de los elementos. Las que solo se realizan cuando el conjunto recibe los mensajes #values e #includes:, dos operaciones que sin hacer un análisis de costo formal creemos que se comportan dentro de lo razonable, o sea, sin costo exponencial.

Conclusiones

Construimos SSets, un framework de conjuntos que resuelve los problemas y falencias del modelo actual de conjuntos que planteamos en la sección *Falencias del Modelo Actual de Conjuntos*. A saber:

- Permite representar conjuntos infinitos utilizando el framework y de manera consistente con los conjuntos finitos.
- Incluye en el modelo la semántica de los conjuntos, facilitando la comprensión de un diseño y achicando el gap semántico con la realidad.
- Puede detectar automáticamente cuando cambian los elementos de un conjunto y cuenta con un sistema de notificaciones para avisar de dichos cambios.
- Al no ser contenedores de elementos y estar definidos declarativamente, los conjuntos de SSets eliminan la necesidad de mantener la sincronización de los conjuntos desde las aplicaciones, simplificándolas.
- Facilita la programación exploratoria, por contar con herramientas que permiten manipular los conjuntos y operar con ellos de manera simple y rápida.

Obtuvimos un modelo sumamente simple y compacto para el framework. Esto ocurrió luego de varias iteraciones y muchos cambios y trae como consecuencia que el modelo sea sencillo de aprender y de utilizar.

El framework amplia el abanico de posibilidades del diseñador, su caja de herramientas. Ofrece una alternativa a considerar en casos donde la necesidad de conjuntos se cubría mediante soluciones *ad-hoc* que muchas veces tenían un alto costo asociado (ver *Falencias del modelo actual de conjuntos*) .

Durante el desarrollo, a medida que utilizamos los conjuntos por comprensión, nos surgió la necesidad de poder enumerar sus elementos. Esto dio origen a la clase `SEnumerableSet`, que representa una especie de intermedio entre los conjuntos matemáticos por comprensión y por extensión. Pueden utilizarse como conjuntos definidos por extensión con respecto a que siempre puede enumerarse sus elementos, pero a la vez tienen semántica (como los conjuntos por comprensión) que se mantiene a través del tiempo, y sus elementos pueden ir cambiando.

Desde el punto de vista de simplificación de código y reuso, no podemos dejar de mencionar la actualización automática de conjuntos. Gracias a esta muchas tareas de las que antes se ocupaban normalmente las aplicaciones ahora son realizadas por el framework, permitiendo un diseño más limpio y claro al mismo tiempo. El mismo no deberá ocuparse de aspectos accidentales como la sincronización de colecciones sino, más importante, de la semántica de las mismas.

Trabajo Futuro

Un terreno que no exploramos y que pensamos que puede derivar en algo sumamente interesante es un refactoring de la jerarquía `Collection` (sí, ¡otro intento!) que implique profundizar en la separación conceptual entre *Collection* y *Contenedor de elementos* y permita tener otros tipos de Collection con semántica, como SOrderedCollection, etc. Tal vez sería interesante intentar reemplazar la herencia por composición o por Traits [UNG91, DUC03].

Otro trabajo a futuro podría ser utilizar el framework en alguna aplicación real, intensivamente, y trabajar más en el aspecto de performance del mismo, que fue dejado un poco de lado en favor de los temas más conceptuales. A pesar de que implementamos el mecanismo de *cache*, no lo probamos y optimizamos lo suficiente como para estar seguros de que la performance es buena en un sistema real.

Hoy en día SSets no da ningún tipo de soporte para resolver la igualdad entre conjuntos. Sería interesante investigar el tema y ver qué ayuda puede ofrecerse a los usuarios, al menos para resolver el problema en casos particulares.

Podría ampliarse el espectro de operaciones con conjuntos que ofrece el framework, ya que al día de hoy están implementadas sólo las operaciones básicas (unión, intersección, diferencia, producto y complemento).

SSets permite representar conjuntos infinitos. Podrían investigarse otras aplicaciones que permitan representarlos y agregar a SSets lo necesario para brindar un soporte más completo para dichos conjuntos.

Pensamos también que sería interesante portar el framework a otras plataformas de desarrollo como ser .Net, Java, etc. y ver si tiene aceptación entre las distintas comunidades de desarrolladores, validando así nuestra idea de que estamos ante un aporte práctico y utilizable en el desarrollo de aplicaciones.

Anexo I: Experiencia de Uso

Para obtener una experiencia de uso del framework de conjuntos y probarlos en el contexto de una aplicación de mediana complejidad desarrollamos Mustache, una implementación basada en SSets del browser de clases de Smalltalk.

Las siguientes figuras muestran el browser actual de clases de Squeak y nuestra implementación. Como se puede apreciar, la herramienta se divide mayormente en un área superior de navegación y en un área inferior de edición.

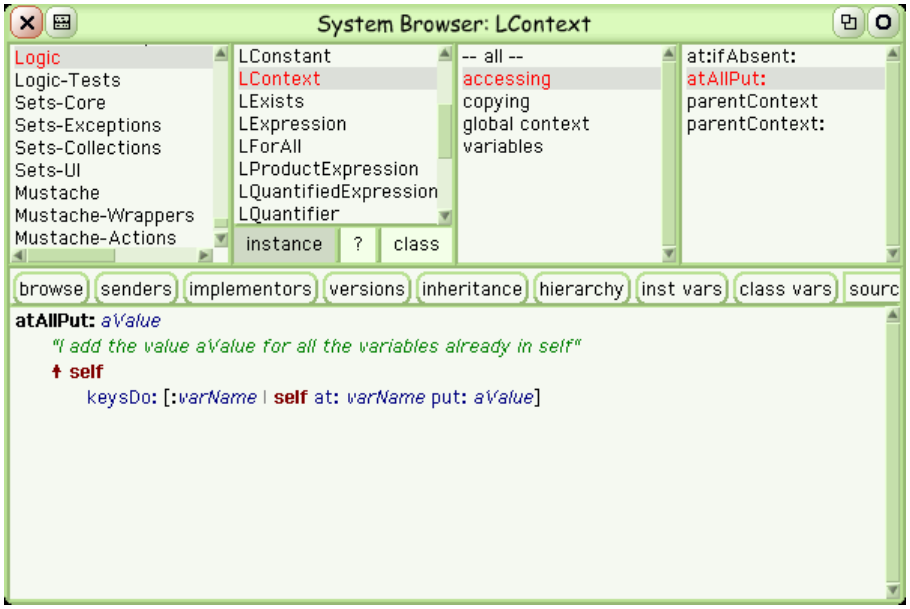


Figura: El `SystemBrowser` de Squeak

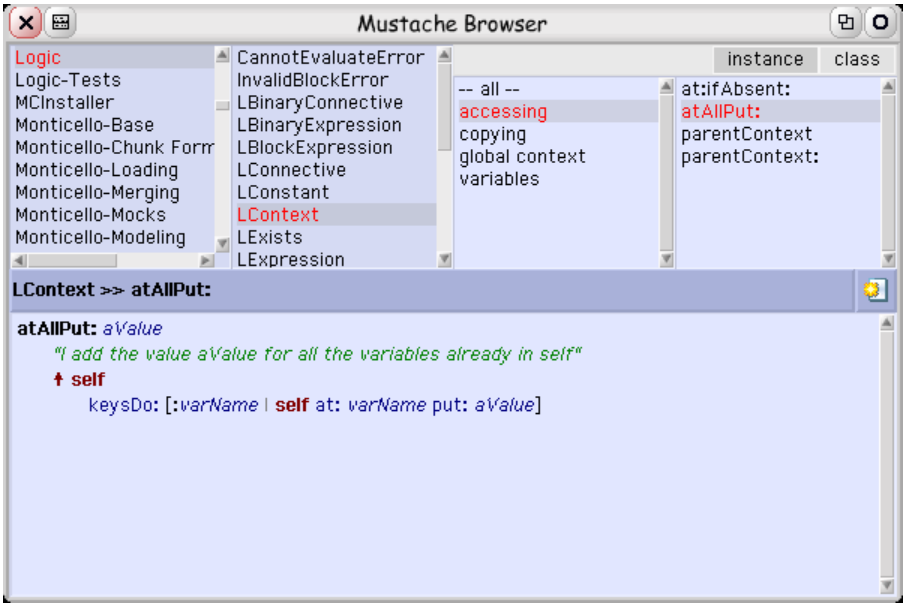


Figura: Mustache Browser

La navegación por las clases del ambiente y de su contenido se realiza en los 4 paneles superiores. El primer panel presenta al usuario las distintas categorías de clases presentes en el ambiente. El segundo panel muestra las clases de la categoría seleccionada. El tercero lista las categorías de los métodos de la clase seleccionada en el panel anterior. Por último, el cuarto panel muestra los métodos de la categoría que fue seleccionada.

La semántica asociada con cada uno de estos paneles no escapa a los usuarios de la herramienta, motivo por el cual la escogimos para la experiencia. Creemos que explicitando la semántica de estos conjuntos obtendremos desde el punto de vista de diseño, una arquitectura más cercana a la visión semántica del usuario. Y desde el punto de vista de interacción, agregaremos posibilidades de uso que enriquecerán significativamente una herramienta que ha sido parte de la familia de Smalltalk sin cambios importantes en los últimos 20 años.

La selección de conjuntos

Los conjuntos que escogimos para representar a los 4 paneles son los siguientes:

- { *Categorías de clases en el ambiente de Smalltalk* }
- { *Clases que forman parte de una categoría de clases* }
- { *Categorías de métodos que forman parte de una clase* }
- { *Métodos que forman parte de una categoría de métodos* }

Si bien esta selección parece lógica podríamos haber escogido en su lugar:

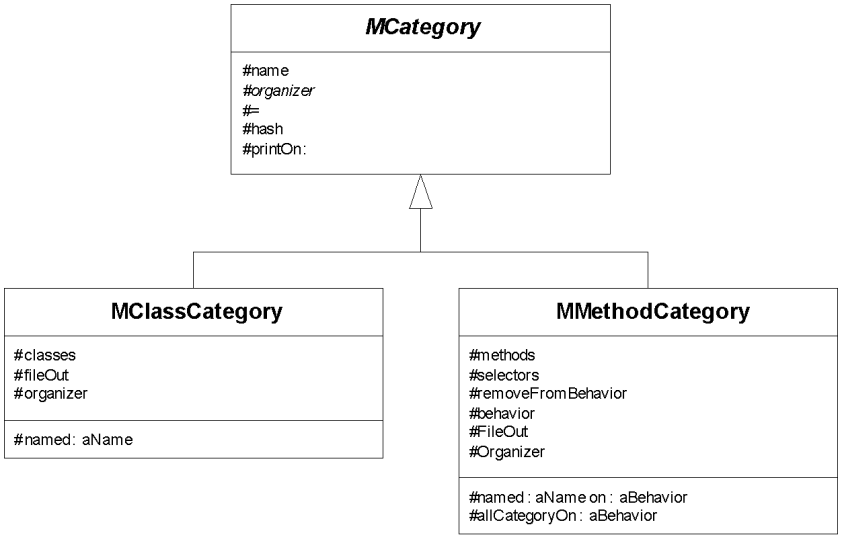
- { *Nombres de las categorías de clases en el ambiente de Smalltalk* }
- { *Nombres de las clases que forman parte de una categoría de clases* }
- { *Nombres de las categorías de métodos que forman parte de una clase* }
- { *Selectores que forman parte de una categoría de métodos* }

La diferencia entre ambos es que mientras los primeros poseen como elementos a instancias de clases, métodos y categorías, los últimos sólo proveen las instancias de `String` que se visualizarán en la interfaz de usuario. Este último enfoque es el actualmente utilizado por Squeak y el cual no compartimos. Utilizando strings perdemos la información de los objetos reales y eso dificulta el uso de los conjuntos para otra cosa más que su visualización.

Reificación de los conceptos Categorías de Clases y Categorías de Métodos

Para la categorización de sus clases y métodos, Squeak utiliza la jerarquía de clases `Categorizer`. Esta y sus subclases, proveen a Squeak del mapeo entre los métodos y las clases a sus correspondientes categorías. Sin embargo, sólo se utilizan instancias de `String` para representar las categorías. Por este motivo Mustache implementa la jerarquía de clases `MCategory`, cuyas instancias le permiten Mustache interactuar con las categorías como objetos en lugar de interactuar con sus nombres.

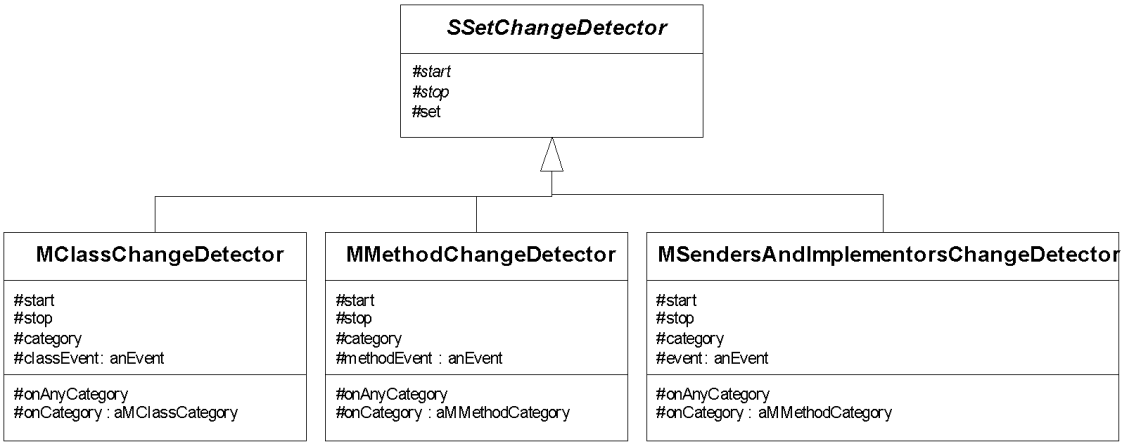
- **La jerarquía de clases de MCategory**



Actualización Automática

El mecanismo de detección de cambios provisto *out of the box* en `SSets` no es eficiente para la detección de cambios de clases y métodos dado que el mismo se basa en *polling*. Por este motivo desarrollamos *change detectors* especializados en este tipo de contenidos y que aprovechan el sistema de notificación de eventos de sistema de Squeak.

- **Clases ChangeDetector utilizadas en MMustache.**



Instanciación

El siguiente código Smalltalk es el responsable de la instanciación de los conjuntos con las categorías de clases del sistema.

```
SSet enumeratorString:
       '{[ SystemOrganization categories
             collect:
               [:categoryName |
                   MClassCategory named: categoryName
               ]
       ]}'.
```

El siguiente código Smalltalk instancia al conjunto de clases de una MClassCategory determinada.

```
SSet enumeratorString:
        '{[ (aCategory organizer listAtCategoryNamed: aCategory name)
               collect:
                 [:aClassName|
                     Smalltalk at: aClassName
                 ]
         ]}'.
```

El siguiente código Smalltalk instancia al conjunto de categorías de métodos de una clase determinada.

```
SSet enumeratorString:
       '{[ ( aClass organization categories
               collect:
                 [:categoryName |
                     MMethodCategory named: categoryName on: aClass
                 ]
           )
        ]}'.
```

El siguiente código Smalltalk instancia al conjunto de métodos de una MMethodCategory determinada.

```
SSet enumeratorString:
       '{[( aCategory selectors collect:
                 [:selector |
                     MethodReference mustacheClass: aCategory class
                                     selector: selector
                  ]
           )
        ]}'.
```

Vistas

Para la visualización de los conjuntos en la interfaz de usuario utilizamos al widget `SSetPluggableList`. Este resulto ser de gran utilidad dado que de forma instantánea y sin código adicional, obtuvimos un widget de lista que automáticamente se refresca ante las actualizaciones de los conjuntos (por la creación de una nueva clase, la recategorización de un método, etc.).

Sin el mecanismo de actualización automática de SSets y sin la sincronización de `SSetPluggableList` con lo anterior, hubiéramos necesitado hacer (por cada conjunto):

1. Registrar Mustache como un *observer* del sistema para que este reciba los eventos de cambios en el ambiente (creación, edición o eliminación de clases, creación, edición o eliminación de métodos, etcétera).

2. Procesar los eventos del sistema y recomputar los conjuntos para que estos tomen en cuenta los cambios en el ambiente que acaban de ocurrir.

3. De haber cambios visibles, notificar al widget de lista para que este refresque su vista.

Los pasos 1 y 2 son realizados hoy por los *change detectors*. Mustache provee su propia implementación de estos pero solo por motivos de eficiencia. Además queremos comentar que la implementación de *change detectors* permite su posterior reutilización a diferencia de haberlo hecho directamente en la clase de Mustache.

El paso 3 es realizado de forma automática por la clase `SSetPluggableList`. Esta se registra como un *observer* del conjunto y automáticamente refresca su contenido cuando el conjunto notifica un cambio.

Otra ventaja muy importante que hemos ganado en utilizar `SSetPluggableList` y SSets es la funcionalidad de filtros.

Sin repetir la descripción de los filtros hecha en la sección *Herramientas Desarrolladas*, queremos resaltar lo útil que los encontramos en el marco de Mustache brindando nuevas formas de interacción con la herramienta.

Figura: La parte superior de un Mustache browser mostrando un filtro en el segundo panel.

Anexo II: Notación de los Diagramas Utilizados

Para acompañar nuestros diseños hemos adoptado al lenguaje UML con algunas modificaciones que lo hacen más adecuado para trabajar con código Smalltalk.

Notación Sintáctica General

Las colaboraciones se describen siempre como "`O m`", donde `O` representa un objeto y `m` un mensaje. Salvo excepciones, cada nombre, tanto de objetos como de mensajes respeta el siguiente formato:

- Los nombres comienzan con minúscula, salvo cuando el nombre tiene alcance global (el nombre de una clase por ejemplo), en cuyo caso la primera letra será en mayúsculas.
- Cuando el nombre es formado por más de una palabra, cada nueva palabra comenzará con mayúsculas y no se usan ni espacios ni guiones para separarlas.

El texto que conforma el nombre de los mensajes `m`, sin los parámetros, se llama *selector*. Existen tres tipos distintos de selectores:

- Unary: no reciben parámetros. Ejemplo: `size`, `refresh`, `open`.
- Binary: reciben un único parámetro. Estos mensajes son símbolos y se usan de forma infija. Ejemplo: `+`, `-`, `<`.
- Keyword: están formados por una lista de pares `keyword , parámetro`, donde `keyword` incluye al carácter de dos puntos al final. Ejemplo: `between: anIntenger and: anotherInteger`.

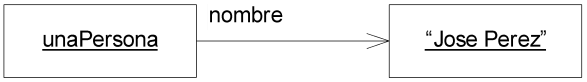
Ejemplo:

```
OrderedCollection new.
clients add: aClient.
totalCount + 1.
```

Diagrama de Instancias (o de objetos)

Estos diagramas representan una vista estructural del modelo. En él se incluyen los objetos participantes durante una situación en particular y las relaciones entre ellos. No es indispensable representar todas las relaciones existentes entre los objetos, sólo las relevantes para el caso de uso que esta siendo descrito.

Ejemplo:

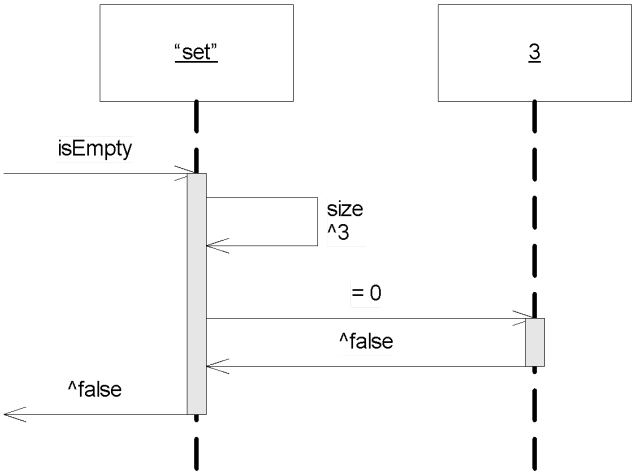


- Cada cajita representa un objeto, una instancia particular.
- Las flechas indican la relación entre los objetos. Mas específicamente, indican que los objetos se conocen y pueden colaborar entre sí (el objeto en el origen de la flecha puede enviar mensajes al objeto apuntado).
- Las etiquetas sobre las flechas indican el nombre con que el objeto origen conoce al objeto destino.

Diagrama de secuencia

Representa una vista dinámica del modelo. En él se incluyen los objetos participantes de la situación que se modela y se muestra cómo colaboran, el intercambio de mensajes entre ellos. Al igual que en el diagrama de objetos, no es indispensable representar todas los mensajes entre los objetos. Sólo lo necesario para transmitir lo que se desea.

Ejemplo:

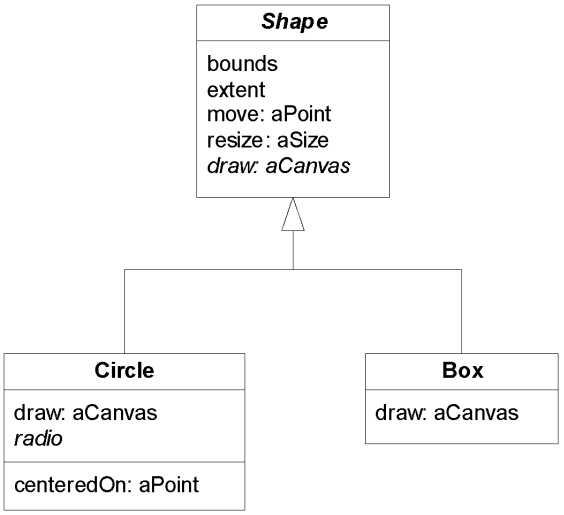


- En un diagrama de secuencia el tiempo fluye de arriba hacia abajo.
- Cada cajita, al igual que en el diagrama de objetos, representa a un objeto.
- Cada flecha representa el envío de un mensaje de un objeto al otro, y si el mensaje retorna un resultado, se pone una flecha en sentido contrario y se antecede el objeto resultante con el carácter de ^.
- Una flecha con el mismo origen y destino representa un mensaje enviado por un objeto a sí mismo.

Diagrama de clases

Representa una vista estática del modelo. En él se incluyen las clases del modelo y sus vínculos de herencia.

Ejemplo:



- Cada cajita representa una clase.
- El nombre de la clase ocupa el primer casillero de la cajita.
- El segundo casillero de la cajita, de estar presente, es utilizado para mostrar el protocolo de instancia de la clase
- El tercer casillero, de estar presente, es utilizado para mostrar el protocolo de clase.
- Cuando el nombre de la clase se encuentra en itálica indica que la misma es abstracta.
- La flecha indica relación de herencia y va desde la subclase hacia la superclase.
- Cuando un mismo mensaje aparece al mismo tiempo en una subclase y superclase se indica que el mismo fue redefinido por la subclase.

Bibliografía

[ARN96] K. Arnold, J. Gosling and D. Holmes (1996). The Java Programming Language.: Ed. Addison-Wesley

[BE93] Andreas Birrer and Thomas Eggenschwiler (1993). Frameworks in the financial engineering domain: An experience report. European Conference on Object-Oriented Programming, Kaiserslautern, Germany: Ed. Springler-Verlag. p 21-35.

[BEE] David Beech. Intentional Concepts in an Object Database Model. p 131-145.

[BRA96] G. Bracha (1996). The Strongtalk Type System for Smalltalk. In Proc. of OOPSLA96, Workshop on Extending the Smalltalk Language. http://bracha.org/nwst.html

[COOK] William R. Cook Interfaces and Specifications for the Smalltalk-80 Collections Classes

[DEU89] L. Peter Deutsch (1989). Design reuse and frameworks in the Smalltalk-80 system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume II: Applications and Experience*. Ed. Addison-Wesley, Reading, MA. p 57-71.

[DOTNET] http://www.microsoft.com/net/

[EIFFEL] http://docs.eiffel.com

[DUC03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew Black. Traits: Composable Units of Behavior. Proc. of European Conference on Object-Oriented Programming (ECOOP'03), LNCS, vol. 2743, Springer Verlag, July 2003, pp. 248-274.

[GAMMA] Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides (1995). Design Patterns. Elements of Reusable Object-Oriented Software: Ed. Addison-Wesley

[GARAU] Fransisco Garau (2001). Inferencia de tipos concretos en Squeak. Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Buenos Aires, Argentina.

[JAVA] Sun Microsystems, Inc. (1994). Java. http://www.sun.com/java/

[JAVAAPI] Java 2 Platform Standard Edition 5.0 - API Specification. http://java.sun.com/j2se/1.5.0/docs/api/index.html

[JF98] Ralph E. Johnson and Brian Foote (1988). Designing reusable classes. Journal of Object-Oriented Programming, 1(2). p 22-35

[JML92] Ralph E. Johnson, Carl McConnell and J. Michael Lake (1992). The RTL system: A framework for code optimization. In Robert Giegerich and Susan L. Graham, editors, *Code Generation - Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation,* Dagstuhl, Germany: Ed. Springer-Verlag. p 255-274.

[LIU96] Chamond Líu (1996). Smalltalk, Objects and Design.: Ed. toExcel

[LUN89] Charlotte Pii Lunau (1989). Separation of Hierarchies in Duo-Talk. Journal of Object-Oriented Programming, July/August, p 20-25

[PYTHON] Guido van Rossum (2005). Python Library Reference. http://docs.python.org/lib/lib.html

[RJ97] Don Roberts and Ralph Johnson (1997). Evolving Frameworks. A Pattern Language for Developing Object-Oriented Frameworks. University of Illinois.

[ROZ01] Dan Rozenfarb (2001). Framework Construction Laboratory. Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Buenos Aires, Argentina.

[RUBY] http://www.ruby-lang.org

[SAN88] D. W. Sandberg (1988). Smalltalk and exploratory programming: ACM Sigplan notices

[SETUTILS] Class SetUtils http://jakarta.apache.org/commons/collections/api-2.1.1/org/apache/commons/collections/SetUtils.html

[SHE83] B. Sheil (1983). Environments for Exploratory Programming: Datamation

[SMA80] Adele Goldberg and David Robson (1983). Smalltalk-80: The Language and Its Implementation.: Ed. Addison-Wesley

[SMAANSI] NCITS J20 DRAFT of ANSI Smalltalk Standard, December, 1997, revision 1.9 http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf

[SQUEAK] http://www.squeak.org

[STR97] B. Stroustup (1997). The C++ Programming Language.: Ed. Addison-Wesley

[SUNIT] http://sunit.sourceforge.net

[SUNIT2] http://minnow.cc.gatech.edu/squeak/1547

[SYN96] Alan Snyder (1986). Encapsulation and Inheritance in Object-Oriented Programming Languages. In Proc. of OOPSLA; ACM Press.

[UNG87] David Ungar and Randall B. Smith (1987). SELF: The Power of Simplicity.: Sigplan Notices. http://research.sun.com/techrep/1994/abstract-30.html

[UNG91] David Ungar, Craig Chambers, Bay-Wei Chang and Urs Hozle (1991). Organizing Programs Without Classes. Journal LISP and Symbolic Computation, vol 4, issue 3, p37-56.

[WIKI] http://www.wikipedia.org

[WIL06] Hernán Wilkinson. Aconcagua. http://map.squeak.org/package/b546b6af-984e-4e89-a7a5-875ce0435710