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

Tesis de licenciatura

Trait refactorings

Mejorando la utilidad de las herramientas de refactorings

Alumno:
Alejandro González
LU 626/98

Director:
Lic. Hernán Wilkinson

Buenos Aires, 14 de agosto de 2008

Agradecimientos

Quiero agradecer a Hernán Wilkinson por proponerme el tema, haberme dirigido y por entusiasmarme para presentar parte del trabajo en el marco del congreso Smalltalks 2007. Agradezco a Gabriela Arévalo por haber participado en las revisiones. A Carlos Ferro, quien guió mis primeros pasos en este trabajo. Y también a Máximo Prieto, toda la cátedra de Objetos, Alan Kay, Dan Ingalls y la comunidad Smalltalker (en especial la de Squeak) por haberme devuelto las ganas de seguir programando.

Agradezco a Cintia, mi vida, que con su amor, consejos y paciencia me ayudó en el tramo más difícil de la carrera, el último.

A mis padres, Vilma y Alberto, y a mi hermano Daniel, por haberme acompañado en todo este largo trayecto.

Finalmente, agradezco a todos mis familiares y amigos a quienes también les robe mucho tiempo de mi presencia para que ésto pudiera concretarse.

Resumen

Traits es un nuevo concepto para estructurar programas orientados a objetos. Este concepto, permite que el comportamiento total de una una clase pueda ser construido composicionalmente a partir de pequeños rasgos de comportamientos llamados traits.

Traits complementa a la herencia simple y es por esto, que su inclusión conlleva la necesidad de querer refactorizar nuestros diseños, ya que con ellos ahora se pueden resolver cuestiones antes no resueltas, o mal resueltas, por la herencia. En particular, aquellas jerarquías cuestionadas por hacer un fuerte uso de la herencia como único mecanismo para compartir comportamiento, son las que se convierten en candidatas a ser refactorizadas con traits [BSD02]. Pero estas refactorizaciones no pueden ser realizadas eficientemente sin la asistencia de una herramienta apropiada.

Muchos ambientes de desarrollo incluyen herramientas para refactorizar código. Éstas soportan, a menudo, refactorings típicos que pueden realizarse en forma semi-automática, como por ejemplo *Push up method* y *Rename method*. El problema es que, éstas operan sobre conceptos conocidos del paradigma: jerarquía de clases, métodos, variables de instancia, etc. Actualmente, ninguna brinda soporte para la refactorización con este nuevo concepto, traits.

En este trabajo se investigan, tanto desde un punto de vista conceptual como implementativo, nuevos refactorings para traits. El resultado es un nuevo conjunto de operaciones de refactorings que, inicialmente surgieron de ideas básicas y operaciones típicas al programar con traits, pero que luego fueron refinándose a medida que las comenzamos a utilizar para realizar diferentes refactorizaciones.

Estos nuevos refactorings se implementaron en *Squeak*, un dialecto open-source de *Smalltalk*, dado que éste es el único ambiente que incluye una implementación sólida de ambos, el nuevo concepto de traits y una herramienta de refactorings conocida como *refactoring browser*.

Abstract

Traits is a new concept for structuring object-oriented programs. This concept allows us to build the whole behavior of a class by composing it from small building blocks of behaviors, called traits.

Traits complement single inheritance, thus their inclusion creates the need for a re-design of our applications, since with them we can now solve design issues that were not solved, or improperly solved, by using single inheritance. Particularly, those hierarchies criticized for a strong use of the inheritance as the only mechanism to share behavior, become candidates to be refactored towards traits [BSD02]. However, these refactorings cannot be efficiently addressed without the assistance of appropriated tools.

Most development environments include tools to refactor code. These support typical refactorings that can be applied semi-automatically, such as *Push up method* and *Rename method*. The problem is, these tools work on well known concepts of the paradigm: class hierarchies, methods, instance variables, etc. Currently, none of them give support for a refactorization towards this new concept, traits.

In this work we investigate, from both a conceptual and an implementative point of view, new trait refactorings. The outcome is a new set of refactoring operations that initially came up from basic ideas and typical operations when working with traits, but then were polished as we use them to perform some refactorings.

These refactorings were implemented in *Squeak*, an open-source *Smalltalk* dialect, since it's the only environment which includes a solid implementation of both, the traits concept and a refactoring tool called *refactoring browser*.

Índice

Índice de figuras

1. Introducción

Traits es un nuevo concepto para estructurar programas orientados a objetos. Este concepto, permite que el comportamiento total de una una clase pueda ser construido composicionalmente a partir de pequeños rasgos de comportamientos llamados traits.

Traits nos permite resolver cuestiones relacionadas al diseño, dado que complementa la herencia simple. Es por esto, que la inclusión de traits usualmente conlleva la necesidad de querer refactorizar nuestros diseños, ya que con traits se pueden resolver cuestiones antes no resueltas, o mal resueltas, por la herencia. En particular, aquellas jerarquías cuestionadas por hacer un fuerte uso de la herencia como mecanismo para compartir comportamiento entre clases, son las que se convierten en candidatas a ser refactorizadas con traits [BSD02].

Dada la importancia del refactoring durante el desarrollo y mantenimiento de un software, muchos ambientes de desarrollo incluyen herramientas para refactorizar código. Éstas soportan, a menudo, refactorings típicos que pueden realizarse en forma semi-automática, como por ejemplo *Push up method* y *Rename method*. Estos refactorings operan sobre conceptos bien conocidos dentro del paradigma de objetos: jerarquía de clases, métodos, variables de instancia, etc. Pero ninguna de estas brinda, actualmente, soporte para la refactorización con este nuevo concepto, traits.

El objetivo de este trabajo es obtener un nuevo conjunto de refactorings para traits, implementarlos e integrarlos al motor de refactorings de Squeak [SqueakWeb].

Con esto lo que se busca es, primero facilitar el uso y adopción de traits, ya que con la incorporación de este nuevo concepto en un ambiente de desarrollo, también es necesaria la incorporación de herramientas que le den soporte. Trabajar con jerarquías de clases tampoco seria sencillo si no tuviésemos los entornos de desarrollo apropiados que nos permitieran navegar a través de ellas, encontrar clases fácilmente, refactorizarlas, etc.

Segundo, se busca automatizar las operaciones más frecuentes para que no tengan que realizarse manualmente cada vez que el programador desee refactorizar con traits. Si como programadores tuviésemos la posibilidad de poder mejorar varias clases extrayendo algo de comportamiento en un trait, que luego podría ser reutilizado en otras clases, deberíamos hacerlo manualmente dada la falta de refactorings para traits. Es decir, tendríamos que crear el trait apropiado, copiar métodos de la clase al trait, borrar los métodos de las clases y por último hacer que estas clases usen el trait recién extraído. Toda esta refactorización manual se convierte en una tarea tediosa y muy propensa a errores, sobre todo si encontramos que tenemos que realizarla reiteradas veces.

1.1. Agenda

Las siguientes dos secciones de este capítulo detallan los dos conceptos fundamentales en los que este trabajo se desarrolla, refactorings y traits. La tercer sección describe la metodología utilizada a lo largo de todo el trabajo.

Luego, el capítulo 2 detalla los trait refactorings encontrados.

El capítulo 3 muestra los cambios hechos en el motor de refactorings, necesarios para poder implementar los trait refactorings.

Los capítulos 4, 5 y 6 describen casos prácticos. Es decir, describen las diferentes refactorizaciones realizadas que nos llevaron a obtener el conjunto de traits refactorings presentados previamente en el capítulo 2.

Por último, los capítulos 7 y 8 cierran este trabajo con conclusiones y posibles investigaciones futuras.

1.2. Refactoring

Refactorings son operaciones de reestructuración que favorecen el diseño, evolución y reuso de aplicaciones orientadas a objetos. Además, estas operaciones preservan el comportamiento de la aplicación [WFO92].

En el modelo *waterfall* [WFWeb], el diseño es el primer paso y codificar es el siguiente. Con una evolución necesaria de la aplicación, el código es modificado (ya sea para la resolución de defectos o para la inclusión de nuevas funcionalidades) y los diseños que en principio eran buenos, comienzan a decaer gradualmente [SSO02].

Refactorizar es tomar el código y diseño existente y modificarlos hasta obtener un mejor código y un mejor diseño.

Refactoring es una práctica necesaria que complementa todas nuestras otras prácticas, y que nos ayudará a mantener la calidad del software desde un punto de vista de diseño, claridad del código, etc.

Hoy día los distintos tipos de refactorings existentes se pueden categorizar de la siguiente manera:

- Class refactorings: refactorings aplicados a clases o jerarquía de clases.
- Method refactorings: refactorings aplicados a métodos y al envío de mensajes.
- Variable refactorings: refactorings aplicados a variables, ya sean de instancias o de clases.

1.3. Traits

1.3.1. Concepto

Trait es un rasgo o una característica.

Por ejemplo, los números son elementos que pueden compararse, entonces un rasgo de los números, es el hecho de que son comparables. Este rasgo no es exclusivo de los números. Las fechas también la comparten al ser comparables entre si. Lo mismo pasa con las distancias, los volúmenes y otras magnitudes.

1.3.2. En el paradigma de objetos

En el paradigma de objetos, traits son grupos de métodos que sirven como entidades reusables para la construcción de clases [SDNB02]. Permiten que un grupo de métodos puedan, como conjunto, ser nombrados y reutilizados en cualquier parte del sistema, sin importar las jerarquías de clases en donde se los use [DB04].

Con este nuevo concepto, las clases no sólo pueden heredar y definir comportamiento sino que también pueden incorporar nuevos rasgos. Es decir, ahora las clases también pueden incorporar los métodos provistos por la composición de uno o más traits.

De esta forma los traits permiten un nuevo estilo de programación, en la cual éstos son la forma primaria de reuso de código, en vez de ser las clases y la herencia.

Traits no es una variante a la herencia, sino una nueva herramienta, tanto desde el punto de vista conceptual como del implementativo, que complementa a ésta.

A diferencia de otros mecanismos de reuso, como mixins [MixWeb] y la herencia múltiple [MIWeb], los traits no usan la herencia como operador de composición. La composición de traits se basa en un conjunto de operadores que son complementarios a la herencia simple.

1.3.3. Propiedades

Los traits tienen las siguientes propiedades:

- Proveen un conjunto de métodos que implementan comportamiento.
- Requieren un conjunto de métodos que parametrizan el comportamiento provisto.
- No poseen colaboradores internos. Esto implica que no tienen estado y por lo tanto sus métodos no pueden acceder directamente, por ejemplo, a variables de instancia.
- El orden de composición de traits es irrelevante.
- La semántica de una clase no es afectada por el uso de traits. Esta propiedad, conocida como *flatten property*, significa que la semántica de una clase es la misma que si todos lo métodos provistos por traits estuviesen definidos directamente en esa clase.
- No sólo las clases pueden estar compuestas por traits. Los traits también pueden estar compuestos por otros traits.

1.3.4. Ejemplo

El siguiente diagrama muestra dos traits: `TCircle` y `TDrawing`. Cada flecha representa un requerimiento del trait y cada línea terminada en círculo es un método provisto por éste [1].

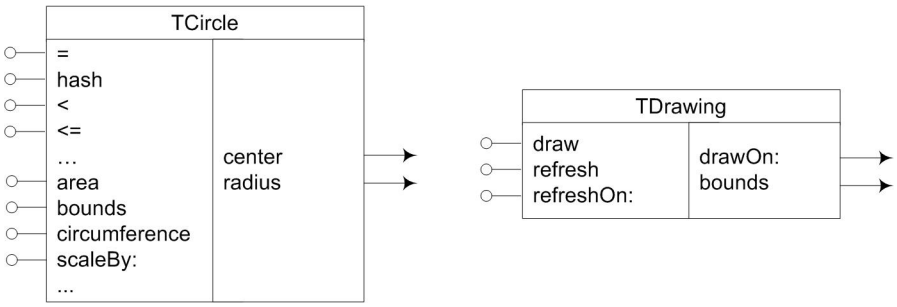

Figura 1.1: Traits `TCircle` y `TDrawing`

El trait `TCircle` provee las propiedades y el comportamiento básico de un círculo, mientras que el trait `TDrawing` provee las propiedades y el comportamiento básico de un objeto dibujable en pantalla.

[1] El apéndice A detalla la notación gráfica utilizada.

Veamos entonces la definición de estos dos traits.

```
Trait named: #TCircle
  uses: {}

= anObject
  ^ self radius = anObject radius and:
   [self center = anObject center]


area
  ^ self radius * Float pi squared

bounds
  ^ Rectangle
    origin: self center - self radius
    corner: self center + self radius

center
  self requirement


circumference
  ^ 2 * self radius * Float pi

radius
  self requirement
```

```
Trait named: #TDrawing
  uses: {}

refreshOn: aCanvas
  aCanvas form
    deferUpdatesIn: self bounds
    while: [ self drawOn: aCanvas ]

refresh
  ^ self refreshOn: World canvas

bounds
  self requirement


drawOn: aCanvas
  self requirement


draw
  ^ self drawOn: aCanvas
```

Nótese, que los requerimientos de un trait, no son mas que métodos regulares en los que ocurre el envío del mensaje `#requirement`.

El siguiente diagrama muestra como se construye, con traits, una clase con estos dos rasgos ortogonales entre sí, ser un círculo y ser dibujable en pantalla: `DrawableCircle`. Las flechas (requerimientos del trait) conectadas a las líneas con círculos (métodos provistos por la clase o trait) denotan cómo la clase satisface los requerimientos de los traits.

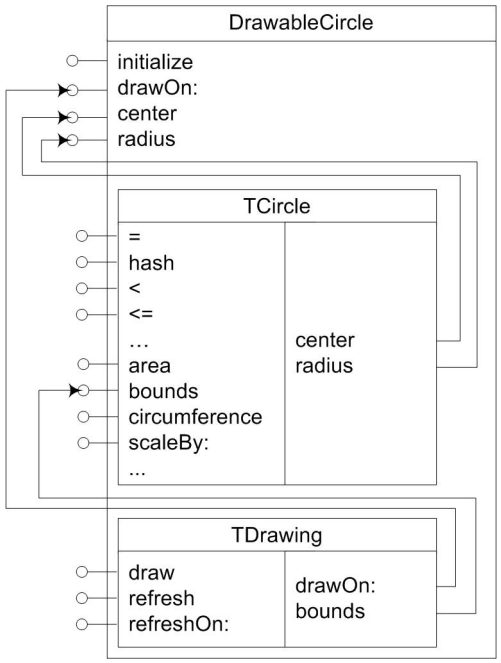

Figura 1.2: Clase `DrawableCircle` compuesta por los traits `TCircle` y `TDrawing`

```
Object
  subclass: #DrawableCircle
  instanceVariableNames: 'center radius'
  uses: { TCirle + TDrawing }

initialize
  center := 0@0.
  radius := 50
```

`DrawableCircle` satisface los requerimientos del trait `TCircle` proveyéndole los métodos `#center`, `#center:`, `#radius`, y `#radius:`.

```
center
  ^ center

center: aPoint
  center := aPoint

radius
  ^ radius

radius: aNumber
  radius := aNumber
```

`DrawableCircle` satisface el requerimiento `#drawOn:` del trait `TDrawing` proveyéndole el siguiente método.

```
drawOn: aCanvas
  aCanvas fillOval: self bounds
  color: Color black
```

Por último, el requerimiento `#bounds` de `TDrawing` es satisfecho automáticamente por `DrawableCircle` al hacer uso del trait `TCircle`.

1.3.5. Resolución de conflictos

Cuando dos o más traits son compuestos, puede que sus métodos conflictuen y éstos deben ser resueltos explícitamente por el programador.

Con este propósito existe el operador de exclusión, que permite excluir de la composición métodos de un trait. En caso de que no se desee perder el método al excluirlo, existe también un operador de aliasing que permite que un método esté disponible con otro nombre también.

El siguiente ejemplo muestra cómo resolver un conflicto entre dos traits que implementan el método `#hash`. En la composición, se excluye el método `#hash` de uno de los traits sin perderlo al mismo tiempo, ya que éste también está disponible con el nombre de `#circleHash`.

```
Objects
  subclass: #ColoredCircle
  uses: TColor + TCircle @ { #circleHash -> #hash } - { #hash }
  ...
```

1.4. Metodología utilizada

No basta encontrar alguna transformación que preserve el comportamiento del sistema, para decir que se ha encontrado un nuevo refactoring. Usualmente, nuevos refactorings surgen cuando una comunidad de programadores, encuentra que cierto tipo de problemas recurrentes

en diseño, pueden mejorase una y otra vez de la misma manera, casi de forma mecánica, y que el resultado es, generalmente, un sistema mejor diseñado.

Para asegurarnos de encontrar refactorings útiles y aplicables en una cantidad considerable de escenarios, decidímos obtenerlos a partir de casos de uso reales.

Los capítulos 4, 5 y 6 detallan diferentes casos de uso. En cada uno de ellos, se tomó un conjunto de clases del ambiente Squeak, en donde fuese razonable el uso de traits, y luego se refactorizó dicho conjunto de clases utilizando los trait refactorings obtenidos hasta el momento. En los casos donde se encontró que los trait refactorings implementados hasta el momento no eran adecuados a un nuevo escenario, se los modificaba generalizándolos para servir a éste, o se creaban nuevos trait refactorings de ser necesario.

Lo que se evitó durante todo el proceso, fue la intervención manual en las refactorizaciones. Es decir, si ninguno de los trait refactorings obtenidos hasta el momento servía para realizar cierta refactorización, y si tampoco era razonable la creación de un nuevo refactoring dado que éste sería muy específico al nuevo escenario, sólo en este caso se optaría por realizar la refactorización manualmente. Afortunadamente, no existió tal escenario, y en todos los casos se pudo refactorizar sólo aplicando los trait refactorings y sin intervención manual.

También en algunas ocasiones, se ha tomado más de un camino a la hora de refactorizar. Por ejemplo, aplicando en secuencia un conjunto de trait refactorings ya disponibles, y por otro lado creando uno nuevo, para luego verificar que con ambas refactorizaciones el resultado era el mismo.

Es así, como a través de estos casos de uso, se obtuvo la lista de trait refactorings presentados en el capítulo 2.

Por último, a esta metodología también se la acompañó de *Test Driven Development* [KB02]. En principio, cada vez que se creaba un nuevo trait refactoring, también se creaba un nuevo unit test, que describía tanto el escenario de donde el nuevo trait refactoring surgía, como lo que se esperaba de éste en dicho escenario.

Luego, por cada nuevo escenario que requería modificar o adaptar algún trait refactoring, se describía este escenario en forma de unit test, se corrían todos los unit tests y se verificaba que el recientemente agregado fuese el único en fallar. Entonces se hacían los cambios necesarios, se volvían a correr los unit tests y se verificaba que esta vez ninguno de ellos fallase.

Es así, como se dejó documentado, en forma de unit tests, todos los escenarios a los cuales se sometieron los trait refactorings, y qué resultados se esperaba de ellos en cada uno de éstos.

2. Trait refactorings

A continuación se listan los trait refactorings encontrados.

- Add method to trait
- Create missing requirements
- Move method to trait
- Extract method to trait
- Extract trait
- Flatten trait
- Replace inheritance with composition
- Replace dependency with composition
- Use trait

Todos ellos son el resultado de los casos de uso descriptos en los capítulos 4, 5 y 6.

2.1. Add method to trait

Este refactoring consiste en agregar un método `#m` a un trait `T`.

El refactoring agrega el método manteniendo la ya existente funcionalidad en los usuarios del trait y sin crear conflictos en éstos. Es decir, si algún usuario de `T` no define dicho método `#m`, o si lo define pero el nuevo método a agregar al trait conflictuará con éste, entonces el refactoring excluirá de la composición del usuario al nuevo método una vez que éste fue agregado al trait.

También se remueven de todos estos usuarios los métodos `#m` que comparten la misma implementación del trait. Para esto, sólo son considerados aquellos métodos cuyo árbol de expresión (parse tree) son exactamente iguales. Por lo cual, si el método `#m` de la clase y el trait hacen lo mismo, pero sus definiciones difieren ligeramente, la implementación de la clase se mantendrá.

De esta forma, el refactoring elimina en los usuarios del trait tantos métodos duplicados como puede, para dejar sólo el nuevo método que fue agregado al trait.

2.1.1. Ejemplo

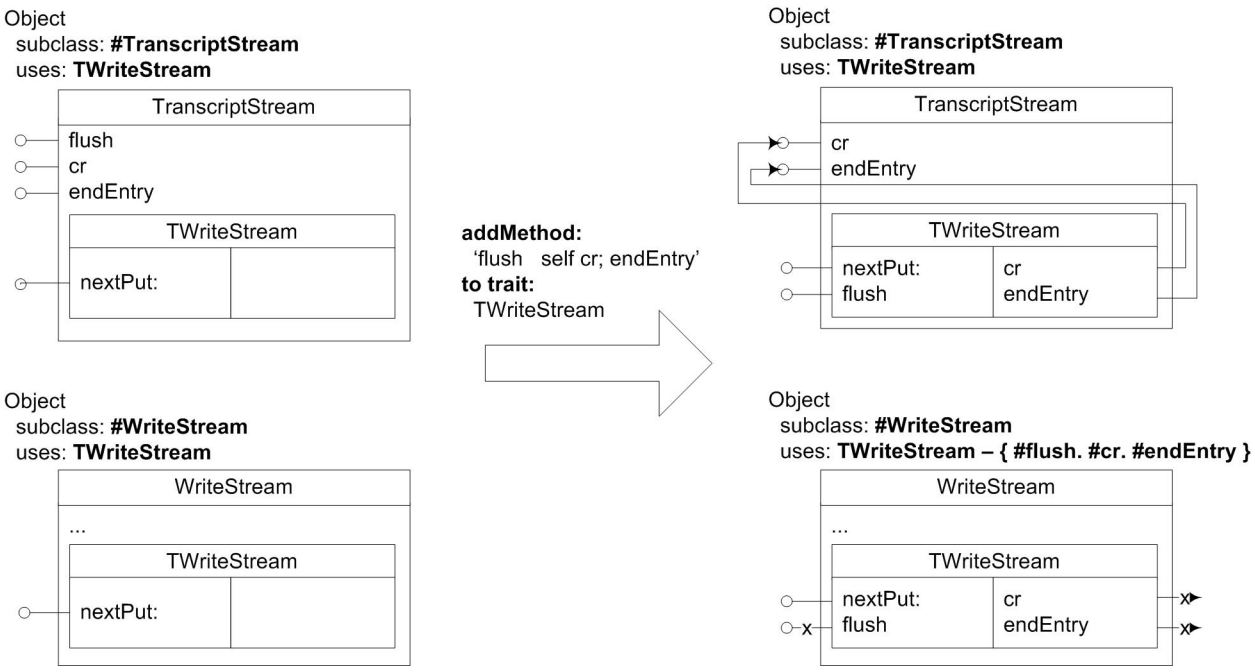

Figura 2.1: Refactoring *Add method to trait*

En el ejemplo, el refactoring agrega el método `#flush` al trait `TWriteStream`.

La clase `TranscriptStream` ya definía este método, y esta definición era exactamente la misma que la que se agregó en `TWriteStream`. Para no dejar ambas definiciones, el refactoring eliminó la definición local en `TranscriptStream`. De esta manera mantuvo la funcionalidad de esta clase al tiempo en que eliminó un método duplicado.

Por otro lado, la clase `WriteStream` no definía el método `#flush`. Por lo cual, el refactoring excluyó dicho método en esta clase para así mantener intacta su funcionalidad.

Por último, un paso adicional del refactoring fue crear los requerimientos `#cr` y `#endEntry`, ya que el trait no los define y ahora los requiere al haberse agregado el método `#flush`.

2.1.2. Motivación

Los refactorings *Move method to trait*, *Create missing requirements* y *Extract trait*, en cierto punto realizaban el siguiente conjunto de operaciones:

- Agregaban un nuevo método al trait con el que estaban operando.
- Removían todas las definiciones duplicadas entre el trait y sus usuarios, para sólo dejar la definición del trait.
- Todas estas operaciones se realizaban de forma tal de no crear conflictos en los ya existentes usuarios del trait.
- Por último, creaban los nuevos requerimientos en el trait, ya que un nuevo método había sido agregado a éste.

Habiendo observado que este conjunto de operaciones estaba implícito en estos tres refactorings (cada uno con su propia implementación) y que en todos ellos el conjunto de operaciones representaban lo mismo, es que se reificó este refactoring para no sólo darle un nombre, sino también unificar todas las operaciones en una sola implementación.

2.1.3. Precondiciones

El método a agregar no debe existir en el trait. En caso de existir, ambas definiciones, el método a agregar y el ya definido en el trait, deben ser las mismas (mismo parse tree).

2.2. Create missing requirements

Este refactoring consiste en hacer explícitos todos lo requerimientos implícitos en un trait.

Los requerimientos de un trait son todos aquellos métodos no provistos por éste, para los que se envía un mensaje a la pseudo-variable `self` desde un método sí provisto por él.

Agregar un requerimiento a un trait no es más que agregar un método a éste. Por consiguiente, agregar un requerimiento podría crear nuevos conflictos en los usuarios del trait. Para evitar esto, este refactoring agrega los requerimientos por medio del refactoring *Add method to trait*, que como vimos en la sección anterior, es una operación que no introduce nuevos conflictos.

2.2.1. Ejemplo

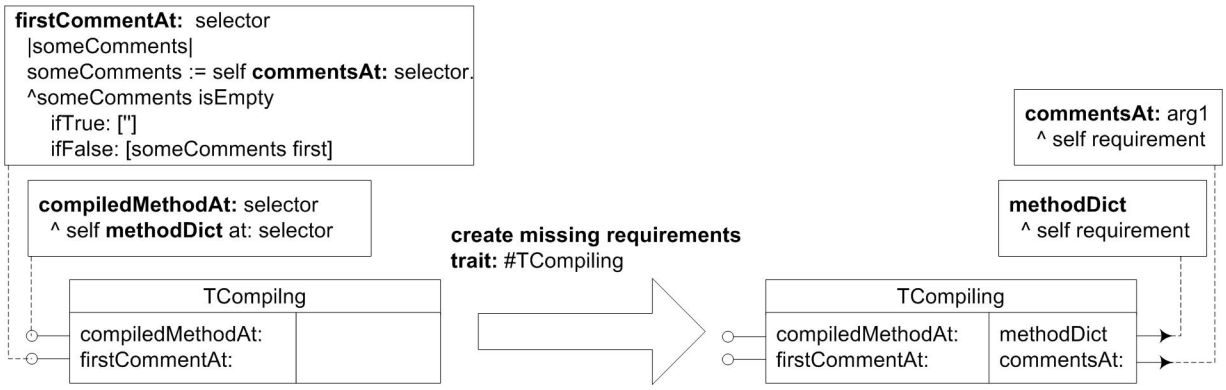

Figura 2.2: Refactoring *Create missing requirements*

En el ejemplo, el refactoring encuentra dos envíos de mensajes a `self`: `#commentsAt:` y `#methodDict`.

Ninguno de estos mensajes son provistos por el trait, por lo que el refactoring los creó como requerimientos.

2.2.2. Motivación

El capítulo 4 describe como se utilizaron los requerimientos como guía para diseñar un nuevo trait mientras se refactorizaba. En este proceso, se determinaban cuales eran los próximos métodos a ser movidos a un trait, estudiando los requerimientos de éste (que eran creados por este refactoring).

Si estuviésemos creando un trait y a priori no sabemos como va a evolucionar, se puede comenzar por agregarle un método. Luego creamos los requerimientos, y a partir de ellos podemos encontrar cuales son los próximos métodos que podemos agregar al trait. Por ejemplo, si sabemos que vamos a necesitar un trait `TCollection`, en donde no sabremos con seguridad que métodos contendrá, podemos comenzar agregandole un método inicial que estamos seguros deberá contener, por ejemplo `#detect:`.

```
detect: aBlock
  ^ self detect: aBlock ifNone: [self errorNotFound: aBlock]
```

Una vez agregado el método, ejecutando este refactoring encontraremos dos nuevos requerimientos: `#detect:ifNone:` y `#errorNotFound:`. Luego si encontramos que estos métodos pueden también implementarse en este trait, los implementamos, creamos nuevamente los requerimientos y repetimos el proceso.

De esta forma, se puede llegar a completar la implementación de este trait. En todo este proceso, los requerimientos creados por este refactoring, fueron la guía de implementación del trait.

2.2.3. Precondiciones

Este refactoring no tiene precondiciones, dado que sólo crea requerimientos y esto siempre es posible sin crear conflictos.

2.3. Move method to trait

Este refactoring consiste en mover la definición de un método de una clase a un trait presente en la composición de ésta.

Dado que los traits no pueden acceder directamente a variables de estado, todas las referencias directas a éstas son abstraídas al mover el método de la clase al trait. Esto también significa que cada variable de instancia abstraída se convierte en un requerimiento en el trait.

Por otro lado, mover el método de la clase al trait consiste en borrarlo de la clase y luego agregarlo al trait, o viceversa. Para agregar el método al trait, este refactoring utiliza internamente el refactoring *Add method to trait*, lo que implica que los nuevos requerimientos del trait son creados y que los métodos duplicados en sus usuarios son eliminados.

2.3.1. Ejemplo

Figura 2.3: Refactoring *Move method to trait*

En el ejemplo, el refactoring encuentra que el método `#setToEnd` referencia dos variables de instancia, `position` y `readLimit`.

```
setToEnd
  position := readLimit
```

Por lo tanto el refactoring las abstrae, quedando en el trait la siguiente definición.

```
setToEnd
  self position: self readLimit
```

Los métodos `#position:` y `#readLimit` son requerimientos del trait `TReadStream`, los cuales son satisfechos por la clase `ReadStream`, siendo uno un setter de la variable de estado `position`, y el otro un getter de la variable de estado `readLimit`.

Por otro lado, si la misma definición del método `#setToEnd` estuviese presente en otra clase usuaria de `TReadStream`, dicha duplicación se borraría luego de ejecutar este refactoring, ya que al mover el método al trait también se ejecutó implícitamente el refactoring *Add method to trait*.

2.3.2. Motivación

Este refactoring tiene sus orígenes en situaciones donde el programador va encontrando métodos repetidos en diferentes clases, que luego decide refactorizar en un trait.

Es decir el programador inicialmente encuentra dos clases con un mismo método repetido en cada una de ellas. Entonces crea un trait, compone ambas clases con él y luego usando este refactoring, mueve una de las dos implementaciones al trait, lo que remueve el código duplicado dejando una sola definición del método.

Habiendo concluido con esto, el programador puede ayudarse de los nuevos requerimientos para ver si existen más métodos que puedan ser movidos al trait, o buscar otros métodos repetidos (para mantener la cohesión, estos métodos deberian estar relacionados con los que el trait ya tiene) y continuar con la refactorización.

El capítulo 6 describe esta metodología aplicada a la creación de un trait extraído inicialmente de una clase.

2.3.3. Precondiciones

La clase de donde el método es tomado debe estar compuesta por el trait destino del método.

Además, el trait no debe definir este método o en caso de hacerlo, ambas definiciones, la de la clase y la del trait, deben ser las mismas.

Esta última precondición, resultó ser muy estricta a la hora de usar este refactoring (ver capítulo 4). Para relajar la precondición y hacer al refactoring más usable, se dejó en manos del programador la decisión de continuar o no con la ejecución del refactoring si ésta no se cumple.

A pesar de que las definiciones de dos métodos sean diferentes, pueden que hagan lo mismo. En estos casos será el programador quién, basado en su conocimiento, garantice la preservación del comportamiento al decidir sobreescribir el método del trait con una nueva definición.

2.4. Extract method to trait

Este refactoring consiste en tomar un fragmento de código del método de una clase, y con él crear un nuevo método en algún trait presente en la composición de la misma.

Para lo cual, el refactoring primero extrae el fragmento de código seleccionado utilizando el refactoring *Extract method*, y luego mueve el método extraído al trait utilizando el refactoring *Move method to trait*.

2.4.1. Ejemplo

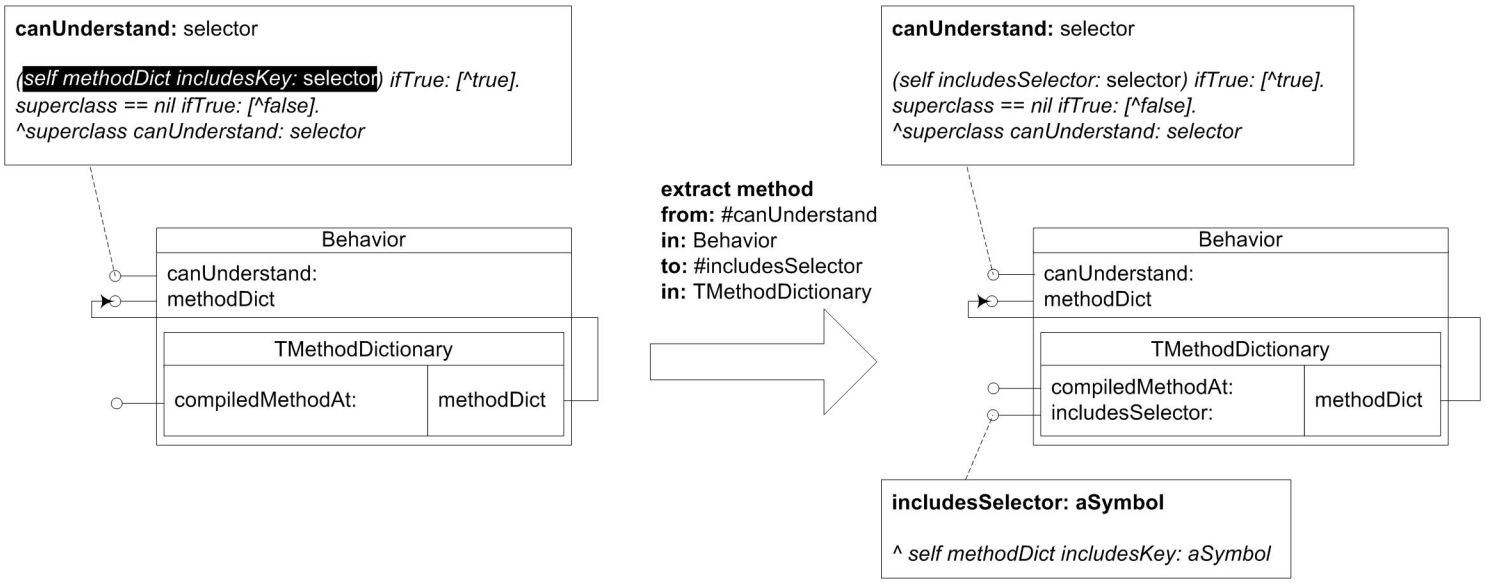

Figura 2.4: Refactoring *Extract method to trait*

El ejemplo muestra como un fragmento de código, tomado del método `#canUnderstand:` definido en la clase `Behavior`, es extraído como un nuevo método llamado `#includesSelector:` en el trait `TMethodDictionary`.

2.4.2. Motivación

Cuando un programador sospecha cierta duplicación de código entre dos o más clases (duplicación que puede ser refactorizada en un trait), puede ocurrir que éste no encuentre métodos exactamente duplicados entre estas clases. Aunque, es muy probable que, en principio, el programador sí encuentre fragmentos de código duplicados que sí puedan ser refactorizados en un trait. De no contar con este refactoring, el programador debería manualmente extraer estos fragmentos de código y luego moverlos al trait.

El capítulo 4 describe situaciones similares, a partir de las cuales se vió la necesidad de contar con este refactoring.

2.4.3. Precondiciones

Este refactoring no tiene precondiciones propias, sólo tiene las precondiciones impuestas por los refactorings *Extract method* y *Move method to trait*, que son ejecutados internamente.

2.5. Extract trait

Este refactoring consiste en tomar un conjunto de métodos de una clase y a partir de ellos extraer un trait.

Si el trait a ser extraído ya existe, el refactoring consiste en completar al trait con los métodos seleccionados. Es decir, el refactoring sólo saltéa su primer paso en la ejecución: la creación inicial del trait.

Por último, el refactoring puede ser configurado para hacer que la clase haga uso del trait una vez que éste fue extraído.

2.5.1. Ejemplo

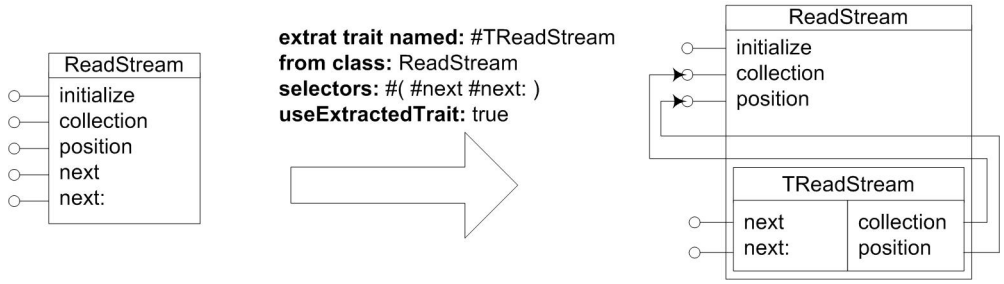

Figura 2.5: Refactoring *Extract trait*

El ejemplo muestra como el trait `TReadStream` es extraído a partir de los métodos `#next` y `#next:` de la clase `ReadStream`.

En este caso, el refactoring fue configurado para hacer que `ReadSteam` haga uso de `TReadStream` una vez que éste fue extraído.

2.5.2. Motivación

Como se puede ver en los capítulos 4, 5 y 6, muchas de las refactorizaciones con traits, comienzan por la extracción de un trait a partir de clases ya existentes.

2.5.3. Precondiciones

El trait a extraer no debe existir. En caso de existir, se deja en manos del programador la decisión de ejecutar el refactoring o no ejecutarlo en absoluto. En caso de decidir ejecutarlo, los métodos seleccionados para la extracción completarán al ya existente trait.

2.6. Flatten trait

La idea de este refactoring es opuesta a la del refactoring *Extract trait*.

Este refactoring aplana en una clase todos los métodos que un determinado trait le provee. Es decir, la clase definirá localmente todos los métodos que antes tenía definidos al hacer uso del trait. El refactoring logra esto compilando en la clase los métodos del trait. De esta forma, los métodos pasan a quedar definidos localmente en la clase.

Aquellos métodos que define tanto el trait como la clase, no son aplanados, ya que la clase dispone de una definición local para dicho método y no usa la definición provista por el trait.

Los métodos excluídos tampoco son aplanados, y los aliases son aplanados en la clase con el alias como selector.

Una vez aplanado el trait, éste es removido de la composición de la clase. Luego de esto, y en caso de no existir mas usuarios del trait, se remueve el trait del sistema si el programador así lo desea.

2.6.1. Ejemplo

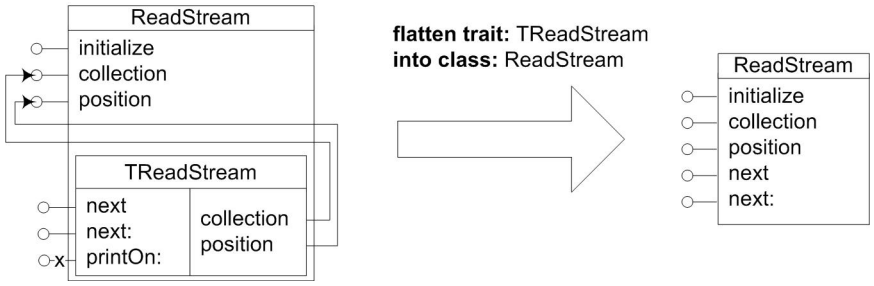

Figura 2.6: Refactoring *Flatten trait*

El ejemplo muestra como los métodos `#next` y `#next:` son aplanados en `ReadStream`, y también como el trait `TReadStream` se remueve de la composición de ésta.

A diferencia de `#next` y `#next:`, el método `#printOn:` no fue aplanado, ya que éste estaba excluido en la composición original.

2.6.2. Motivación

Este refactoring es particularmente útil en situaciones en donde el programador explora diferentes alternativas a la hora de refactorizar con traits. Los primeros traits extraídos o creados pueden no ser los más adecuados, y en estos casos el programador siempre puede dar un paso atrás aplicando este refactoring.

2.6.3. Precondiciones

No puede existir en la clase conflictos con el trait a aplanar, si existieran el refactoring no sabría si aplanar o no los conflictivos métodos.

2.7. Replace inheritance with composition

Este refactoring consiste en reemplazar una relación de herencia entre una subclase y su superclase, por una relación de composición entre un trait y dicha subclase. El trait provee el comportamiento que antes proveía la superclase.

Todos los métodos provistos por la superclase y deshabilitados en la subclase (`#shouldNotImplement`) se convierten en exclusiones del trait en la composición de dicha subclase.

En la subclase, las llamadas a `super` se resuelven con aliases. Es decir se transforman en llamadas a `self` pero con el método del trait renombrado.

Por último, las variables de estado en la superclase se pasan a definir en la subclase directamente. Esto se logra ejecutando el refactoring *Push down variable* por cada una de las variables de estado de la superclase.

2.7.1. Ejemplo

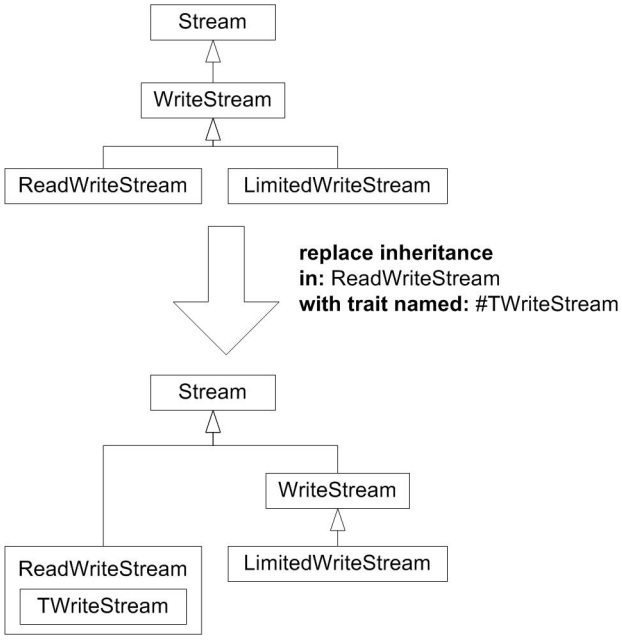

Figura 2.7: Refactoring *Replace inheritance with composition*

Las llamadas a `super` en `ReadWriteStream`, cuando `WriteStream` las define, se transforman en llamadas a `self` con el método renombrado. Por ejemplo, supongamos que tenemos el siguiente método en `ReadWriteStream`

```
ReadWriteStream >> position: anInteger
  self readLimit: (self readLimit max: position).
  super position: anInteger
```

y `WriteStream` define `#position:`. Entonces, luego de aplicar el refactoring, esto se transforma en

```
ReadWriteStream >> position: anInteger
  self readLimit: (self readLimit max: position).
  self writeStreamPosition: anInteger
```

siendo `#writeStreamPosition:` un alias para `#position:`

```
Stream
  subclass: #ReadWriteStream
  uses: TWriteStream @ { #writeStreamPosition: -> #position: }
  ...
```

Las llamadas a `super` en `WriteStream` quedan iguales en `TWriteStream`.

2.7.2. Motivación

Dado que la herramienta principal para compartir código entre clases hasta el momento era la herencia, es natural que encontremos muchas clases diseñadas con el único propósito de

poner en ellas todo el código en común entre muchas de sus subclases (aún cuando no exista una relación clara entre subclases y clase base).

Un ejemplo de esto es la clase `Object`. Esta clase, actualmente, tiene alrededor de 400 métodos. Muchos de estos métodos, como por ejemplo `#confirm:`, están en esta clase sólo para que cualquier subclase no tenga que implementarlos en caso de necesitarlos.

Reemplazar tales diseños basados en herencia por unos basados en composición de traits, puede resultar más beneficioso.

El capítulo 5, muestra cómo una relación de herencia entre `Magnitude` y sus subclases es reemplazada por una relación de composición de traits, y los beneficios del refactoring.

2.7.3. Precondiciones

Este refactoring no tiene precondiciones.

2.8. Replace dependency with composition

Este refactoring consiste en tomar dos traits presentes en la composición de una o más clases, y reorganizarlos de manera tal que las dependencias de un trait al otro desaparezcan. Básicamente, esto se logra haciendo que uno de ellos (el dependiente) incluya en su composición al otro (la dependencia).

2.8.1. Ejemplo

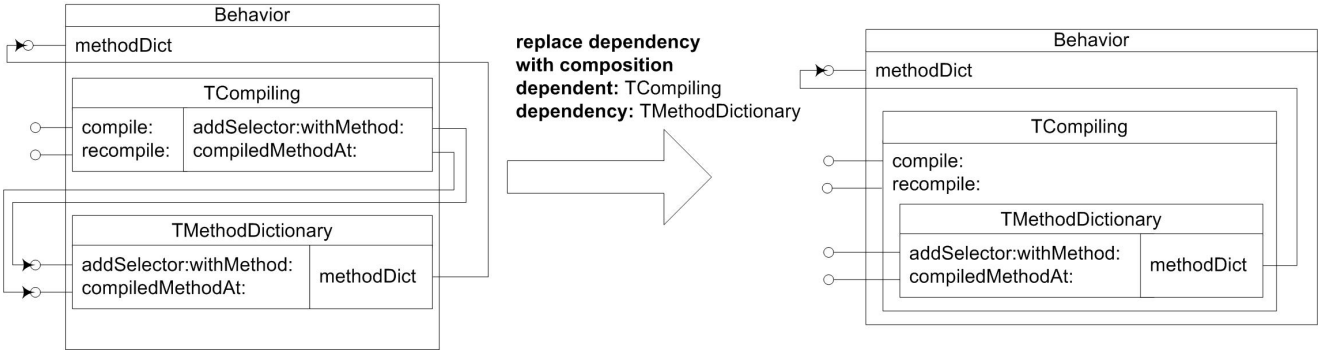

Figura 2.8: Refactoring *Replace dependency with composition*

En el ejemplo, el trait `TCompiling` depende del trait `TMethodDictionary` ya que los requerimientos del primero están satisfechos por el segundo.

El ejemplo también muestra como los requerimientos del trait `TCompiling` fueron removidos debido a que ahora éste provee los métodos al haber incluído a `TMethodDictionary`.

Las composiciónes de todas las clases también son modificadas para no incluir al trait `TMethodDictionary`. Por ejemplo, las siguientes clases

```
Object subclass: #Behavior
  uses: TCompiling - {#addSelector:withMethod:. #compiledMethodAt:} + TMethodDictionary
  instanceVariableNames: ...

Object subclass: #TraitBehavior
  uses: TCompiling - {#addSelector:withMethod:. #compiledMethodAt:} + TMethodDictionary
  instanceVariableNames: ...
```

luego del refactoring, cambiarían su definición por

```
Object subclass: #Behavior
  uses: TCompiling
  instanceVariableNames: ...

Object subclass: #TraitBehavior
  uses: TCompiling
  instanceVariableNames: ...
```

pero mantendrían su actual funcionalidad.

2.8.2. Motivación

El capítulo 6 describe una situación en la que se habían extraído dos traits de una clase y luego se encontró que componer uno de ellos con el otro resultaba en un mejor diseño todavía, dado que gran cantidad de requerimientos dejaban de ser necesarios en este nuevo diseño.

También, en la sección 4.4 Improvements at the Trait-Level de [Lien04], se repite el mismo escenario. En este trabajo se encuentra que gran número de los requerimientos de un trait, son satisfechos por otro trait al mismo nivel de composición. Luego de remarcar las desventajas de tal dependencia entre traits, el trabajo describe cómo es que este problema fue solucionado haciendo que el trait dependiente incluya la dependencia. Es decir, en este trabajo, se realizó la misma refactorización, aunque en forma manual.

2.8.3. Precondiciones

En toda composición donde esté el dependiente, debe estar la dependencia también. Es decir, no puede ocurrir que una clase use el trait dependiente y no la dependencia.

2.9. Use trait

Este refactoring consiste en agregar un trait a la composición de una clase y eliminar de la clase todos aquellos métodos duplicados con el trait recién incluido.

Al igual que en el refactoring *Add method to trait*, los métodos considerados para la eliminación son aquellos cuyas definiciones son exactamente iguales a las del trait. Es decir, sólo aquellos métodos cuyo árbol de expresión (parse tree) son exactamente iguales, se consideran duplicados y todas estas definiciones son removidas de la clase.

2.9.1. Ejemplo

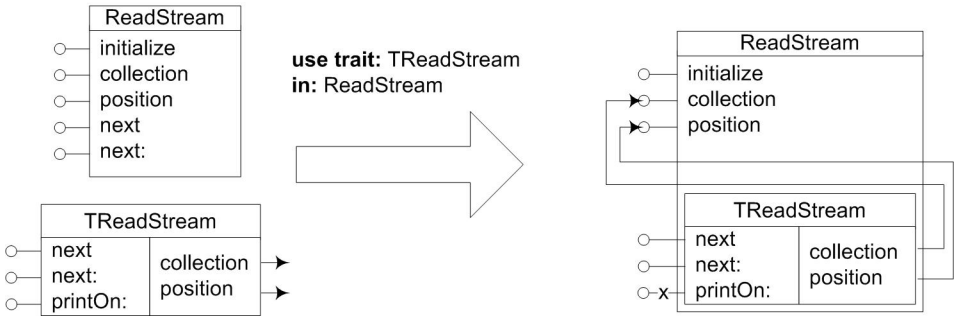

Figura 2.9: Refactoring *Use trait*

En el ejemplo, tanto el método `#next` como `#next:`, tienen exactamente la misma definición en el trait `TReadStream` como en la clase `ReadStream`. Es por esto que el refactoring, luego de incluir el trait en la composición de `ReadStream`, eliminó las definiciones locales de estos métodos.

2.9.2. Motivación

Este refactoring complementa a *Extract trait* dado que en muchas de las ocasiones, cuando el programador extrae un trait, lo hace pensando en incluirlo en otras clases también.

Este refactoring también resulta útil en situaciones donde el programador, ya teniendo el trait, encuentra una clase que podría beneficiarse de éste. En estos casos, puede rápidamente incluir el trait en la clase haciendo uso de este refactoring y evaluar su conveniencia o no. En caso de no resultar en el diseño esperado, siempre puede deshacer los cambios aplicando el refactoring *Flatten trait*.

2.9.3. Precondiciones

Este refactoring no posee ninguna precondición. Siempre es posible agregar un trait a la composición de una clase sin crear conflictos y manteniendo el comportamiento provisto por la misma. En el peor de los casos, el refactoring excluirá todos los métodos del trait para no crear conflictos.

3. Implementación

En esta sección se detallan los principales objetos que componen al motor en el cual todos lo refactorings son implementados, los cambios necesarios a dicho motor para poder implementar los trait refactorings y en qué forma estos objetos colaboran para llevar a cabo las refactorizaciones.

3.1. Refactoring Objects

Cada subclase de `Refactoring` reifica el concepto de un refactoring en particular. Básicamente un refactoring colabora con instancias de `RBNamespace`, `RBClass` y `RBMethod` para llevar a cabo las transformaciones sobre el sistema.

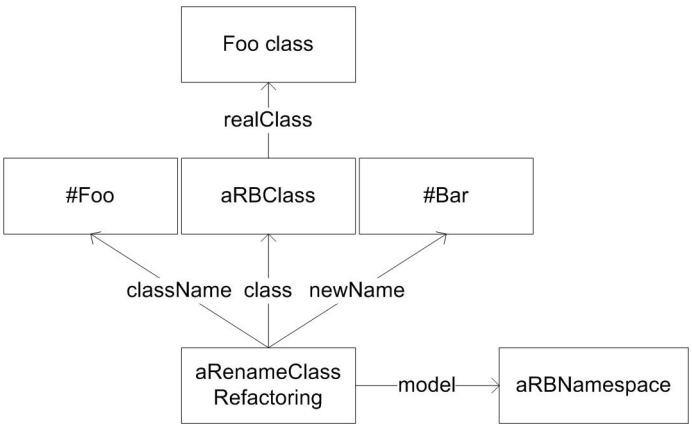

Figura 3.1: Instancia de una subclase de `Refactoring`

Para la inclusión de los nuevos refactorings la jerarquía de `Refactoring` no tuvo que ser cambiada, sólo fue extendida con nuevas subclases.

Siguiendo el patrón provisto por el motor de refactorings, la implementación de cada nuevo refactoring sólo implicó definir su comportamiento en básicamente dos métodos: `#preconditions` y `#transform`. Estas dos son las únicas responsabilidades a ser definidas por cada subclase de `Refactoring`.

3.2. RB Objects

`RBNamespace` representa un espacio de nombres. Cada clase agregada o borrada de un `RBNamespace`, no genera cambios en el sistema.

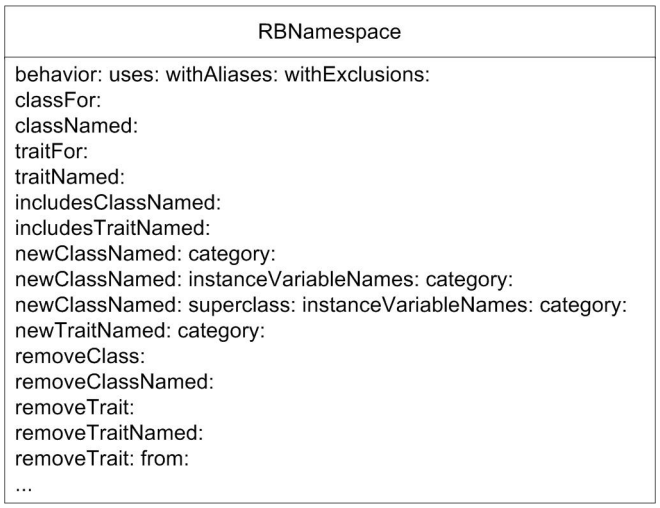

Figura 3.2: Clase `RBNamespace`

`RBAbstractCondition` y sus subclases representan distintos tipos de precondiciones a ser cumplidas por un refactoring. Básicamente funcionan como formulas proposicionales que colaboran con instancias de `RBNamespace`, `RBClass` y `RBMethod` para ser evaluadas.

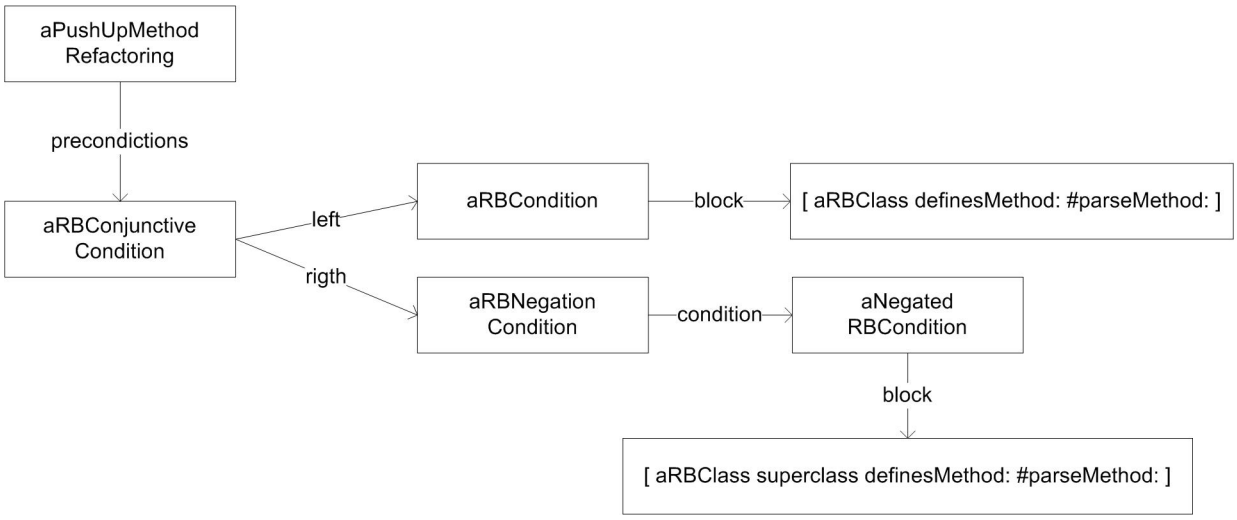

Figura 3.3: Objetos precondiciones de un refactoring

`RBClass` y `RBMetaClass` representan clases y meta clases. La principal diferencia entre la conocida clase `Class` y `RBClass` es que toda operación realizada en esta última no afecta al sistema pero sí al `RBNamespace` que la contiene. Lo mismo sucede con `RBMetaClass`

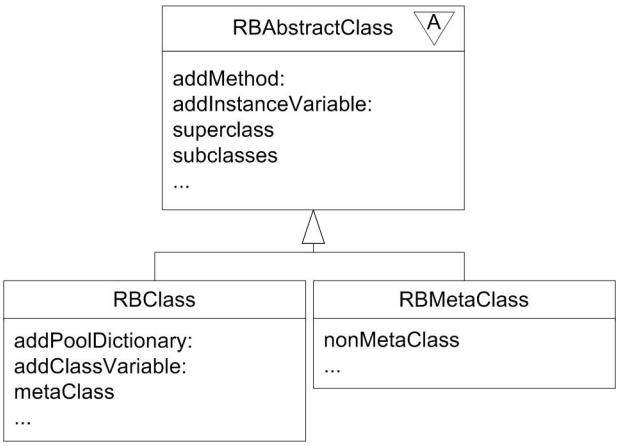

Figura 3.4: Clase `RBAbstracClass` y subclases

`RBMethod` representa un método y tampoco modifica el sistema, sólo a la `RBClass` que la contiene, que por consiguiente modifica al `RBNamespace` que la contiene.

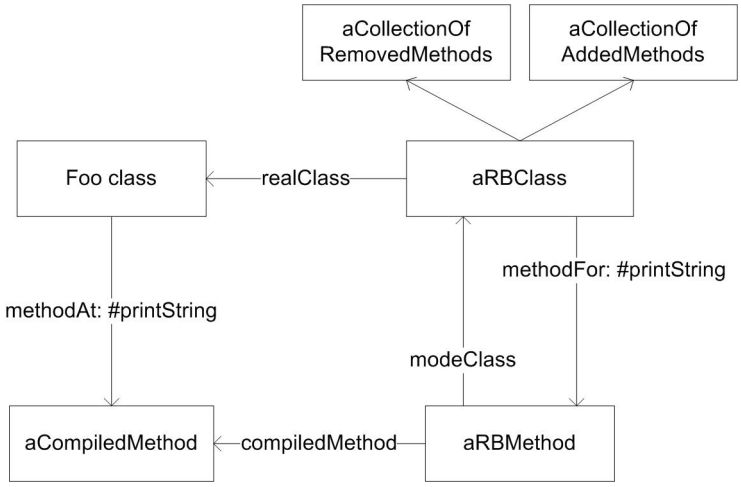

Figura 3.5: Instancia de `RBClass` representando una clase existente

Para la inclusión de los nuevos trait refactorings, esta jerarquía de RB objects no sólo tuvo que ser extendida, sino que también ha tenido que cambiar.

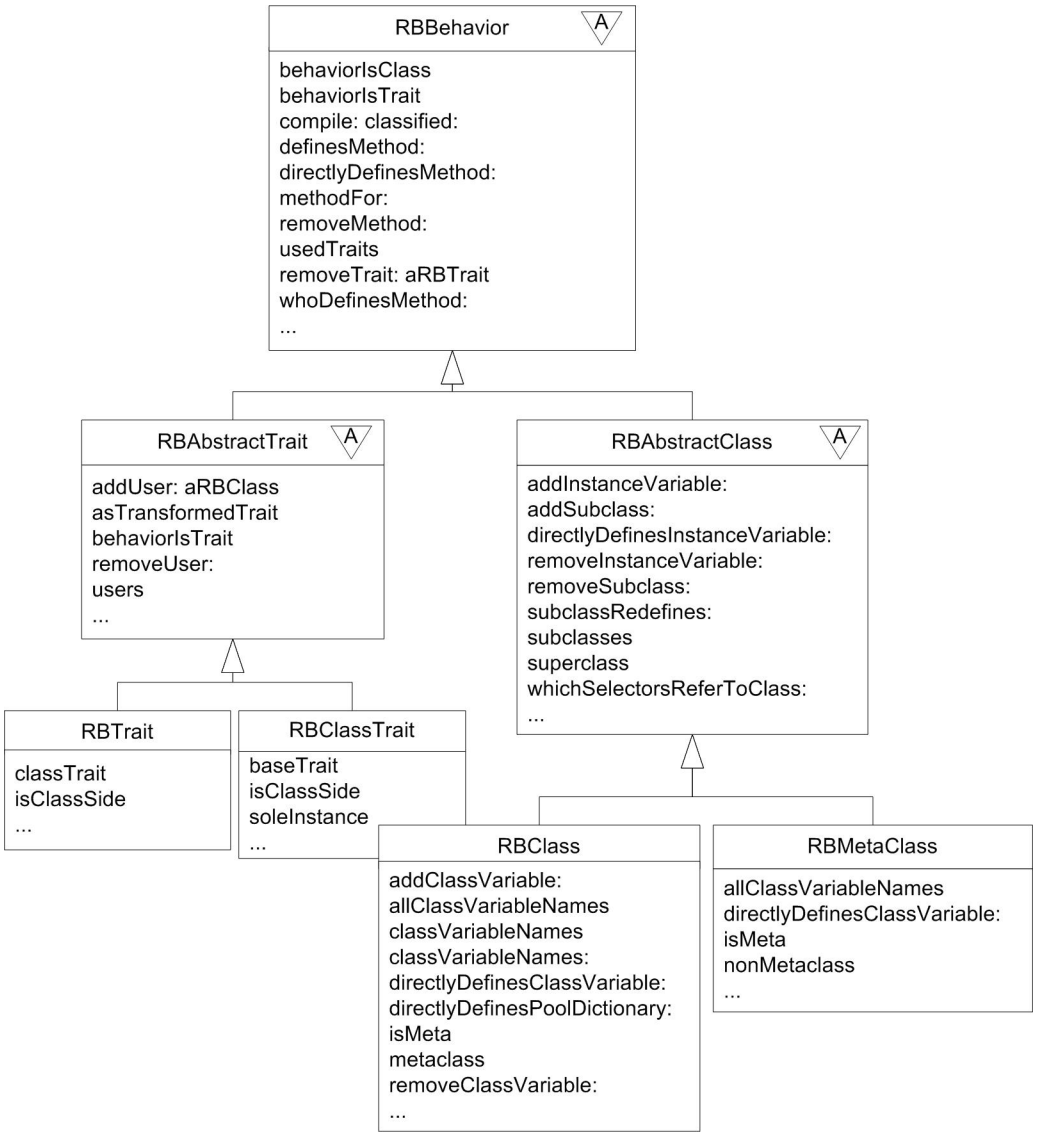

Figura 3.6: Jerarquía de `RBAbstracClass` modificada para la inclusión de traits

También fue necesaria la creación de una clase cuyas instancias fuesen el reflejo de un trait transformado por operaciones de exclusión y aliasing al momento de ser compuesto en una clase u otro trait. Esta clase es `RBTransformedTrait`.

Figura 3.7: Clase `RBTransformedTrait`

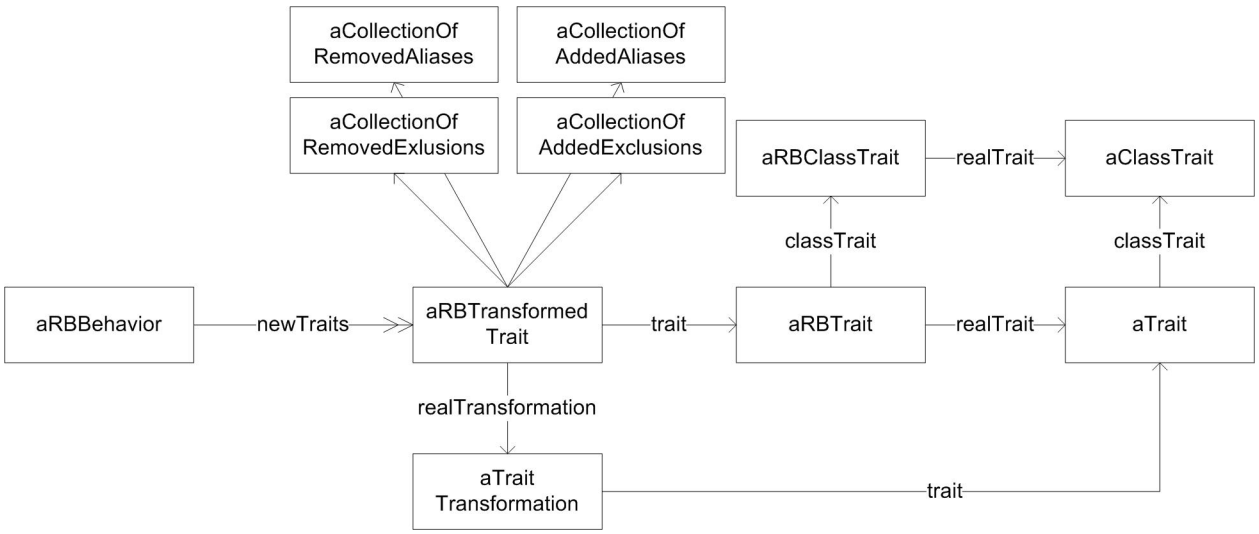

Figura 3.8: Instancia de `RBTrait` representando un trait existente

3.3. Change Objects

Cada subclase de `RefactoryChange` representa un cambio sencillo al sistema, como ser la creación de una clase, el borrado de una variable de instancia, el agregado de un método a una clase, etc.

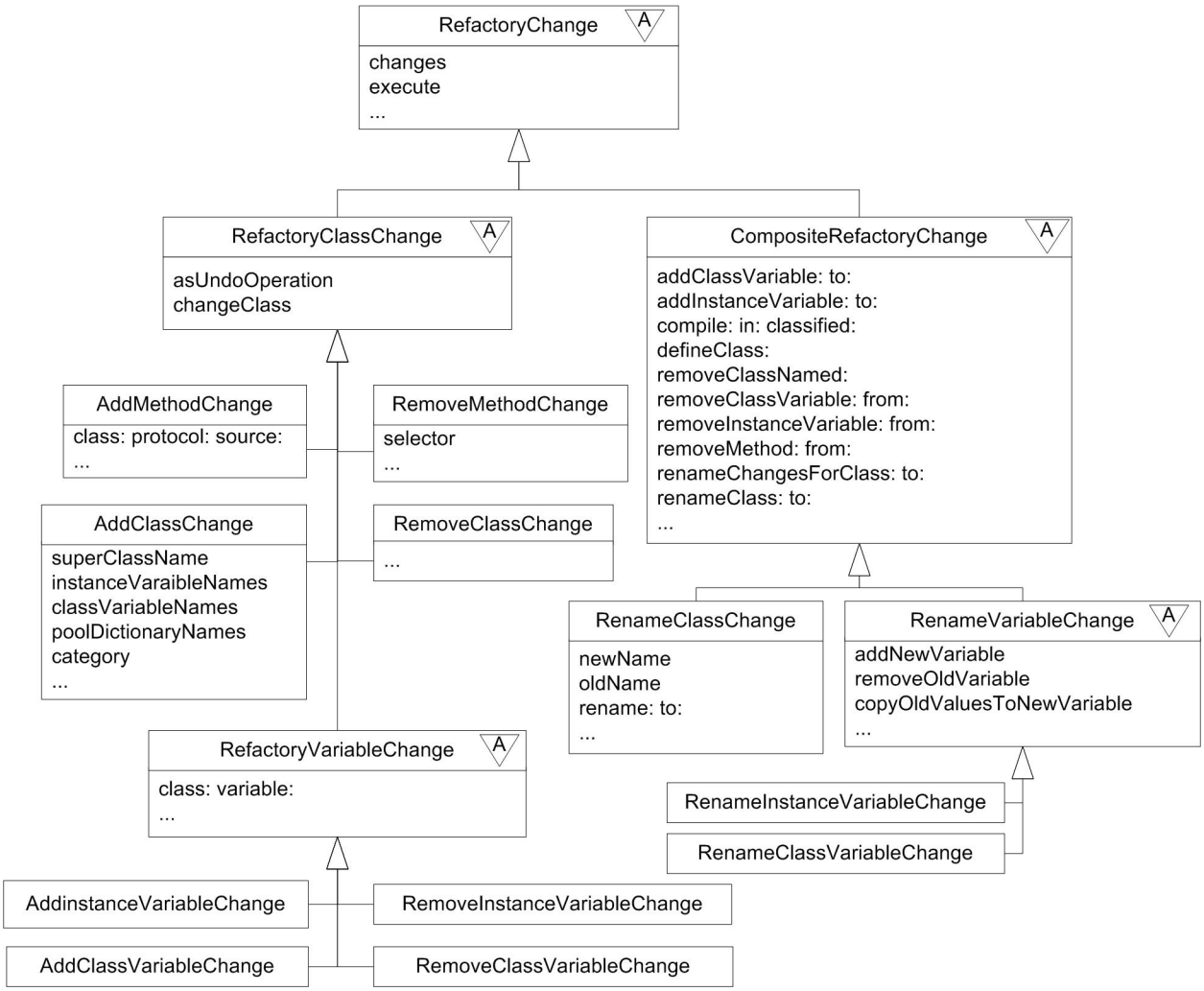

Figura 3.9: Clase `RefactoryChange` y subclases

Dentro de esta jerarquía existe una subclase que merece ser mencionada en forma particular: `CompositeRefactoryChange`. La mayoría de las subclases de `CompositeRefactoryChange` son clases que representan renombramientos a cierto tipo de objeto, como por ejemplo `RenameClassChange`, `RenameInstanceVariableChange` y `RenameClassVariableChange`. Estos change objects se destacan del resto por que además de representar un cambio en el sistema, también contienen en sí mismos referencias a otros change objects, haciendo de estos una especie de objetos compuestos.

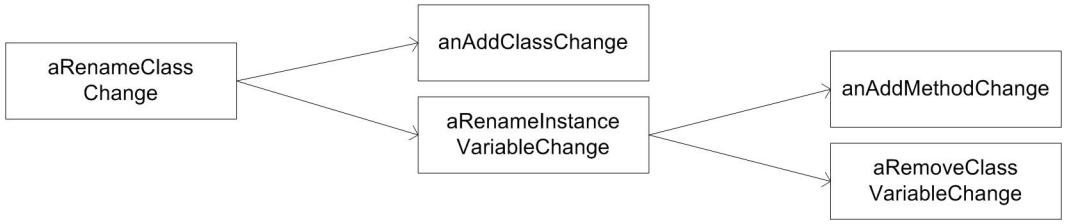

Figura 3.10: Instancia de una subclase de `CompositeRefactoryChange`

Para la inclusión de los nuevos trait refactorings esta jerarquía no sólo tuvo que ser extendida, sino que también ha tenido que cambiar para poder incluir cuatro nuevas clases relacionadas con traits:

- `AddTraitChange` para la creación de un nuevo trait,
- `RemoveTraitChange` para el borrado de un trait existente,
- `UseTraitChange` para agregar un trait, con sus exclusiones y aliases, a la composición de otro trait o clase y por último
- `RemoveUsedTraitChange` para remover un trait de la composición de otro trait o clase.

Otro de los motivos para modificar esta jerarquía de clases fue para poder generalizar clases como `RemoveMethodChange` y `AddMethodChange` ya que no aplicaban sólo a clases, sino que también aplicaban a traits.

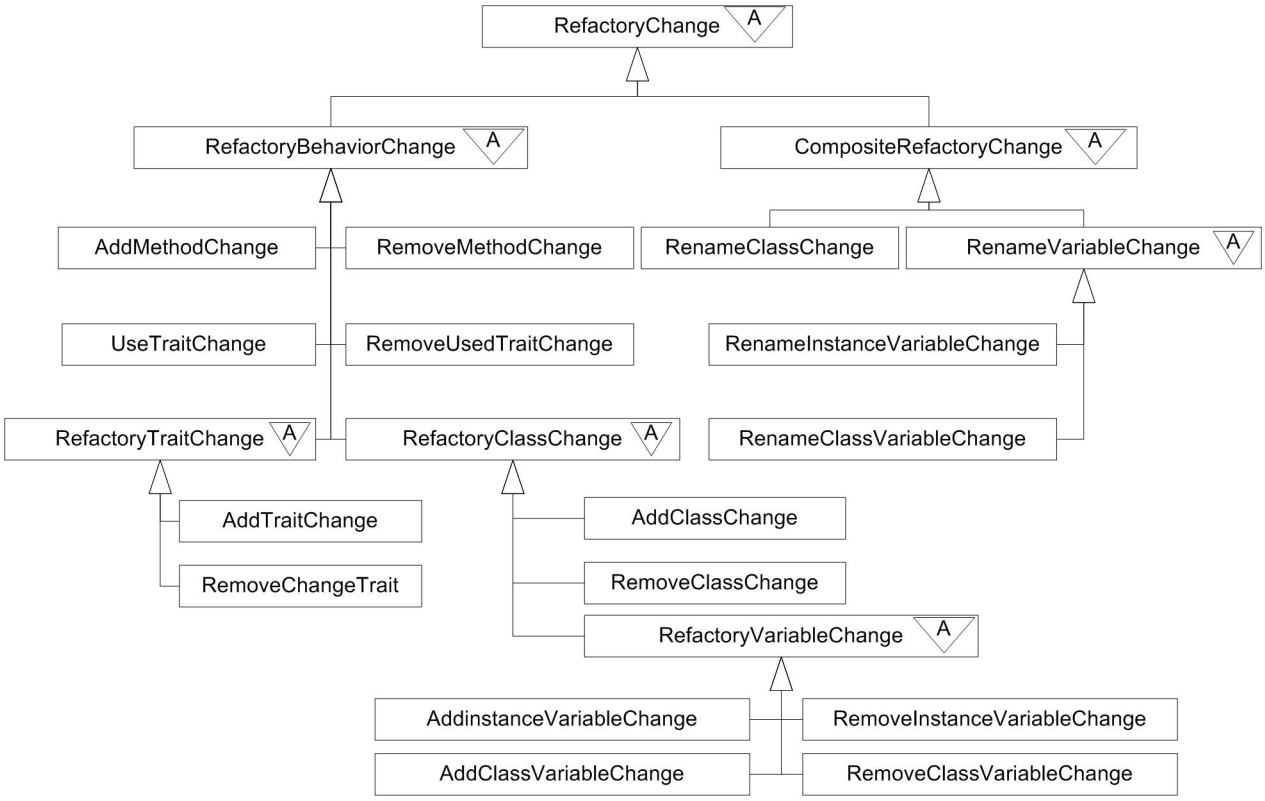

Figura 3.11: Jerarquía de `RefactoryChange` modificada para la inclusión de traits

3.4. RefactoringManager Object

La única instancia de `RefactoringManager` mantiene una lista de los refactorings ejecutados hasta el momento.

La única instancia de `RefactoryChangeManager` es responsable de realizar en el sistema, los cambios producidos por los refactorings ejecutados hasta el momento. Es también responsable

de mantener dos listas para poder deshacer y rehacer los cambios producidos por los últimos refactorings ejecutados.

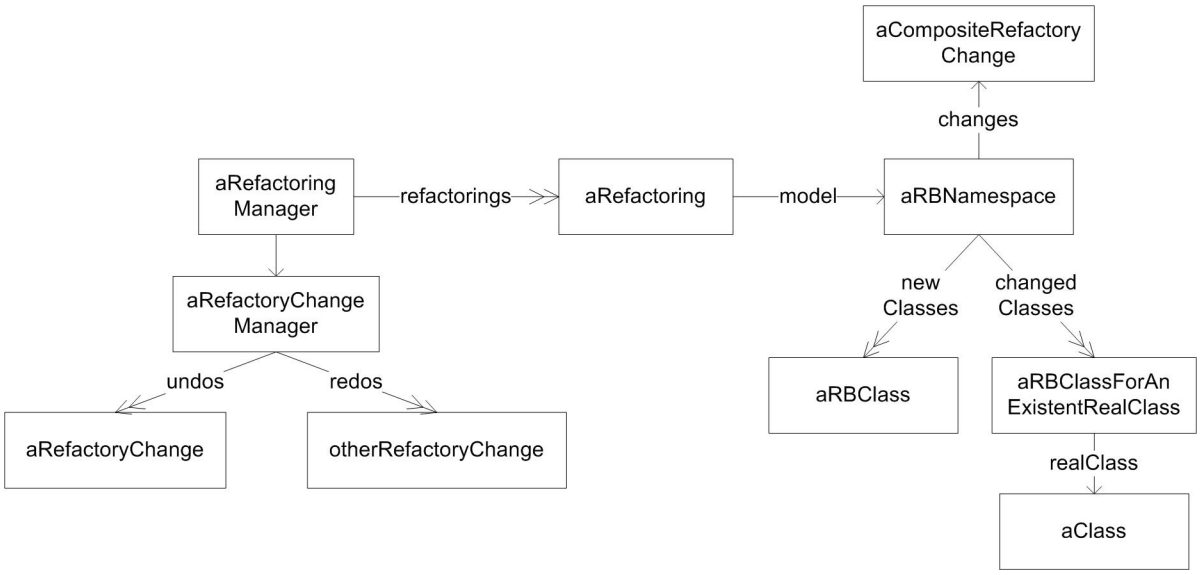

Figura 3.12: Única instancia de `RefactoryChangeManager`

Para la inclusión de los nuevos refactorings, estas clases no necesitaron cambios ni extensiones.

3.5. Principales Colaboraciones

Los refactorings cumplen con sus responsabilidades (la de refactorizar) en dos pasos. Esto se puede ver al observar el método `Refactoring≫primitiveExecute`.

```
Refactoring >> primitiveExecute
  self checkPreconditions.
  self transform
```

El primer paso chequea si están dadas las condiciones necesarias para que el refactoring pueda cumplir con su responsabilidad sin alterar el comportamiento del sistema siendo refactorizado. Si no están dadas las precondiciones necesarias, `#checkPrecondicions` lanzará una excepción conteniendo la razón por la cual las precondiciones no fueron cumplidas.

El segundo paso consta de realizar las transformaciones necesarias para llevar a cabo el refactoring: crear una clase, renombrar un método, borrar un variable de instancia, etc.

Todas las transformaciones o cambios producidos por el refactoring no son hechos sobre el sistema. Esto significa que si desde un workspace evaluamos lo siguiente

```
refactoring := PushDownMethodRefactoring
      pushDown: #(#name:)
      from: LintRuleTest.
refactoring primitiveExecute.
```

y luego abrimos algún browser de clases, no veremos los resultados del refactoring recientemente ejecutado. Esto se debe a que ningún refactoring opera o realiza sus transformaciones directamente sobre el sistema y sus clases.

Los refactorings siempre operan sobre un conjunto de objetos que representan a las clases, las metaclases y los métodos pero que no son realmente clases, metaclases y métodos. Es decir, operan con instancias de `RBClass`, `RBMetaClass` y `RBMethod`. A estos objetos de ahora en adelante los llamaremos "reflejos de".

Estos reflejos de clases, metaclases y métodos con los que un refactoring opera, son a su vez producidos por otro objeto: el reflejo de un espacio de nombres. Es decir, una instancia de `RBNamespace`.

Entonces los refactorings sólo conocen el reflejo de un espacio de nombres y obtienen de éste los reflejos de clases, metaclases y métodos sobre los que se realizarán las transformaciones.

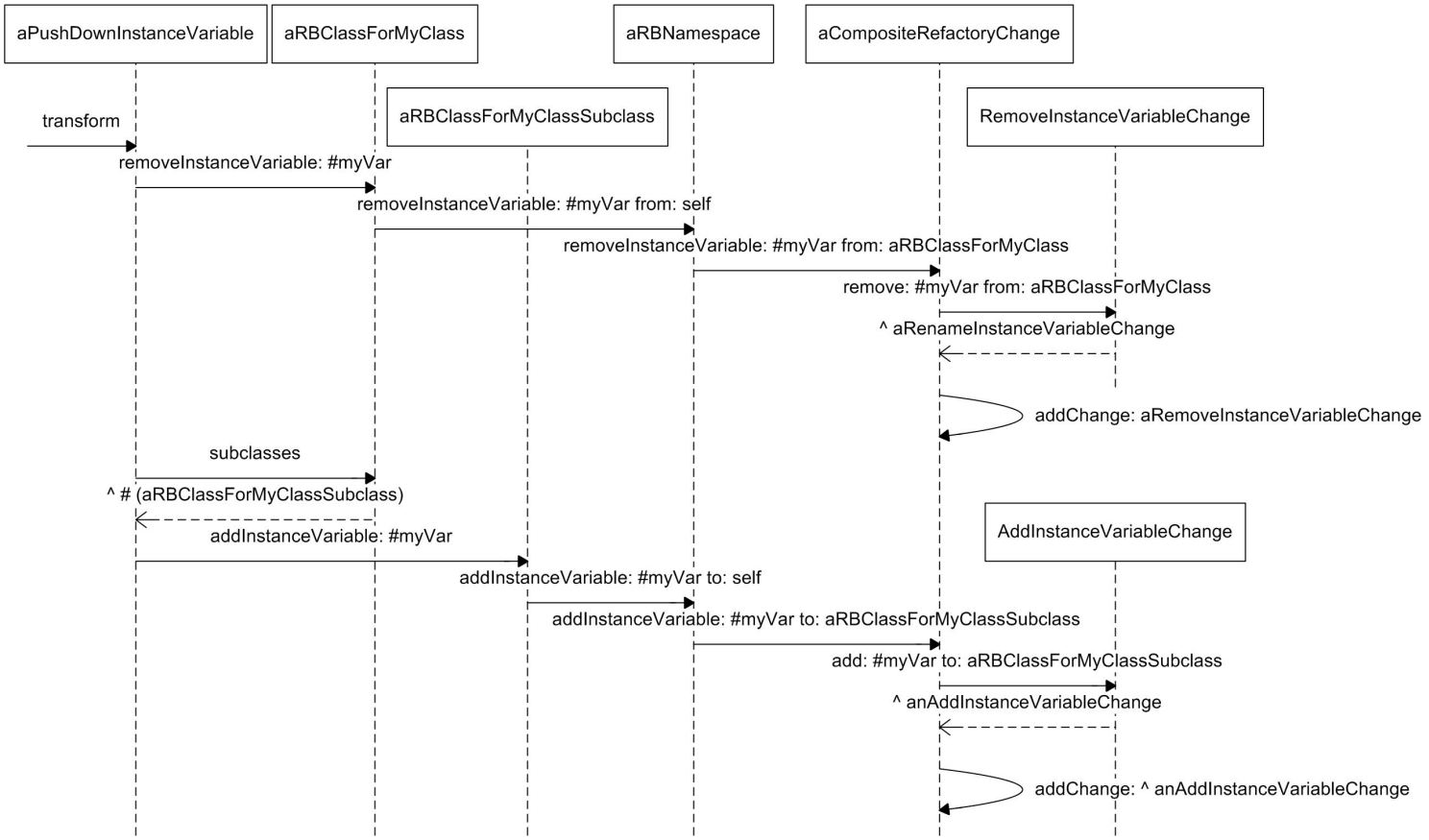

Figura 3.13: Refactoring realizando transformaciones

Una vez concluida la transformación, el reflejo del espacio de nombres tendrá listo un colaborador interno instancia de `CompositeRefactoryChange`, que representará a todos los cambios a realizar en el sistema. Este objeto es una composición de los que llamaremos "change objects".

Estos change objects representan operaciones pequeñas a realizar sobre un sistema. La principal diferencia entre éstos y un refactoring, es que los primeros no tienen precondiciones dado que las operaciones que realizan son demasiado sencillas y además estos change objects operan directamente sobre el sistema. Otra particularidad que los distingue de un refactoring, es la característica de poder ser "reversibles" por medio del mensaje `#asUndoOperation`. Por ejemplo, un change object que agrega una variable de instancia responde al mensaje

`#asUndoOperation` con un change object opuesto que la remueve.

El encargado inicial de hacer que estos change objects sean ejecutados y que el refactoring sea finalmente aplicado sobre el sistema, es el mismo refactoring en el método `Refactoring≫execute`.

```
Refactoring >> execute
  self primitiveExecute.
  RefactoringManager instance addRefactoring: self
```

Aunque la ejecución de los change objects producidos, es también resultado de la colaboración de otros dos objetos: un refactoring manager y un changes manager.

```
RefactoringManager >> addRefactoring: aRefactoring
  RefactoryChangeManager instance performChange: aRefactoring changes.
  refactorings add: aRefactoring class name

RefactoryChangeManager >> performChange: aRefactoringChange
  self ignoreChangesWhile: [ self addUndo: aRefactoringChange execute ]
```

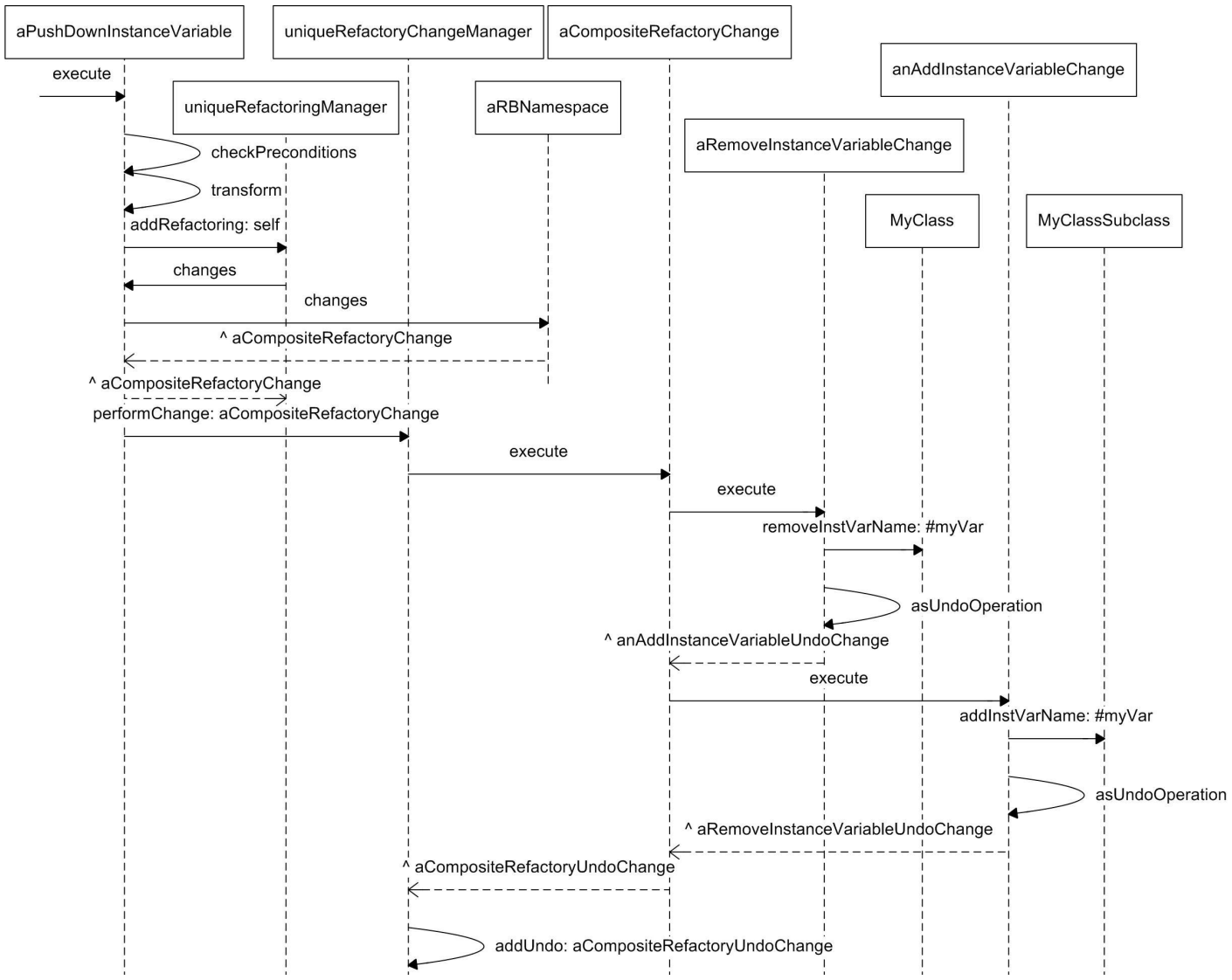

Figura 3.14: Realizando los cambios producidos por un refactoring

Una vez realizada la verdadera transformación en el sistema, el change manager dispone de una serie de change objects para deshacer las operaciones a pedido. Por ejemplo, evaluando el siguiente código desde un workspace, podríamos deshacer los cambios producidos por el último refactoring ejecutado.

```
RefactoryChangeManager instance undoOperation
```

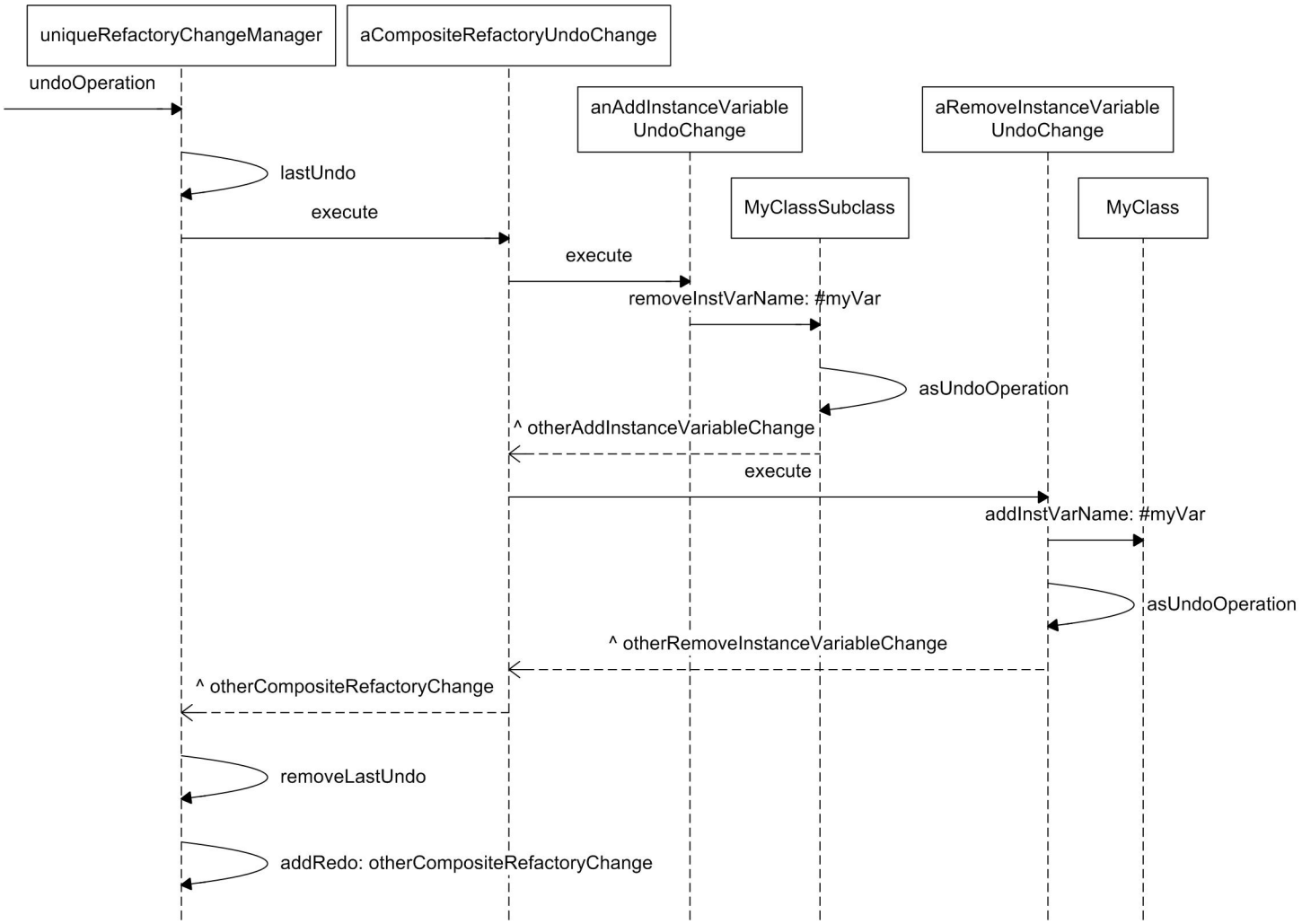

Figura 3.15: Deshaciendo los cambios producidos por un refactoring

4. Caso 1. Refactoring browser

Una de las herramientas más importantes para trabajar en Smalltalk es el browser del sistema (figura 4.1). Este browser permite que el programador pueda navegar por las clases y traits definidos en el sistema, ver cómo están organizadas y agrupadas, e inspeccionar y modificar código.

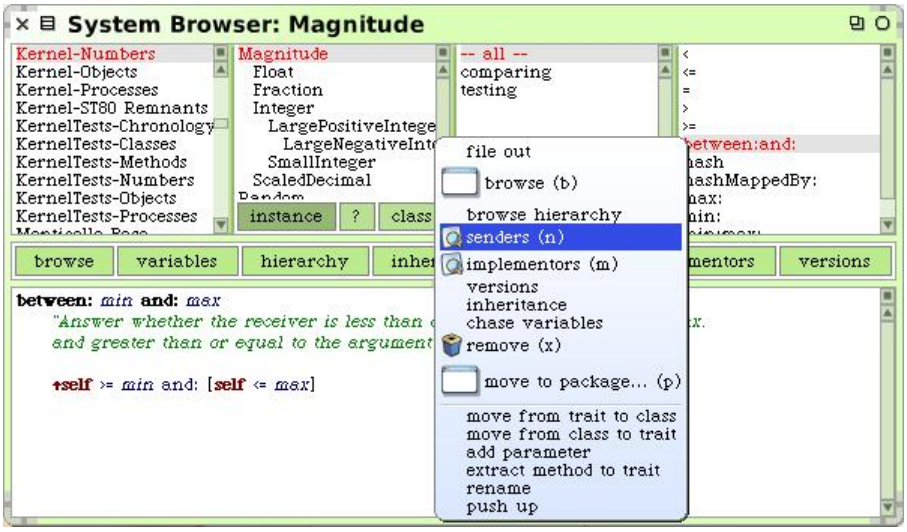

Figura 4.1: Browser del sistema

El Refactoring Browser (figura 4.2) es un browser del sistema que además incluye operaciones semi-automáticas de refactorings, tales como *Push up method*, *Push down instance variable*, *Create sibling class*, etc [RBJ97].

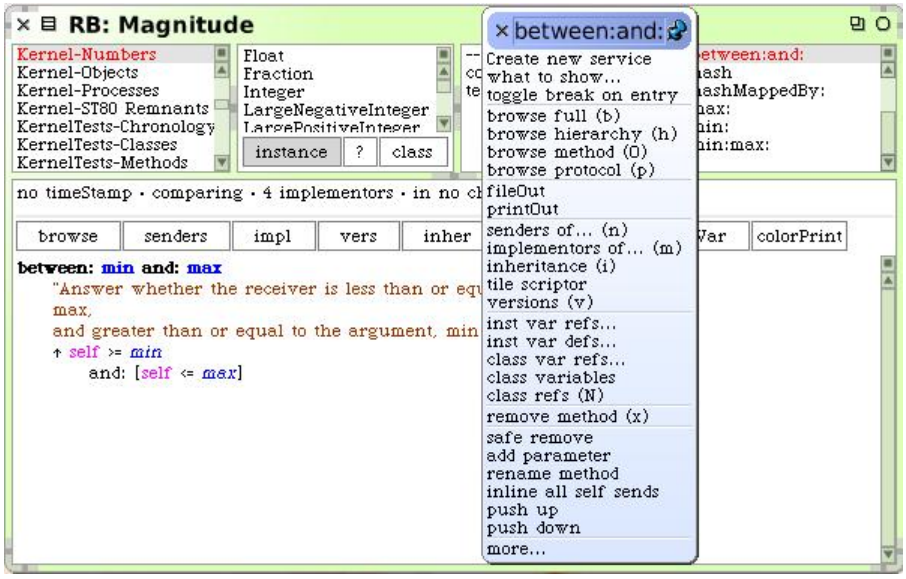

Figura 4.2: Refactoring Browser

4.1. Motivación

El Refactoring Browser era el único conectado al motor de refactorings, en consecuencia el único que proveía operaciones de refactorings desde una interfaz gráfica.

Los trait refactorings en principio habían sido integrados a este browser. Al ser éste el único conectado al motor de refactorings, el costo de agregarle unos pocos refactorings era mucho menor al costo de tener que conectar dicho motor a otro browser para luego poder agregarle los trait refactorings.

Por otro lado, existe otro browser del sistema, llamado Omnibrowser [OmniBr]. Este browser es más sencillo, cómodo e intuitivo que el refactoring browser, pero una desventaja del Omnibrowser era la de no estar integrado al motor de refactorings, sin proveer operaciones de refactorings desde menúes, botones o alguna otra forma de interacción con el usuario.

Había que comenzar a realizar refactorizaciones en diferentes lugares del sistema y así poder obtener resultados para este trabajo. Esto nos obligaba a trabajar intensivamente con el refactoring browser, que como se comentó no es tan cómodo o conveniente como el Omnibrowser.

Es por esto que como primer caso se decidió refactorizar el refactoring browser y el Omnibrowser para que este último quede integrado al motor de refactorings, y así poder trabajar las futuras refactorizaciones sólo desde el Omnibrowser y al mismo tiempo mantener la actual funcionalidad del refactoring browser.

4.2. Extrayendo un trait inicial

Se comenzó eligiendo un método conocido de la clase `RefactoringBrowser`, `#flattenTraitInto:`. A partir de allí se extrajo un trait llamado `TRefactoringBrowser` para contener dicho método.

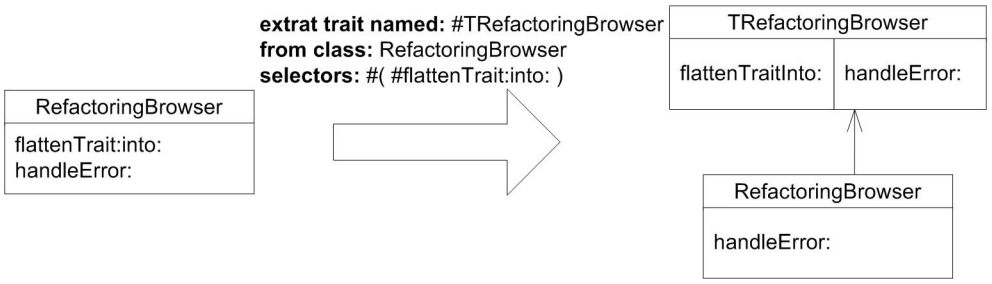

Figura 4.3: Extrayendo trait `TRefactoringBrowser`

4.3. Moviendo métodos al trait

A partir de ese momento, se revisó el trait creado para ver que requerimientos contenía y ver si alguno de estos requerimientos podían llegar a extraerse de la clase `RefactoringBrowser`. Algunos requerimientos pudieron ser extraídos de esta clase, como por ejemplo `#handleError:`.

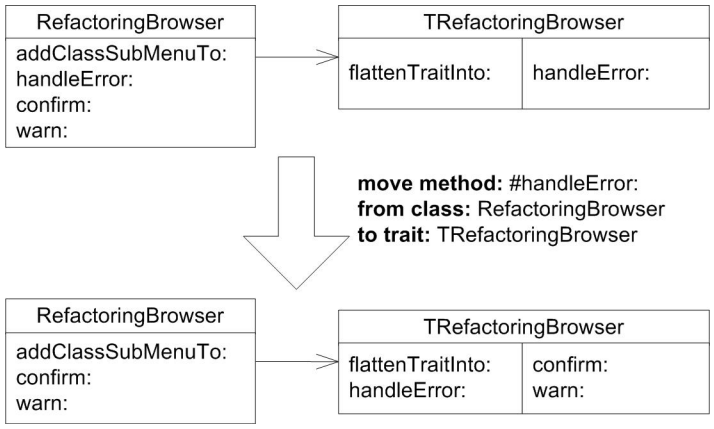

Figura 4.4: Moviendo a `TRefactoringBrowser` la implementación provista por la clase `RefactoringBrowser` del requerimiento `#handleError:`

Estos primeros pasos realizados con los refactorings fueron de gran ayuda a la hora de evaluar la conveniencia o facilidad de uso de los trait refactorings. Por ejemplo, una de las precondiciones del refactoring *Move method to trait* era que el método a ser movido no podía estar definido en el trait destino, para evitar así sobreescribir dicha definición. Esta precondición resultó ser una restricción a la hora de mover el método `RefactoringBrowser≫hadleError:` al trait `TRefactoringBrowser`, dado que dicho método ya existía como requerimiento en este trait. Es así como se cambió el refactoring para que esta precondición no aplique, si el método existe en el trait destino y es un requerimiento.

4.4. Repitiendo el proceso

Una vez mejorada la precondición y ejecutado el refactoring, se revisó otra vez la lista de requerimientos del trait para ver cuáles eran nuevos, en este caso se encontró `#confirm:` y `#warn:`.

Estos dos métodos también fueron movidos al trait y luego se repitió el proceso hasta encontrar que no había más requerimientos cuyas implementaciones pudieran ser movidas de la clase al trait. Llegado a ese punto se eligió otro método que todavía no estuviese en el trait, por ejemplo `#extractTraitFrom:`, y se repitió todo el proceso nuevamente.

4.5. Refinando métodos

Terminado este proceso, se empezó a trabajar más refinadamente a nivel método. Es así como se examinó cada uno de los métodos restantes para ver como podían ser refactorizados y así poder mover más implementaciones de la clase al trait. Luego de repetidamente descomponer métodos en métodos más pequeños para entonces mover algunos de estos al trait, se encontró que este proceso manual y repetitivo era un buen candidato a convertirse en un trait refactoring. Es así como se llegó al refactoring *Extract method to trait*.

Una vez creado este refactoring, el proceso manual antes usado se convirtió ahora en una tarea sencilla y automática. Sólo era necesario seleccionar que parte del método extraer, seleccionar el trait destino y darle un nuevo nombre al método extraído.

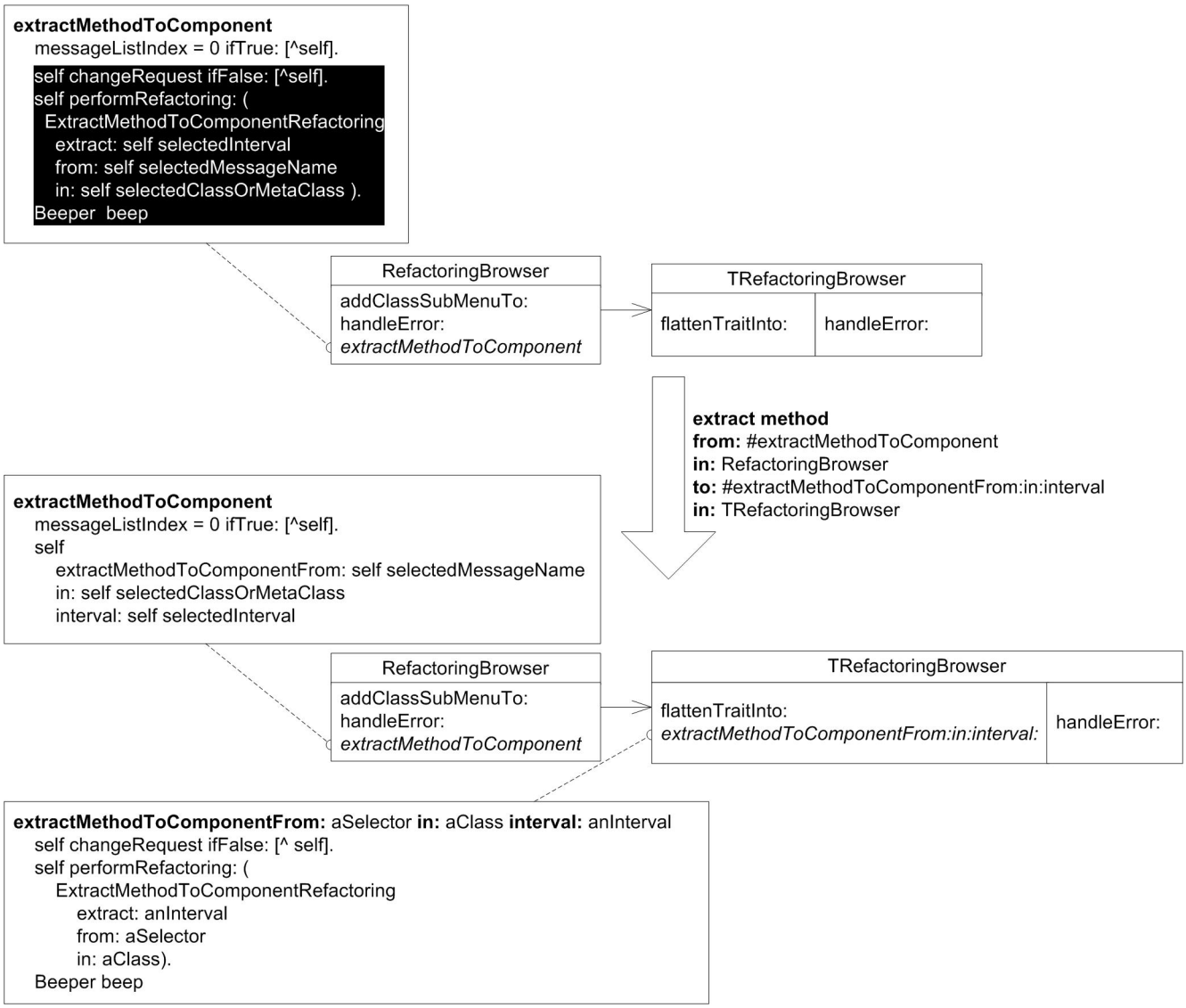

Figura 4.5: Extrayendo método en el trait `TRefactoringBrowser`

4.6. Utilizando el trait

Una vez que el trait `TRefactoringBrowser` parecía estar completo, es decir todos los métodos acerca de refactorings eran provistos por el trait y no por la clase misma, sólo fue necesario crear una nueva clase que usase dicho trait, implementar algunos requerimientos, conectar algunos métodos (glue code), y por último conectar esta nueva clase al Omnibrowser. De esta forma se terminó la refactorización que hizo posible ejecutar también desde el Omnibrowser los refactorings disponibles en el refactoring browser.

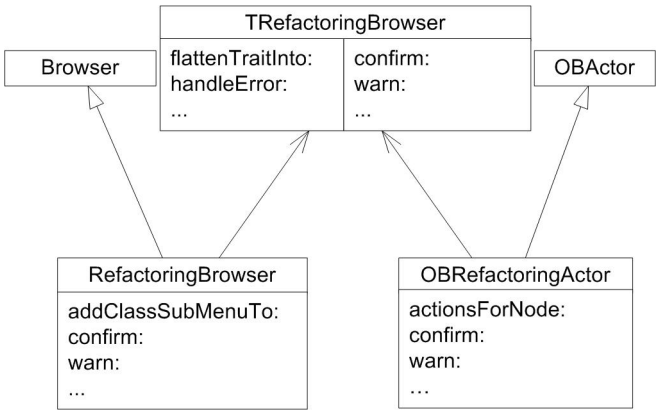

Figura 4.6: Conectando el OmniBrowser al motor de refactorings usando el trait `TRefactoringBrowser`

4.7. Resultados

El objetivo de esta refactorización fue obtener resultados acerca de los trait refactorings, y no acerca de la refactorización en sí misma. Si bien el resultado final de este proceso de refactorización no fue el ideal, es decir el diseño final no resultó más claro y sencillo que el que había en principio, con ella se pudo ajustar el uso de los trait refactorings existentes. También se pudo encontrar una serie de pasos mecánicos y reiterativos que finalmente pudieron ser conceptualizados como nuevos trait refactorings.

Los refactorings utilizados fueron:

- *Extract trait*. Ejecutado sólo una vez al principio de todo el proceso.
- *Move method to trait*. Ejecutado reiteradas veces para mover métodos desde la clase al trait en construcción. Este refactoring fue ejecutado, aproximadamente, un %70 en relación a todas las otras operaciones de refactorings.
- *Create missing requirements*. Ejecutado implicitamente como último paso del refactoring *Move method to trait* (no se ejecutó en forma explícita). En principio no existía este refactoring como tal. Fue durante este proceso de refactorización que se encontró que la última operación realizada por *Move method to trait*, podía ser implementadá como un refactoring por sí mismo. Este refactoring fue de mucha ayuda, ya que creó los requerimientos que fueron la guía para elegir los futuros métodos a ser movidos. Además, el disponer de *Create missing requirements* como refactoring por si mismo, también nos permitió agregarlo al refactoring browser para que el usuario pudiese ejecutarlo en cualquier momento sobre un trait dado.
- *Extract method to trait*. Ejecutado reiteradas veces para descomponer métodos que no podían, o no correspondían, ser movidos al trait, pero que contenían fragmentos que sí podían ser movidos al mismo. Este refactoring fue ejecutado aproximadamente un %25 en relación a todas las otras operaciones de refactorings que fueron ejecutadas.

4.8. Conclusiones

Las precondiciones de los refactorings limitan muchas veces su flexibilidad. Si la precondición es sencilla, se puede dejar en manos del programador la decisión acerca de que hacer cuando la precondición no se cumple. Este tipo de cambios hacen más flexibles a los refactorings.

Un ejemplo de esto es el refactoring *Move method to trait*, donde cierta precondición inicialmente consistía en verificar que el trait no definiera el método a ser movido.

```
MoveMethodFromClassToTraitRefactoring >> preconditions
  ^ ( ( RBCondition definesSelector: selector in: toTrait ) not )
  & ... other preconditions
```

Esta precondición es sencilla de entender por el programador, quién conoce cuales son las consecuencias de mover un método nuevo al trait sobreescribiendo el que éste ya tenía.

Inclusive, aún cuando el trait ya defina tal método, puede que el programador quiera mover una nueva implementación de éste al trait, ya que por ejemplo la nueva versión es más eficiente, corrige algún defecto o hace lo mismo pero es más comprensible.

Es decir, en estos casos, aún cuando la precondición programática no se cumpla, el programador puede asistir al refactoring indicándole que es seguro seguir adelante y refactorizar.

Volviendo al ejemplo de *Move method to trait*, la precondición fue cambiada para que el programador pueda asistir al refactoring indicándole cuando es seguro seguir adelante con la refactorización, y cuando no.

```
MoveMethodFromClassToTraitRefactoring >> preconditions
  ^ ( RBCondition withBlock: [
      (self traitDefinesDifferentMethod) ifTrue: [
        self refactoringWarning:
          'Trait already has a body for #', selector,
          '. Do you want to override it?' ].
        true ] )
  & ... other preconditions
```

El fragmento de código anterior, muestra que la precondición no fue cambiada. Sólo se le agregó soporte para que el programador pueda asistir al refactoring en ciertas circunstancias.

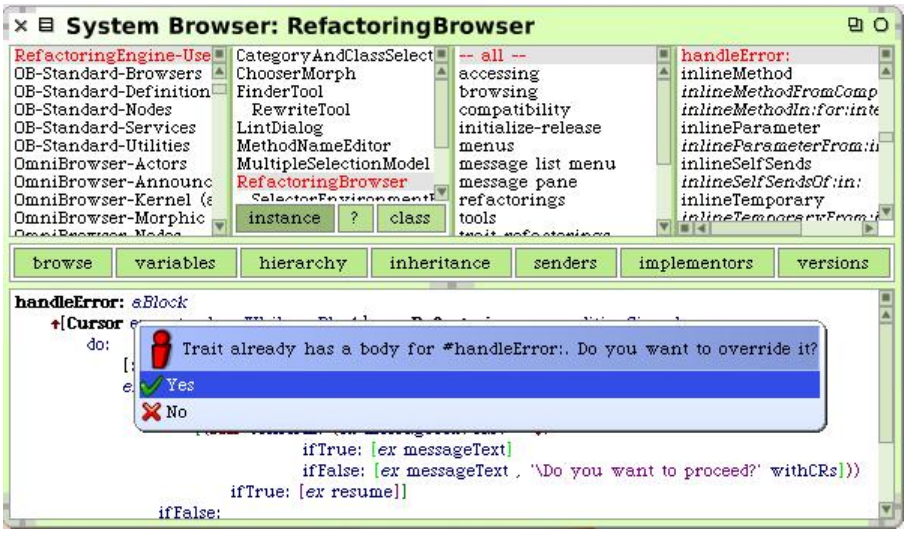

Figura 4.7: Refactoring requiriendo decisión del usuario

En la imágen anterior, si el usuario respondiese que sí a la pregunta, la precondición se cumple y el refactoring se ejecuta, en caso contrario la precondición no se cumple y en consecuencia el refactoring se cancela.

Lamentablemente las precondiciones no siempre son sencillas y si el programador no comprende bien las consecuencias de no cumplir con cierta precondición, puede ocurrir que se deje lugar para que cometa errores inadvertidamente.

5. Caso 2. Magnitude

`Magnitude` es una clase abstracta que provee el protocolo e implementación necesaria para objetos que necesiten ser comparados a lo largo de una dimensión lineal [GR83].

El protocolo especificado por esta clase refleja el hecho de que cualquier instancia de cualquier subclase de `Magnitude`, puede ser comparado con cualquier otra instancia de la misma clase usando operadores relacionales tales como `#=`, `#<`, `#>`, `#<=`, `#>=`. Subclases de `Magnitude` incluyen por ejemplo `Date`, `Time` y `Number`.

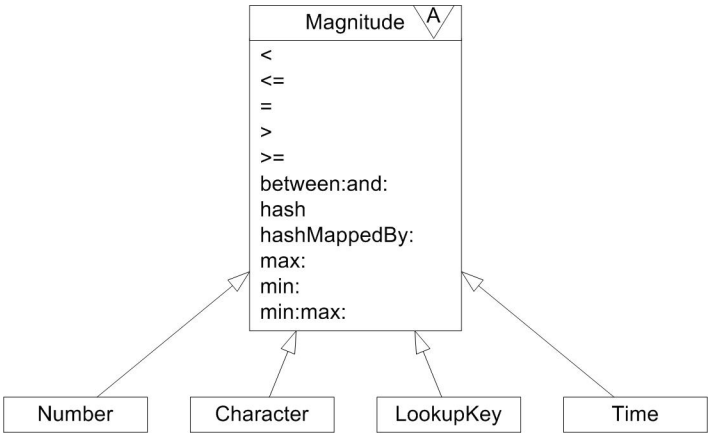

Figura 5.1: Magnitude y subclases

5.1. Motivación

Supongamos, un programador se encuentra trabajando dentro del framework *MVC (Model-View-Controller)* [KP88], y tiene una clase del dominio del modelo que necesita el protocolo básico de comparación que provee `Magnitude`. Ante esta situación, el programador tiene dos alternativas: o bien decide heredar su clase de `Magnitude` y copiar los métodos y variables de estado de `Model`, o bien decide heredar su clase de `Model` y copiar los métodos de `Magnitude`. En ambas alternativas el programador se ve forzado a duplicar código.

Éstas son situaciones donde el concepto de traits se convierte en una buena alternativa que no incluye la duplicación de código. Ésto nos motivo a refactorizar `Magnitude` para obtener su equivalente en trait.

5.2. Convirtiendo Magnitude en un trait

Se comenzó creando un nuevo refactoring, *Class as trait*.

La idea al crear este refactoring era la siguiente: convertir una clase en trait y hacer que todas sus subclases usen el trait en vez de heredar el comportamiento de dicha clase.

Una vez creado el refactoring se lo aplicó a `Magnitude`. En este caso, se esperaba crear un trait llamado `TMagnitude` y hacer que todas las subclases de `Magnitude` (`Number`, `Character`, `LookupKey`, `DateAndTime`, `Duration`, `Timespan`, `Time`, `MessageTally`, etc) usen dicho trait y dejen de subclasificar de `Magnitude`, para pasar a ser subclases de `Object` (superclase de `Magnitude`). Por último, luego de aplicado el refactoring se borraría la clase `Magnitude`.

Dado que la clase `Number` no pudo ser recompilada (ésto era necesario para cambiar la superclase de `Number` de `Magnitude` a `Object`), el refactoring no funcionó en su totalidad. El entorno no permite cambios como éste, en ésta y otras clases (conocidas como *kernel classes*), ya que de la integridad de éstas depende el sistema entero.

5.3. Reemplazando la herencia en una subclase

Dado que este método falló se pensó en extraer un trait inicial y luego empezar a usar el trait en tantas subclases de `Magnitude` como se pudiese. Es así como se vió la necesidad de crear un nuevo refactoring llamado *Replace inheritance with composition*. La idea de este refactoring era parecida a la de *Class as trait* pero con ligeras variantes que hicieron que éste funcionase en el caso donde el anterior no. El funcionamiento de este refactoring sería el siguiente: crear un trait a partir de la superclase de una clase dada, este trait contendría todos los métodos de dicha superclase; y luego, la clase que antes heredaba de ésta ahora heredaría de la super-superclase, obteniendo del trait creado toda la funcionalidad que antes heredaba. Si el trait a ser creado ya existiese, se lo usaría en vez de crear uno nuevo.

Habiendo implementado este refactoring, se lo aplicó a una subclase de `Magnitude`, en este caso `Character`, y se obtuvo un trait llamado `TMagnitude`.

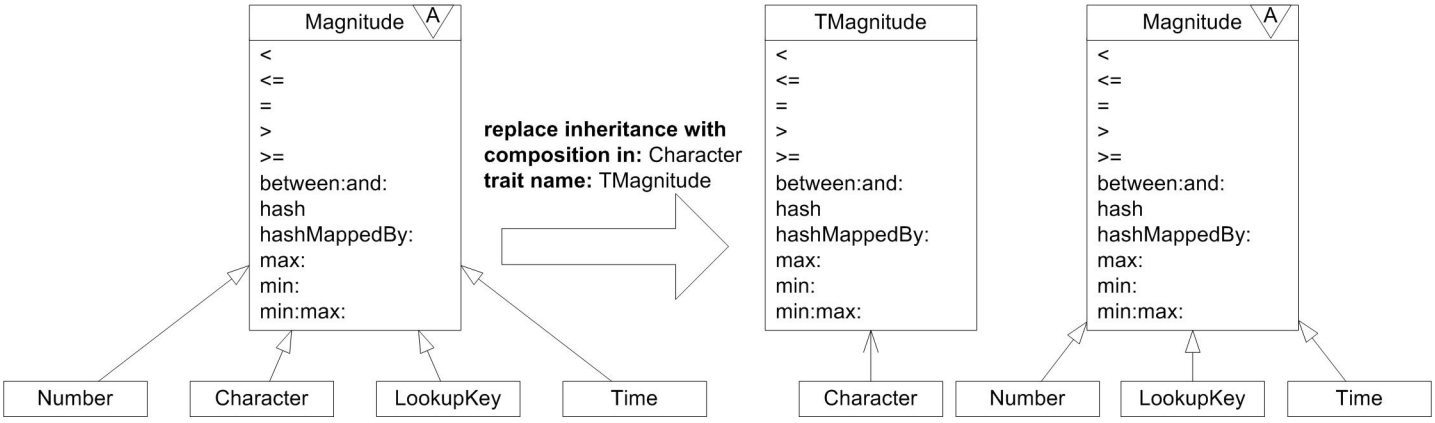

Figura 5.2: Reemplazando superclase de `Character` por trait `TMagnitude`

5.4. Reemplazando la herencia en cada subclase

Una vez aplicado este refactoring se tomó otra subclase de `Magnitude`, esta vez `Time`, y se volvió a aplicar el refactoring. Este segundo proceso de refactoring no creó un nuevo trait, sólo hizo que `Time` use `TMagnitude` y ahora herede de `Object` (superclase de `Magnitude`).

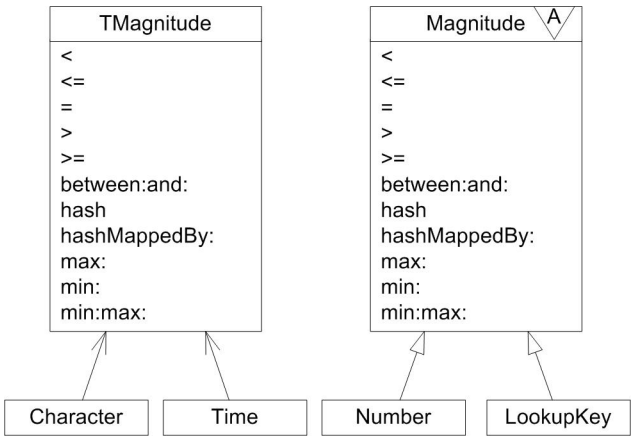

Figura 5.3: Reemplazando la superclase `Magnitude` por el trait `TMagnitude` en todas las subclases donde fue posible

Este proceso se repitió en todas las subclases de `Magnitude`, excepto en `Number` y `LookupKey`. Estas fueron las únicas clases en las que el entorno no permitió el refactoring debido a la necesidad de una recompilación.

5.5. Removiendo duplicación entre Magnitude y TMagnitude

Por último, para remover duplicación de código, se aplicó *Extract trait* a `Magnitude`. Para esto fue necesario transformar, en este refactoring, una precondición a decisión del programador: *Extract trait* no permitía extraer un trait si este ya existía, ahora en cambio, se permite que se use el existente en vez de crear uno nuevo si el programador así lo desea.

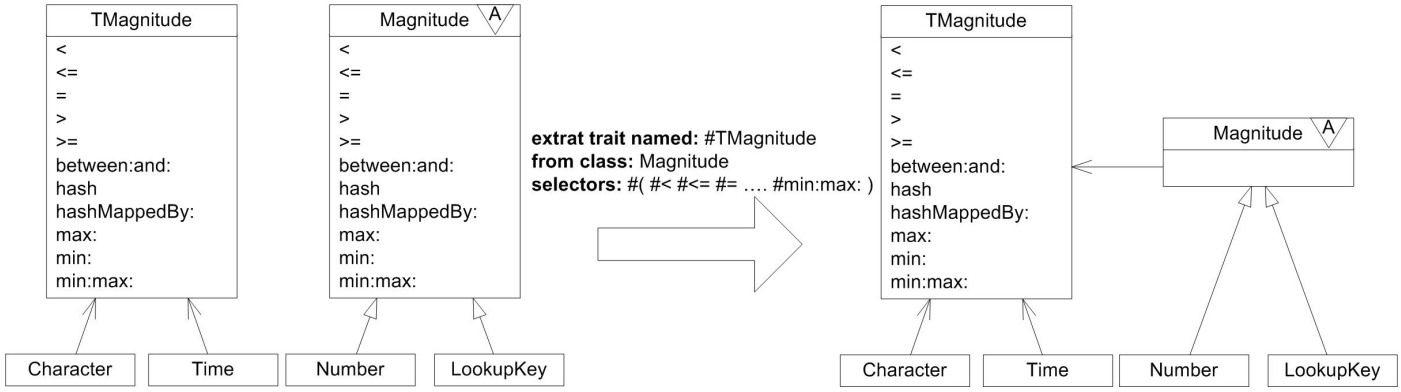

Figura 5.4: Extrayendo el trait `TMagnitude` de `Magnitude` para eliminar métodos duplicados

5.6. Buscando más usuarios para TMagnitude

Una vez aplicado este refactoring, se buscó alguna otra clase que pudiese ser usuaria de `TMagnitude`.

Esta búsqueda se realizó ejecutando el siguiente fragmento de código en un workspace:

```
| classes protocol |
protocol := TMagnitude selectors copyWithoutAll: #( #between:and: ).
classes := Smalltalk allClasses select: [:aClass | aClass selectors includesAllOf: protocol ].
classes := classes reject: [ :aClass| aClass inheritsFrom: Magnitude ].
classes inspect
```

Usando este método se encontró que la clase `Point` contenía todo el protocolo de `TMagnitude` a excepción del método `#between:and:`. Es aquí donde se comprobó que copiar métodos para incluir funcionalidad que no pudo ser heredada, no es una buena alternativa, ya que en el proceso de copiado puede que nos olvidemos de algún método ó, en el caso que no olvidemos nada, puede que nuestra clase no evolucione a la par de la clase de donde copiamos los mismos. Por ejemplo, en este caso pudo haber sucedido que `#between:and:` haya sido olvidado en el proceso de copiado, que éste haya sido agregado a `Magnitude` en algún momento posterior a la creación de `Point` ó que exista algún otro motivo para no incluir `#between:and:` en `Point`.

Aquí nuevamente se puede comprobar los beneficios de implementar `Magnitude` como un trait. Ya que si `Magnitude` hubiese sido un trait, `Point` podría haber incluido su funcionalidad sin duplicar código. Si más tarde se agregase `#between:and:` al trait, `Point` se vería inmediatamente beneficiada de dicha inclusión. Además, si existiese algún motivo para que `Point` no incluya `#between:and:`, sólo bastaría con excluirlo en `Point`, quedando así explícita la decisión del programador.

5.7. Aplicando TMagnitude en Point

Retomando el proceso de refactorización, se procedió a extraer de `Point` el trait `TMagnitude`. El protocolo de `Point` constaba de 91 métodos, de los cuales sólo 11 de ellos pertenecían al protocolo de `TMagnitude`. Contando con esta información, se aplicó el refactoring *Extract trait* a `Point`, seleccionando sólo estos 11 métodos e ingresando el nombre de nuestro ya existente trait `TMagnitude`. El resultado fue una clase `Point` usando el ya existente trait `TMagnitude` y con menos métodos implementados localmente por la clase (ya que el refactoring removió de `Point` los métodos de igual implementación para dejar sólo los de `TMagnitude`).

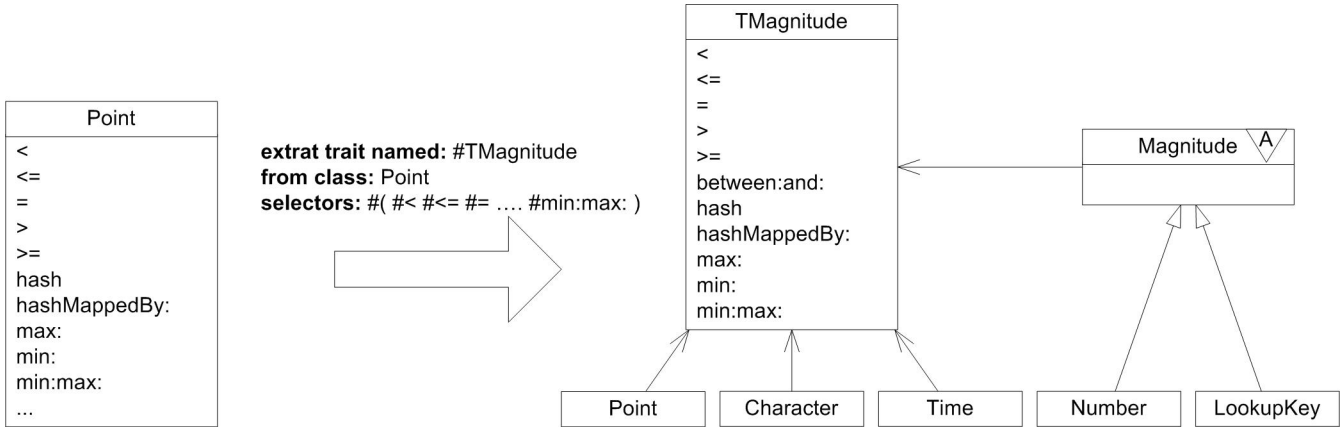

Figura 5.5: Extrayendo el trait `TMagnitude` de `Point` para eliminar métodos duplicados

Por último y siguiendo la premisa de que un refactoring no debe alterar el comportamiento de un sistema, se excluyó el método `#between:and:` en `Point` ya que instancias de esta clase

no sabían responder a este mensaje antes del refactoring. Está claro que si supiésemos que no existen motivos para excluir `#between:and:` de `Point`, no haríamos dicha exclusión.

Con este último paso se concluyó el proceso de refactoring de `Magnitude`.

5.8. Resultados

A medida que más trait refactorings se iban implementando, fueron apareciendo paralelamente trait refactorings más pequeños. Ésto permitió que la construcción de nuevos trait refactorings futuros fuese más fácil. Por ejemplo, cuando se creó *Replace inheritance with composition*, se encontró que creando un nuevo refactoring se podía remover mucho del código duplicado entre éste y el refactoring *Move method to trait*. Es así es como surgió *Add method to trait*, refactoring sencillo y de corta funcionalidad pero que ayudó a remover código duplicado de los dos refactorings anteriormente mencionados.

Los refactorings más usados fueron:

- *Replace inheritance with composition*. Ejecutado una vez por cada subclase de `Magnitude`, excepto en `Number` y `LookUpKey`.
- *Extract trait from class*. Ejecutado dos veces. La primera fue para remover duplicación de código entre el trait `TMagnitude` y la clase `Magnitude` (que estábamos obligados a mantener ya que `Number` debía seguir heredando de `Magnitude`). La segunda fue para remover código de `Point` y tomarlo de `TMagnitude`.

5.9. Conclusiones

Refactorings que aplican a un conjunto de clases de un tamaño considerable, pueden no ser del todo útiles. Para que dichos refactorings puedan ser aplicables, todo tiene que cumplir una serie de precondiciones. En la práctica hacer que todo un conjunto de clases cumplan con estas precondiciones, es sumamente difícil, sobre todo si el conjunto es grande.

Además tales refactorings pueden no aplicar los cambios en la misma forma en que lo habíamos planificado inicialmente. Si este es el caso, después habrá que deshacer todo el refactoring, para luego realizarlo en forma manual (lo cual puede ser costoso y propenso a otros errores), o habrá que encontrar cuales fueron las clases afectadas por el refactoring, para luego verificar una a una si el cambio aplicó como esperábamos y entonces reparar las fallas encontradas. La capacidad de poder ver todos los cambios realizados, o a realizar, por un refactoring y luego poder deshacer, o excluir, cambios particulares sería de mucha ayuda en casos como estos. Actualmente el motor de refactorings no tiene esta capacidad.

De esto surge la conclusión de que es mejor tener refactorings que aplican cambios más pequeños. En este caso el programador tiene mejor control de qué cambios se aplican y cuáles no, y puede paso a paso ir retocando los resultados de los refactorings que se van aplicando, en caso de que alguno de ellos no produzca el resultado que se esperaba.

Por otro lado, los operadores de exclusión de los traits, nos permitieron flexibilizar los refactorings. En vez de tener precondiciones para evitar la compilación de métodos en caso de conflictos, se optó por remover tales precondiciones y manejar los conflictos haciendo uso de estos operadores de exclusión. Sacar estas precondiciones dejaron los refactoring en un estado mucho más simple y flexible a la hora de su uso.

Por último, utilizar un refactoring con propósitos para los que éste no fue creado, no es conveniente. Aún cuando el resultado que fuésemos a obtener es el que esperamos. Como por

ejemplo el caso del refactoring *Extract trait*, que fue aplicado en la clase `Magnitude` y `Point` con el fin de componerlos con el trait ya existente `TMagnitude` y al mismo tiempo remover la duplicación de métodos. La idea detrás de este refactoring es extraer un trait, no usar uno existente y manejar la duplicación de métodos. La utilización de este refactoring con estos fines es impracticable para cualquiera que no conozca los detalles del funcionamiento del mismo.

De esta inconveniencia, surgió la idea de un nuevo refactoring llamado *Use trait*, que fue implementado más tarde (ver capitulo 6). Con este refactoring pudimos obtener los mismo resultados que del proceso antes mencionado, aunque en esta ocasión, disponiendo de un refactoring con una funcionalidad mucho mas reveladora, clara y definida para el mismo propósito.

6. Caso 3. Aconcagua

Aconcagua es una librería que reifica el concepto de medidas, unidades y su aritmética [WPR05]. Este modelo representa las medidas como como objetos que encapsulan un número y su unidad.

La siguiente figura muestra las principales clases que componen a Aconcagua.

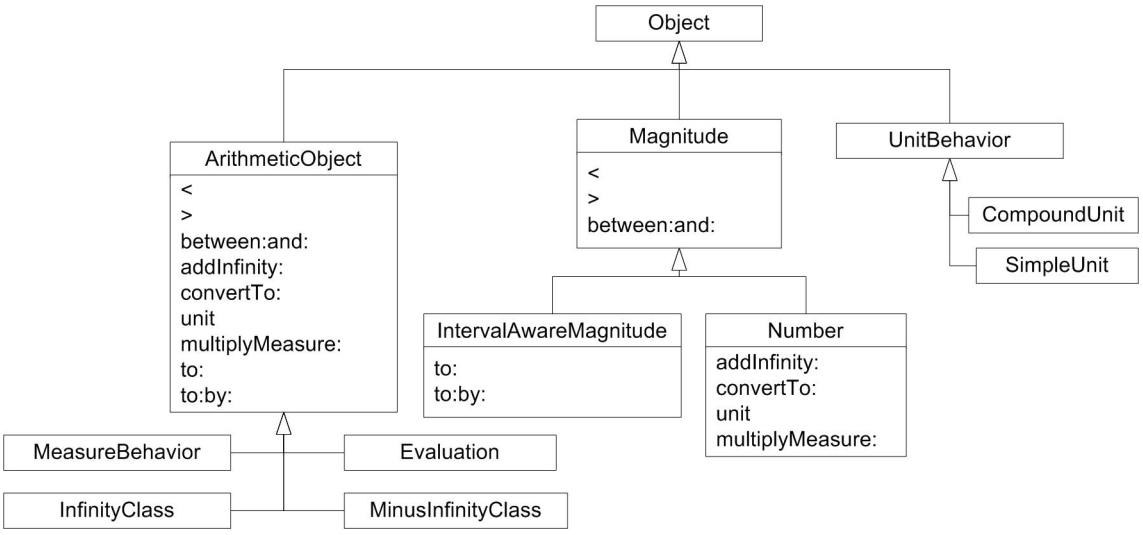

Figura 6.1: Aconcagua

6.1. Motivación

En Aconcagua se fueron encontrando diferentes métodos duplicados a lo largo de diferentes clases no relacionadas por la herencia. Por ejemplo:

- `ArimethicObject` implementaba el protocolo completo de `Magnitude` ya que no heredaba de ésta.
- `Number` se encontraba extendida por un conjunto de métodos que también se encontraban implementados en `ArimethicObject` o subclases de ésta, como por ejemplo `#addInfinity:`, `#convertTo:`, `#unit`, `#multiplyMeasure:`, etc. Todo este protocolo es un buen candidato a ser un trait por si mismo que podría ser usado tanto en subclases de `ArimethicObject` (o en la misma `ArimethicObject`) como en `Number`.
- `IntervalAwareMagnitude` es subclase de `Magnitude`, que como se vió en la sección previa, es más conveniente como trait que como clase.
- Subclases de `IntervalAwareMagnitude` y `ArimethicObject` comparten implementación en el protocolo relacionado a la creación de intervalos: mensajes `#to:` y `#to:by:`. Todo este protocolo es un buen candidato a ser un trait por sí mismo.

Toda esta duplicación en métodos esparcidos en diferentes jerarquías nos motivó a aplicar traits en ellos aprovechando que éstos permiten un nuevo estilo de programación en el cual, los traits se convierten en la unidad básica de reuso de código [DB03] [DB04].

6.2. Comenzando por TMagnitude

El proceso comenzó por aplicar el trait `TMagnitude` a la jerarquía de clases de Aconcagua.

Fue en este momento donde se hizo evidente la necesidad de contar con el refactoring que se había previsto en la sección anterior: *Use trait*.

Una vez creado el refactoring, se procedió a aplicarlo en la clase `ArimethicObject`. El resultado fue una nueva versión de esta clase que proveía la misma cantidad de métodos, todos ellos con la misma implementación, pero una menor cantidad de métodos definidos localmente por la clase.

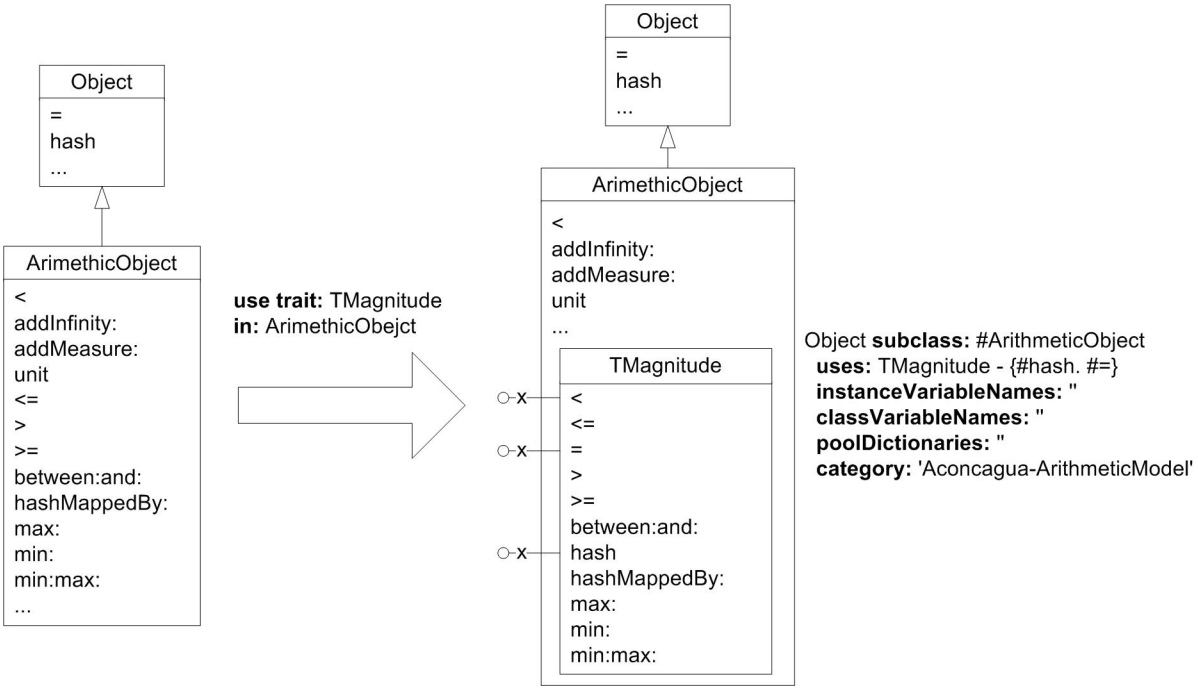

Figura 6.2: Usando trait `TMagnitude` para remover implementaciones duplicadas

Habiendo completado la tarea de remover todo código duplicado de `ArimethicObject` relacionado al protocolo de `TMagnitude`, el paso siguiente consistió en cumplir con el mismo objetivo pero esta vez en `IntervalAwareMagnitude`.

Dado que `IntervalAwareMagnitude` era subclase de `Magnitude` el refactoring aplicado en este caso fue *Replace inheritance with composition*. Este refactoring nos requirió el nombre del trait que vendría a reemplazar a `Magnitude`. El nombre ingresado fue el del ya existente trait `TMagnitude`. Luego que el refactoring reconociera al trait como ya existente, tuvimos que confirmar la pregunta de si queríamos usar el ya existente trait. Confirmando la pregunta, el refactoring paso a reemplazar la superclase `Magnitude` por el trait `TMagnitude`.

6.3. Buscando nuevos traits

Habiendo aplicado `TMagnitude` en Aconcagua, se pasó a revisar con más detalle los métodos de cada clase de esta librería. Así es como se encontró que los mensajes `#between:andNotInclusive:`, `#notInclusiveBetween:and:` y `#notInclusiveBetween:andNotInclusive:` estaban implementados en diferentes lugares de la jerarquía: `Measure`, `Evaluation` e `IntervalAwareMagnitude`.

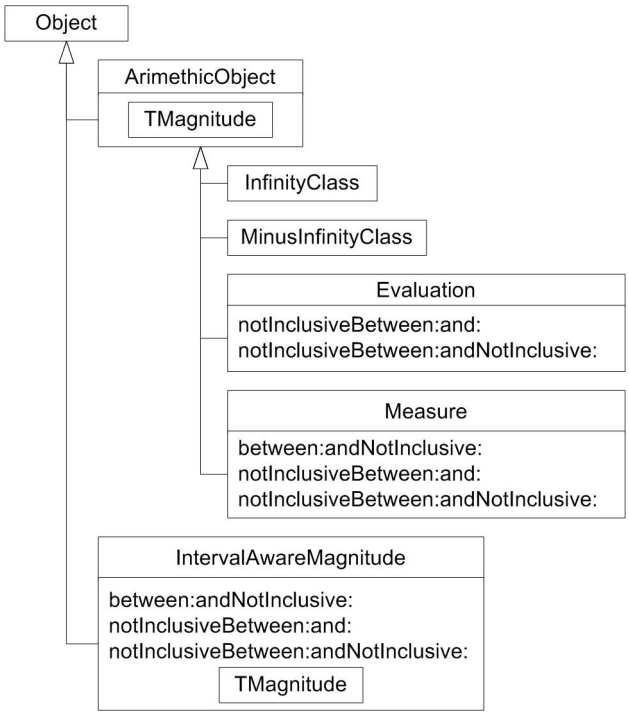

Figura 6.3: Métodos implementados en diferentes lugares de la jerarquía de clases de Aconcagua

Si bien todas estas clases compartían dicho protocolo, no existían iguales implementaciones. Por ejemplo la implementación de `#notInclusiveBetween:and:` era diferente en cada clase.

```
Evaluation >> notInclusiveBetween: min and: max
  | val |
  val := self value.
  ^val > min and: [val <= max]

Measure >> notInclusiveBetween: min and: max
  ^self > min and: [self <= max]

IntervalAwareMagnitude >> notInclusiveBetween: min and: max
  ^min < self and: [self <= max]
```

Si bien `IntervalAwareMagnitude` y `Measure` parecían tener la misma implementación, no la tenían. Los operadores `#<` y `#>` no son operadores primitivos como en cualquier otro lenguaje de programación, éstos son mensajes regulares. En el caso de `Measure` se envía el mensaje `#>` a sí mismo pasándose el objeto `min` como parámetro. En cambio, en `IntervalAwareMagnitude` se envía el mensaje `#<` al objeto `min` pasándose a sí mismo como parámetro.

Buscando la forma de hacer que ambas implementaciones sean la misma, se encontró que la implementación de `#>` en `IntervalAwareMagnitude` era:

```
IntervalAwareMagnitude >> >
  ^ min < self
```

y entonces pudimos refactorizar `IntervalAwareMagnitude≫notInclusiveBetween:and:` para reemplazar su definición por

```
IntervalAwareMagnitude >> notInclusiveBetween: min and: max
  ^self > min and: [self <= max]
```

Así obtuvimos dos implementaciones iguales que podían ser refactorizadas en un trait. La única desventaja de esto era una posible penalización en performance, ya que estábamos introduciendo el envío de un mensaje más en la implementación de `#notInclusiveBetween:and:`.

Por último, en `IntervalAwareMagnitude`, el mismo refactoring pudo ser aplicado en `#notInclusiveBetween:andNotInclusive:` aunque no así en `#between:andNotInclusive:`.

Por otro lado el hecho de que, por ejemplo, clases como `InfinityClass` y otras subclases de `ArimethicObject` no implementasen este protocolo no parecía tener sentido, ya que éste podía ser implementado en cualquier clase que tuviese incorporado el protocolo de `TMagnitude`. Es así como se decidió aplicar el refactoring *Push up method* en los tres métodos de `Measure` antes de aplicar traits.

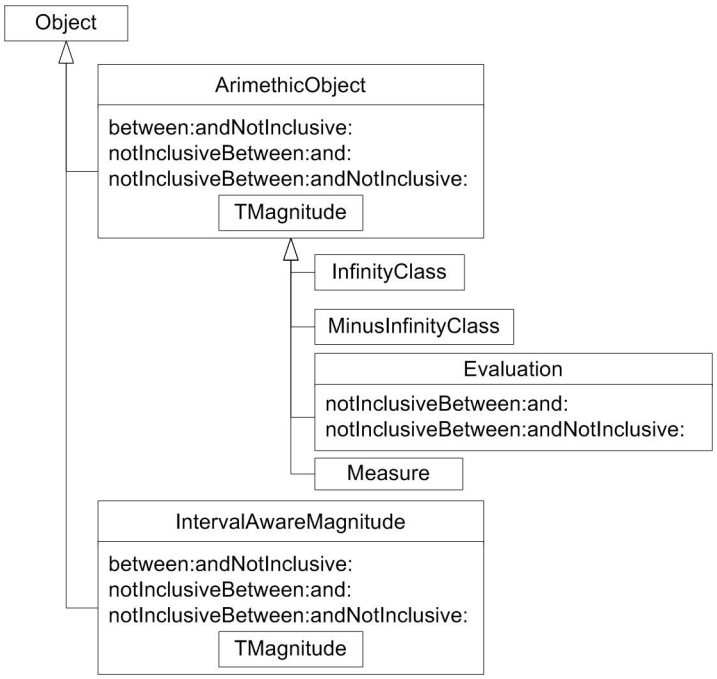

Figura 6.4: Refactorizando métodos antes de aplicar traits a la jerarquía de clases de Aconcagua

6.4. Extrayendo y reutilizando un nuevo trait

Habiendo realizado todos estos refactorings, nos encontrábamos en una mejor posición para extraer un trait que contenga estos tres métodos.

Se aplicó el refactoring *Extract trait* y obtuvimos el trait llamado `TAconcaguaMagnitude`.

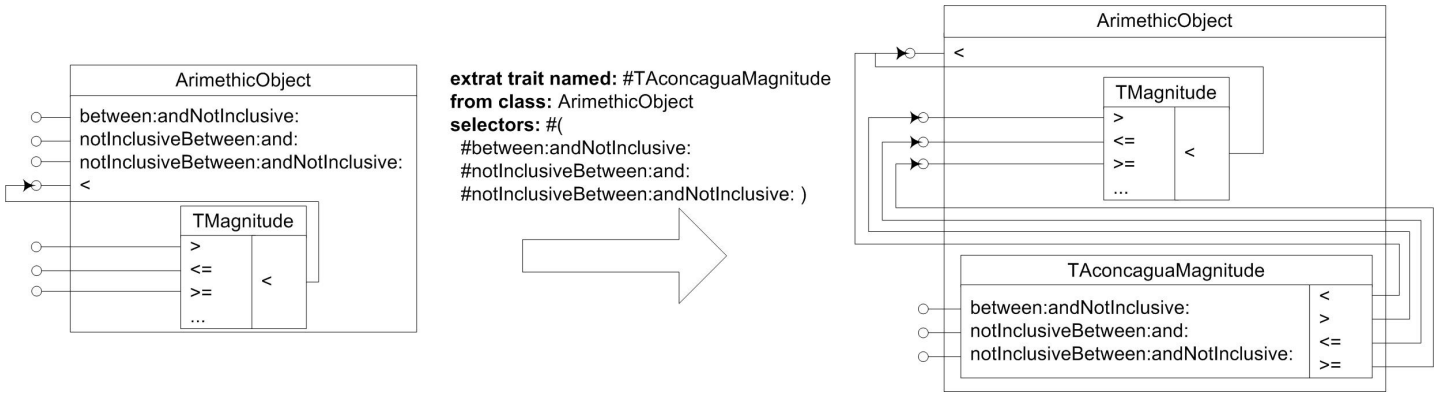

Figura 6.5: Extrayendo trait `TAconcaguaMagnitude`

Disponiendo de este trait, el paso siguiente fue aplicar el refactoring *Use trait* en la clase `IntervalAwareMagnitude`.

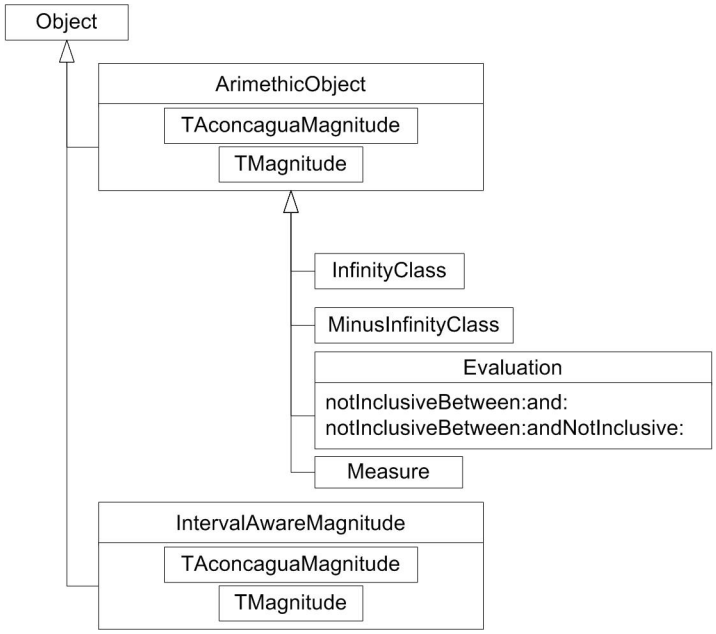

Figura 6.6: Usando `TAconcaguaMagnitude` en la jerarquía de clases de Aconcagua

6.5. Removiendo dependencias

Observando la relación entre los traits `TMagnitude` y `TAconcaguaMagnitude` se encontró que existía una fuerte dependencia del trait `TAconcaguaMagnitude` hacia el trait `TMagnitude`, ya que el primero tenía 3 requerimientos satisfechos por el segundo.

El problema con este tipo de dependencias entre traits es que hace a `TAconcaguaMagnitude` más difícil de reusar en otros contextos, ya que no podría ser usado sólo, sino que dependería de que `TMagnitude` también se encuentre allí.

Es así como surgió la idea del refactoring *Replace dependency with composition*.

La motivación de crear este refactoring no sólo estuvo basada en este caso. La sección 4.4

Improvements at the Trait-Level de [Lien04] describe un escenario en que el mismo refactoring fue aplicado, aunque en forma manual. En ese mismo trabajo, también se remarca las desventajas de tener traits cuyos requerimientos son satisfechos por otros traits en la misma composición, en vez de ser satisfechos por la clase usuaria.

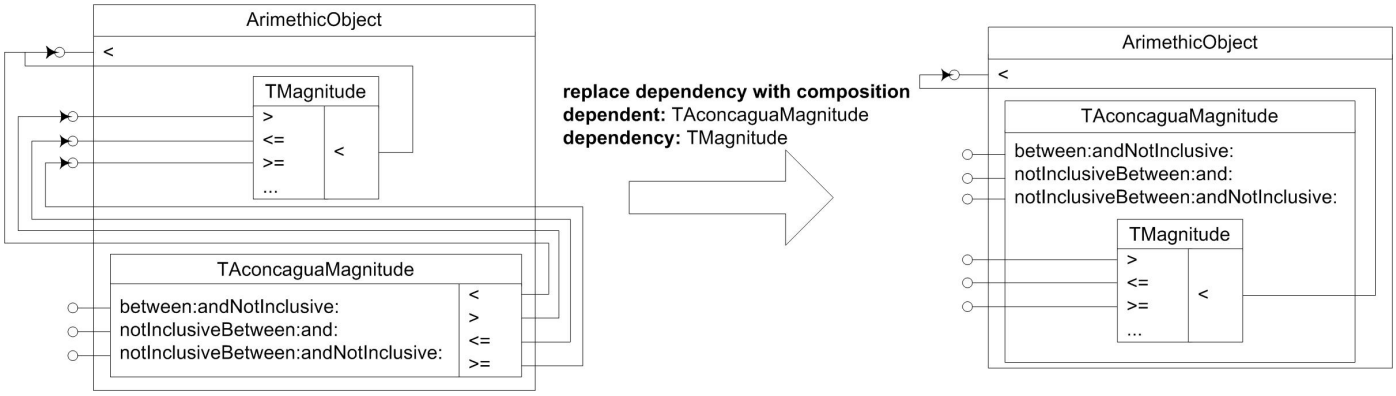

Figura 6.7: Removiendo dependencia del trait `TAconcaguaMagnitude` al trait `TMagnitude`

Además, la idea de aplicar tal refactoring en esta situación en particular, estuvo reforzada por el hecho de que `TAconcaguaMagnitude` parecía ser mas bien una extensión a `TMagnitude` y no un trait separado de éste.

Una vez aplicado el refactoring *Replace dependency with composition* nos encontrábamos ante un diseño de las clases de Aconcagua con menos código duplicado y con un trait `TAconcaguaMagnitude` con mejores expectativas de usos futuros.

6.6. Buscando y extrayendo mas traits

Para concluir con la refactorización de Aconcagua sólo nos restaba hacer dos refactorizaciones más: remover la duplicación de código relacionada a la creación de intervalos y remover la duplicación de código relacionada a extensiones aritméticas en `Number` y que se encontraban también en `ArimethicObject` y sus subclases.

Por lo tanto el siguiente paso consistió en la aplicación de los refactorings *Extrac trait* en `IntervalAwareMagnitude`, seleccionando los métodos `#to:` y `#to:by:` para obtener un trait llamado `TIntervalAware`, y luego aplicar el refactoring *Use trait* en `ArimethicObject`. La última refactorización no fue tan directa ya que los métodos duplicados no se encontraban definidos directamente en `ArimethicObject` sino en subclases de ésta, por lo que primero fue necesario aplicar el refactoring *Push up method* antes de aplicar *Use trait* para remover el código duplicado.

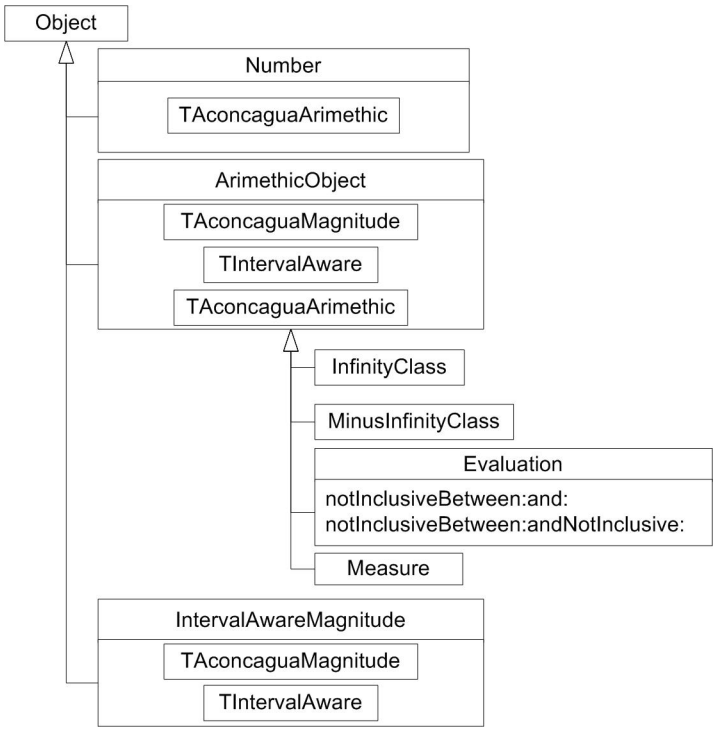

Figura 6.8: Jerarquía de clases de Aconcagua refactorizada con `TIntervalAware`

En este punto sólo restaba refactorizar las extensiones de `Number` en un trait que pudiese ser reutilizado en clases de Aconcagua.

Dado que esta refactorización involucraba una cantidad considerable de métodos (las extensiones a `Number` consistían de unos 51 métodos), se procedió a la refactorización en una forma mas fina y granular, muy parecida a la refactorización realizada con la clase `RefactoringBrowser` en el capítulo 4. Es decir, se comenzó por buscar un método con implementación duplicada para luego extraer de `Number` un trait llamado `TAconcaguaArimethic` que contuviese dicho método. Luego el trait fue aplicado también en `ArimethicObject`. Habiendo aplicado el trait en ambas clases, se procedió a aplicar el refactoring *Move method to trait* para mover de `Number` a `TAconcaguaArimethic` cada método que fuésemos encontrando duplicado.

6.7. Resultados

Los refactorings más usados fueron:

- *Use trait*. Utilizado 4 veces. Se lo aplicó a `ArimethicObject` para componerla con los traits `TMagnitude`, `TIntervalAware` y `TAconcaguaArimethic`. También se lo usó para componer `IntervalAwareMagnitude` con el trait `TAconcaguaMagnitude`.
- *Replace inheritance with composition*. Utilizado una sola vez en `IntervalAwareMagnitude` para romper la herencia con `Magnitude` y componerla con `TMagnitude`.
- *Extract trait*. Utilizado 3 veces. En `ArimethicObject` para extraer `TAconcaguaMagnitude`, en `IntervalAwareMagnitude` para extraer `TIntervalAware` y por último en `Number` para extraer un trait inicial de un método, `TAconcaguaArimethic`.

- *Replace dependency with composition.* Utilizado una sola vez para reemplazar la dependencia de `TAconcaguaMagnitude` a `TMagnitude`.
- *Move method to trait.* Utilizado 50 veces, una vez por cada método que se fue encontrando duplicado entre `Number` y `ArimethicObject`. Por cada uno de éstos, se tomo una implementacion, ya sea la de `Number` o la de `ArimethicObject`, y se la movió al trait `TAconcaguaArimethic`.

6.8. Conclusiones

Actualmente el mecanismo de extensión de clases de Squeak funciona mediante convenciones en el sistema de categorías de métodos. Por ejemplo, las extensiones de Aconcagua a `Number`, son todos los métodos de esta clase que pertenecen a categorías del estilo **Aconcagua-xxxx*, donde xxxx podría ser cualquier subcategoría como por ejemplo *comparing*.

El problema con este método es que el nombre de una categoría no tiene semántica. Las categorías son solamente convenciones que sirven para describir. Por lo tanto, darle semántica a las categorías para implementar extensiones, no es algo para lo que éstas fueron concebidas y podría traer problemas. Por ejemplo, cualquier herramienta para recategorizar métodos automáticamente debería tener en cuenta esta regla para no perder extensiones en el proceso.

Durante esta refactorización, las extensiones de Aconcagua a `Number` fueron extraídas como un trait. Si bien los traits no surgieron como una alternativa o mejora al mecanismo de extensión de clases, la idea de usarlos con este propósito resulta atractiva. Ésto resolvería el problema antes mencionado, ya que no se daría semántica a las categorías y cualquier convención y herramienta para categorización de métodos podría usarse con las extensiones también. Además, resulta más fácil refactorizar grupos de métodos pertenecientes a una entidad (ya sea trait o clase), que refactorizar un grupo de métodos relacionados por los nombres de las categorías a las que pertenecen.

Por último, durante esta refactorización se creó un sólo nuevo refactoring, *Replace dependency with composition.* La mayoría de los refactorings utilizados durante este proceso, ya habían surgido de los casos de usos anteriores. Este hecho nos sugiere que varios de los trait refactorings encontrados hasta aquí son sencillos y aplicables en forma genérica a una cantidad considerable de escenarios.

7. Conclusiones

Este trabajo presenta un conjunto de refactorings "traits aware".

Comenzamos modificando el motor de refactorings de Squeak para poder manipular traits en la misma forma en que éste venia manipulando otros conceptos mas familiares como las clases, los métodos y las variables de estado. Luego, comenzamos a refactorizar "towards traits" diferentes librerías y clases de Squeak. Fue durante estas refactorizaciones, que encontramos y luego implementamos cada uno de los traits refactorings presentados en este trabajo.

En los procesos de refactorización aplicados no siempre se buscó la mejor situación donde refactorizar con traits, sino que sólo se buscó alguna situación donde poder aplicarlos. Esto se hizo así con el sólo fin de poder refactorizar y obtener resultados, mejoras y conclusiones acerca de los trait refactorings propuestos.

En todos los casos, el diseño obtenido luego de la refactorización resultó con menos código duplicado. Aunque no siempre resultó en un diseño más claro o comprensible, tal vez perdiendo sentido la refactorización aplicada y los traits obtenidos. Es mas, en ciertos caso se pudo haber refactorizado mucho del código duplicado con simplemente agregar una clase abstracta a la cadena de herencias, en vez de hacerlo con traits.

De esto último concluimos que, a pesar que los traits se presentan como unidades primitivas de reuso, utilizarlos con este sólo propósito no es una buena idea. Así como, por ejemplo, tampoco es buena idea usar la herencia con el sólo propósito de reutilizar y olvidando que su objetivo principal es el de modelar relaciones entre clases del tipo *es un*.

Es muy probable, que al perder de vista el concepto fundamental de traits (rasgos o pequeñas unidades de comportamiento para construir clases) y al utilizarlos sólo como primitivas de reuso de código, lo único que se obtenga sean diseños confusos que tal vez hubiesen sido más claros si estuviesen hechos de maneras mas tradicionales (usando herencia de clases por ejemplo).

Por otro lado, a partir de la implementación y posterior uso de los trait refactorings concluimos que los traits favorecen más las refactorizaciones (ya sean manuales o no) en comparación con otros mecanismos como mixins y múltiple herencia.

Esto se debe a que traits, ofrece un mecanismos más fino de control (operadores de exclusión) a la hora de componer, y además este mecanismo de control está del lado del compositor (la clase que incluye al trait). Esta característica de los traits facilitó mucho la tarea de creación de los trait refactorings de manera tal que mantengan el comportamiento del sistema. Por ejemplo, es fácil agregar métodos en un trait sin perder de vista el alcance de los cambios, y teniendo total certeza acerca de si se está introduciendo efectos indeseables en otras clases.

Con mixins o múltiple herencia tal vez no sería igual de fácil implementar refactorings que preserven el comportamiento. Con éstos es más difícil controlar los cambios que provoca, por ejemplo, agregar un método a un mixin o a una clase base en un esquema de múltiple herencia. Tal cambio podría alterar el comportamiento de alguna otra clase inadvertidamente.

Luego, del capítulo 5 también concluimos que modelar `Magnitude` como trait brinda más beneficios que la actual implementacion como clase. Esto se debe a que, teniendo herencia simple, disponer de `Magnitude` como clase nos obliga a heredar de ésta, perdiendo toda posibilidad de heredar de otras clases más acordes al dominio de nuestro problema.

Por último, las creación de nuevos refactorings se hizo más fácil a medida que disponíamos de refactorings de mas "bajo nivel" como *Add method to trait*. Teniendo refactorings de más bajo nivel es más fácil construir composicionalmente nuevos refactorings más complejos.

8. Trabajo futuro

Las extensiones ó mejoras que consideramos pueden realizarse sobre este trabajo son:

- El `RefactoryTyper` es una herramienta que dada una variable en un fragmento de código, trata de adivinar la clase a la que pertenecerá el objeto referenciado por dicha variable. Esta herramienta, integrada con el motor de refactorings, facilita la tarea del programador al ejecutar refactorings como *Extract method to component*. En este caso, el `RefactoryTyper` sugiere al programador clases candidatas en las que el método extraído podría compilarse.

  Los trait refactorings presentados en este trabajo podrían verse beneficiados por herramientas como estas, aunque adaptadas para traits, facilitando así las refactorizaciones encaradas por el programador.

  Por ejemplo, al aplicar refactorings como *Use trait* resultaría conveniente para el programador que tal herramienta sugiriese una lista de posibles traits a componer en la clase, ó viceversa, es decir, sugiriese una lista de clases en las que un trait podría usarse. Tal herramienta podría derivar de la heurística presentada en la sección 5.6, donde ésta fue usada para encontrar que `Point` era un buen candidato en el cual usar el trait `TMagnitude`.

  Otro ejemplo sería una herramienta que indique los traits existentes que ya implementan los métodos que estamos extrayendo como nuevo trait. Esta herramienta podría ayudar al programador en su refactorización para así evitar la proliferación de traits similares.

- `SLint` es una herramienta que realiza un análisis estático del código de una jerarquía de clases para detectar código sospechoso, confuso e incluso errores de programación. Usualmente, el código sospechoso encontrado por esta herramienta, es código que necesita refactorizarce.

  Contar con "traits aware lints" es otra herramienta que sería de mucha ayuda. Es decir, una herramienta que indique que clases o jerarquías son candidatas a ser refactorizadas con traits dada la cantidad de código duplicado ó que detecte dependencias entre traits, por ejemplo.

- Demostrar formalmente que los trait refactorings aquí presentados son "behavior preserving". Es decir, encontrar cuales son las precondiciones mínimas necesarias para garantizar que en cualquier situación donde se pueda usar un trait refactoring el comportamiento del sistema se mantiene.

- Traits viene incluido en la imagen base de Squeak a partir de la versión 3.9. Se espera la comunidad comience a hacer uso de ellos dadas sus ventajas, sencillez y compatibilidad con sistemas que no soportan traits (debido a la propiedad de flattening).

  Pulir la implementación actual de trait refactorings y hacerla disponible para su uso en Squeak, facilitaría el uso de traits por aquella parte de la comunidad que quiere comenzar a usarlos pero que no lo hacen dada la falta de soporte para este nuevo concepto.

- Dado que el concepto de traits es relativamente nuevo, y existe poca experiencia usándolos, todavía no está claro cuando es conveniente reutilizar por medio de la herencia de clases o reutilizar por medio de la composición de traits.

Es decir, hace falta desarrollar patrones y normas de diseño basados en traits que, por ejemplo, nos den un indicio acerca de cuándo es conveniente implementar un grupo de métodos en una clase o cuándo hacerlo en un trait.

Seguramente, cuando estos patrones de diseños orientados a traits aparezcan, surgirá la necesidad de contar con nuevos trait refactorings que les den soporte y los refuercen.

- La mayoría de las refactorizaciones realizadas en este trabajo tratan de la refactorización de clases para introducir traits.

  Dada la falta de sistemas construidos con traits y librerías de traits, no se pudo evaluar en detalle los trait refactorings aplicados en grafos complejos de traits.

9. Bibliografía y referencias

[WFO92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD Thesis, University of Illinois, 1992.

[FBBOR99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. *Refactoring: Improving the design of existing code*. Addison Wesley, 2nd Edition, 1999.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. *A refactoring tool for Smalltalk*. Theory and Practice of Object Systems (TAPOS), 3(4):253-263, 1997.

[RefacWeb] Martin Fowler's refactoring web page. `http://www.refactoring.com/`.

[SSO02] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.

[TraitsWeb] Traits research page from the Institute of Computer Science and Applied Mathematics of the University of Bern. `http://www.iam.unibe.ch/~scg/Research/Traits/`.

[SDNB02] Nathanael Sch¨arli, St´ephane Ducasse, Oscar Nierstrasz, and Andrew Black. *Traits: Composable units of behavior*. In Proceedings ECOOP 2003, volume 2743 of LNCS, pages 248-274. Springer Verlag, July 2003.

[SqueakWeb] Smalltalk Squeak web page. `http://www.squeak.org/`.

[BSD02] Andrew Black, Nathanael Sch¨arli, and St´ephane Ducasse. *Applying traits to the Smalltalk collection classes*. In Proceedings OOPSLA 2003 (International Conference on Object-Oriented Programming Systems, Languages and Applications), volume 38, pages 47-64, October 2003.

[DB03] Andrew Black and Nathanael Sch¨arli. *Programming with Traits*. Technical Report No. CSE 03-012, Sep 2003.

[DB04] Andrew Black and Nathanael Sch¨arli. *Traits: Tools and Methodology*. Proceedings ICSE 2004, May 2004, pp. 676-686.

[MixWeb] Mixins definition at Cunningham & Cunningham, Inc. `http://c2.com/cgi/wiki?MixIn`.

[MIWeb] Multiple inheritance definition at Wikipedia.com `http://en.wikipedia.org/wiki/Multiple_inheritance`.

[NDRS05] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart and Nathanael Schärli. *Adding Traits to (Statically Typed) Languages*. Technical Report, no. IAM-05-006, Institut für Informatik, December 2005, Technical Report, Universität Bern, Switzerland.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., 1983.

[Lien04] Adrian Lienhard. *Bootstrapping Traits*. Master thesis, University of Bern, Nov. 2004.

[OmniBr] Alexandre Bergel, Stéphane Ducasse, Colin Putney, Roel Wuyts. *Meta-Driven Browsers*. Proceedings of 14th International Smalltalk Conference (ISC 2006), LNCS, vol. 4406, Springer, 2007, pp. 134-156.

[WPR05] Hernán Wilkinson, Máximo Prieto and Luciano Romeo. *Arimethic with Measurements on Dinamically-Typed Object-Oriented Languages*. Practitioner report, Companion booklet of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA 2005.

[KB02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.

[WFWeb] Waterfall model definition at Wikipedia.com `http://en.wikipedia.org/wiki/Waterfall_model`

[KP88] Glenn E. Krasner and Stephen T. Pope. *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80*. Journal of Object-Oriented Programming, 1(3):26-49, August-September 1988.

10. Apéndice A. Notación gráfica

En esta sección se detalla la notación gráfica utilizada para los diagramas expuestos en este trabajo.

La notación utilizada se basa en la propuesta de la materia de "Programación Orientada a Objetos" dictada en la misma facultad en la que se encuentra contextualizado este trabajo. Esta notación, a su vez, fue extendida para poder expresar traits, relación de composición, especificar requerimientos de un trait, etc.

En ninguno de los diagramas es indispensable representar todas las posibles relaciones entre clases y traits ó todas las posibles colaboraciones entre objetos, sino solamente las necesarias para transmitir lo que se desea. Si bien pueden utilizarse para modelar de modo genérico, ésto ocurre rara vez. La intención es representar una situación o relación en particular.

10.1. Diagrama de Clases y Traits

Representa las clases y traits intervinientes en un modelo y el vínculo de herencia o composición existente entre ellas. El diagrama incluye parte del protocolo que poseen.

Por ejemplo:

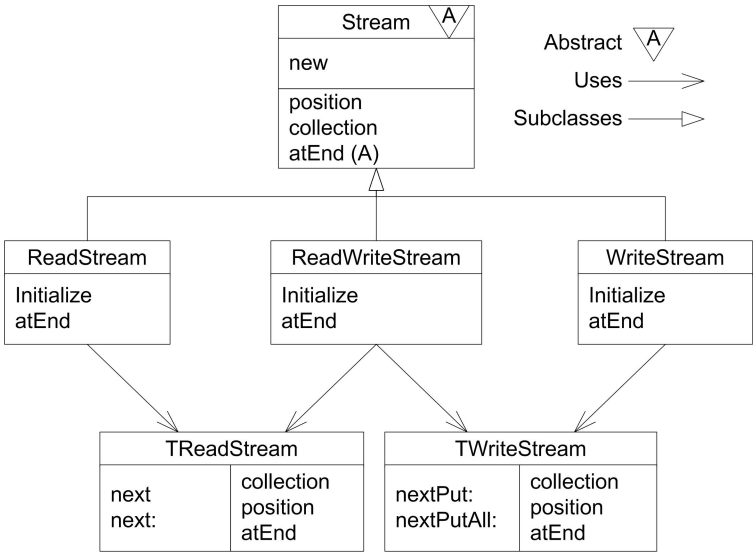

Figura 10.1: Diagrama de clases y traits

- Cada cajita representa una clase ó un trait.
- La flecha entre una clase y otra clase indica relación de herencia y va desde la subclase hacia la superclase.
- La flecha entre una clase o trait y otro trait indica relación de composición y va desde la clase o trait usuaria hacia el trait incluido en la composición.
- El triángulo con una A en el nombre de la clase indica que la misma es abstracta.
- En las clases no se incluyen los colaboradores internos, pero se indican los métodos de acceso a los mismos. De estos, solamente se usan getters en representación de ambos.

- En una clase los métodos de clase van arriba de los de instancia, separados por una línea. Si ningún método de clase es relevante, este sección puede omitirse.
- En un trait los métodos provistos van a la izquierda de los métodos requeridos, separados por una línea. Si ningún requerimiento del trait es relevante, esta sección puede omitirse.

Este otro tipo de diagrama, esta orientado a mostrar con más detalles como un clase o trait es compuesto por otro trait:

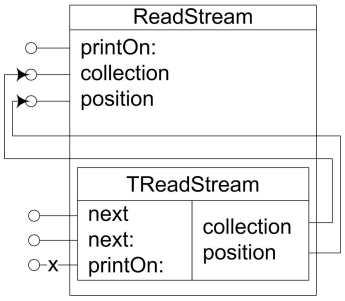

Figura 10.2: Diagrama de clases y traits detallado

- La cajita contenedora representa a la clase ó trait usuaria.
- La cajita contenida representa al trait siendo usado por la clase ó trait usuaria.
- Cada línea terminada en círculo es un método provisto por la clase, ya sea directamente (método definido en la misma clase ó mismo trait) ó indirectamente (método definido por un trait en uso).
- Aquellos métodos provistos por un trait que son excluidos en la composición de un usuario, es decir la clase o trait usuario excluye dicho método cuando usa al trait, se representa en la misma forma que un método provisto pero con una marca x sobre la línea con círculo. La misma representación aplica cuando un método provisto por el trait se encuentra eclipsado por la clase o trait usuario al definir el mismo método.
- La flecha representa un requerimiento del trait. Ésta puede estar conectada a un método provisto por la clase o trait usuaria indicando que la misma clase o trait usuaria satisface dicho requerimiento.

10.2. Diagrama de colaboración

Representa una vista dinámica del modelo, incluyendo la secuencia de envío de mensajes, que tiene lugar entre algunos objetos del modelo para llevar a cabo una tarea.

Por ejemplo:

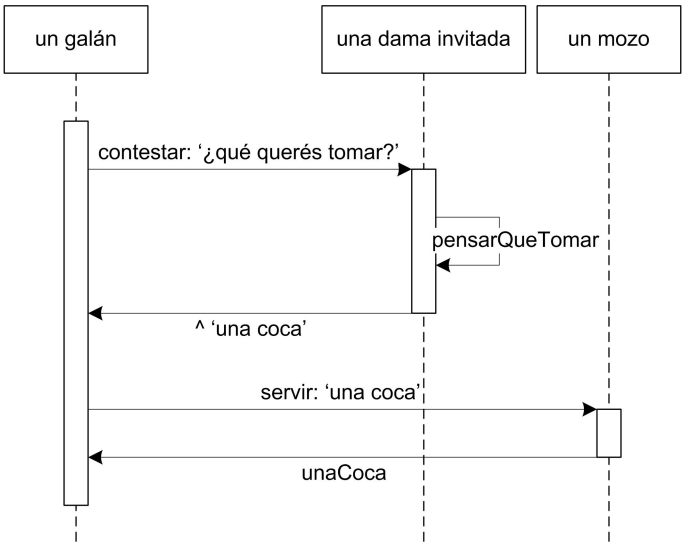

Figura 10.3: Diagrama de colaboración

- Cada cajita, al igual que en el diagrama de objetos, representa un objeto.
- El sentido del avance del tiempo es de arriba hacia abajo.
- Cada flecha representa el envío de un mensaje de un objeto a otro, ó el retorno de un objeto, que se denota mediante el símbolo ^.
- Un mensaje enviado por un objeto a sí mismo se indica con una flecha que baja y vuelve a la misma columna.

10.3. Diagrama de objetos

Representa una vista estática del modelo. En él, se incluyen los objetos intervinientes y las relaciones de colaboración entre ellos.

Por ejemplo:

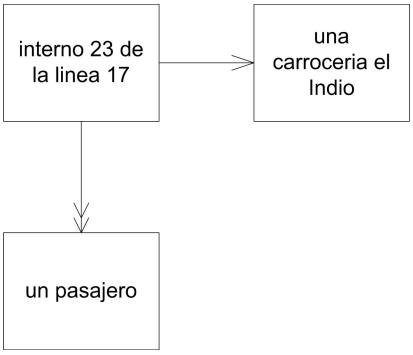

Figura 10.4: Diagrama de objetos

- Cada cajita representa un objeto. Esto es, una instancia particular y única en el universo. No es una forma genérica de referirse a objetos como podría ser `Colectivo`.

- Las flechas indican una relación estructural entre los objetos. Más específicamente, indica que los objetos pueden colaborar enviándose mensajes.
- La flecha simple hace referencia a una instancia particular, mientras que la doble indica una colección de objetos. Esto es un grupo de objetos no necesariamente homogéneos. La relación indica conocimiento del grupo, algo distinto a conocer cada integrante por separado.
- Puede colocarse una etiqueta sobre la flecha, indicando el nombre con que el objeto origen conoce al objeto destino, si esto es importante para distinguirla de otras ó para aclarar el significado de esta relación.