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

Implementación de refactorings automáticos en lenguajes con *LiveTyping*

Tesis de Licenciatura en Ciencias de la Computación

Fernando Balboa

Director: Lic. Hernán Wilkinson

Resumen

Todo programa tiene un ciclo de vida. La expectativa de vida del software en general crece junto con su tamaño y complejidad, y el desarrollo suele estar atado a un presupuesto acotado. A medida que el *codebase* muta y se vuelve más complejo, también se vuelve más difícil de mantener.

Una de las herramientas más frecuentes, que se encuentra por defecto en muchas de las *IDEs* comerciales, son los refactorings automáticos. Se conoce como refactoring a la acción de reestructurar el código, manteniendo su comportamiento actual. El objetivo de estos cambios queda a discreción del programador, aunque suele ser mejorar algún aspecto de mantenibilidad del código.

La automatización de las reestructuraciones provee dos ventajas. La primera es que aseguran al programador que el cambio que está introduciendo es seguro en términos de comportamiento, porque eliminan la posibilidad de error humano. La segunda es que promueven su uso frecuente para mejorar la mantenibilidad, haciendo que sean muy poco costosas en términos de tiempo y esfuerzo para el programador.

Este trabajo se enfoca en el desarrollo de dos refactorings automáticos: *Inline Temporary Variable* e *Inline Method*, ambos implementados en el lenguaje *Smalltalk-80*. La plataforma en la cual fueron integrados es *Cuis University*, la cual provee un sistema de recolección de datos de tipado (*LiveTyping*) que puede ser utilizado para mejorar la calidad de las herramientas. Uno de los objetivos principales es la exploración de las dificultades de implementación de los mismos en un lenguaje con tipado dinámico y cómo puede aprovecharse la información de *LiveTyping* en los refactorings automáticos.

Palabras clave: *refactoring, parser, parse node, source range, AST, intefaz, test, TDD.*

Tabla de contenidos

1. Introducción

Este trabajo presenta y analiza dos implementaciones de refactorings automáticos en un ambiente del lenguaje de programación *Smalltalk*. Un refactoring se define como un conjunto de operaciones que reestructuran un programa preservando su comportamiento, con el objetivo de ayudar en su diseño y evolución [6].

Los refactorings implementados fueron el *Inline Temporary Variable* y el *Inline Method*, ambos disponibles en la mayoría de los *entornos de desarrollo integrados* (*IDEs*) comerciales. La distribución en la cual fueron implementados fue *Cuis University*, una derivación de *Cuis Smalltalk* orientada a la enseñanza de conceptos de programación orientada a objetos que es utilizada en varias universidades del país.

Al tratarse de un lenguaje tipado dinámicamente, *Smalltalk* presenta algunas dificultades a la hora de implementar refactorings automáticos. El problema principal suele ser determinar las posibles clases receptoras de un envío de mensaje en particular. Para lidiar con este problema, *Cuis University* cuenta con un sistema llamado *LiveTyping* [7], que recopila información de tipos en tiempo de ejecución y la disponibiliza al resto del sistema para ser utilizada, por ejemplo, en la implementación de refactorings automáticos.

El objetivo principal del presente trabajo es otorgar a los usuarios de *Cuis University* estas nuevas herramientas para mejorar su experiencia con la plataforma, y al mismo tiempo permitir a los docentes explicar los conceptos detrás de ellos y demostrar su funcionamiento. Otro de los objetivos fue realizar el desarrollo de los mismos utilizando la técnica de *Test Driven Development* [2].

2. Contexto y definiciones

2.1. *Cuis University*

Smalltalk es tanto un ambiente interactivo como un lenguaje dinámico de programación orientado a objetos, diseñado en la década de 1970 por investigadores de *Xerox Palo Alto Research Center*. Su desarrollo fue realizado iterativamente en ciclos: primero se desarrollaba un sistema que resolviera las necesidades de software del momento; luego se creaban aplicaciones que testearan el sistema; finalmente, se reformulaba el entendimiento de dichas necesidades y se rediseñaba el entorno [5]. *Smalltalk-80* es el resultado de la quinta iteración de desarrollo del sistema y la primera en ser distribuida públicamente.

Actualmente, existen muchas distribuciones de *Smalltalk*, de las cuales algunas son de código abierto, siendo *Squeak*[1] y sus derivados las más populares entre ellas. Este último es un ambiente multiplataforma creado en 1996, que entre sus múltiples funcionalidades incluye *Morphic*, una interfaz gráfica que también es utilizada en otras distribuciones basadas en *Squeak*. Como todo ambiente de *Smalltalk*, utiliza una *Virtual Machine (VM)* para ejecutar el *bytecode* que produce el compilador. La implementación de la *VM* es *OpenSmalltalk Virtual Machine*[2], y es compartida con otros entornos de programación, como *Pharo, Cuis* y *Newspeak*.

Cuis Smalltalk[3] es una distribución basada en *Squeak* creada por Juan Vuletich con el objetivo de proveer una implementación multiplataforma rápida y lo más simple posible, intentando reducir al máximo la complejidad del sistema sin la pérdida de funcionalidad. *Cuis University*[4] (de aquí en más, *Cuis*) es a su vez una derivación de *Cuis Smalltalk* creada por Hernán Wilkinson y Máximo Prieto, orientada a la enseñanza del paradigma de programación de objetos y la ingeniería del software. Esta herramienta trae paquetes preinstalados para facilitar el aprendizaje a lo largo de un curso.

Smalltalk es además un ambiente gráfico e interactivo, donde cada componente está diseñado para poder ser presentado visualmente con el objetivo de ser observado y manipulado, e incluye todo tipo de herramientas de desarrollo tales como un administrador de paquetes para controlar versiones y hasta un debugger. Una de las más importantes, que se encuentra presente en toda implementación de *Smalltalk*, es el *Browser*. Desde esta interfaz pueden accederse a todas las clases del sistema, inspeccionar los mensajes y correr tests, además de ver y editar el código fuente. Por esta razón, todos los refactorings implementados en el presente trabajo son accesibles mediante el *Browser* de *Cuis University*.

2.2. Análisis y generación de código fuente

Como se mencionó previamente, en *Smalltalk*, el código fuente no es compilado directamente a lenguaje de máquina, sino que pasa por un estado intermedio conocido como

[1] Entorno *Squeak*: https://squeak.org
[2] Máquina virtual *OpenSmalltalk*: https://opensmalltalk.org
[3] Entorno *Cuis Smalltalk*: http://www.cuis-smalltalk.org
[4] Entorno *Cuis University*: https://sites.google.com/view/cuis-university/inicio?authuser=0

bytecode, el cual es interpretado y ejecutado posteriormente por la *VM*. La compilación del código ocurre de forma transparente al programador cada vez que un método se modifica y se guarda.

Como en todo proceso de compilación, el input es una tira de caracteres (el código fuente) de algún alfabeto. El analizador léxico es el componente encargado de leer cada caracter y agruparlos en entidades sintácticas llamadas *tokens*. El output es un stream de *tokens*, que es enviado al analizador sintáctico en la siguiente fase de compilación. Este último se encarga de analizar la cadena de tokens y determinar si es válida en términos de la definición sintáctica del lenguaje [1].

En *Cuis University*, la lógica de análisis sintáctico (o *parsing*) se encuentra en la clase `Parser`, que es subclase de `Scanner`, donde está implementado el análisis léxico. El parser toma el código fuente y retorna una instancia de `MethodNode` que representa la raíz de un grafo compuesto por instancias de subclases de `ParseNode`, que se asemeja a un árbol. Si bien en este caso no es exactamente un árbol, la estructura es conocida como *Abstract Syntax Tree (AST)*, dado que en la teoría de parsing y compiladores tradicionalmente sí se trata de un árbol. La siguiente Figura muestra el resultado que produce el parser sobre un código fuente sencillo.

```
aMethod

      ^self doSomethingWith: 15 and: 15
```

Id del nodo	Tipo de `ParseNode`	Código fuente	Descripción	Id del nodo padre

1	`MethodNode`	`aMethod`
		
		`^self doSomethingWith: 15 and: 15`
			El método entero. Es la raíz del grafo.	No tiene.

2	`TemporariesDeclarationNode`	`| |`	Declaración de variables temporales implícita. Vacía, ya que no se definió ninguna.	1
3	`BlockNode`	`[^self doSomethingWith: 15 and: 15]`	El cuerpo entero del método. Todo `MethodNode` tiene como hijo a un único `BlockNode` implícito.	1
4	`ReturnNode`	`^self doSomethingWith: 15 and: 15`	Expresión de retorno del método.	3

5	`MessageNode`	`self doSomethingWith: 15 and: 15`	Envío de mensaje a `self`	4
6	`VariableNode`	`self`	Receptor del mensaje	5
7	`SelectorNode`	`doSomethingWith:and:`	Selector del mensaje	5
8	`LiteralNode`	`15`	Literal que funciona como argumento del mensaje	5

Fig. 1 - Ejemplo de un *AST*

En la generación del *AST*, puede ocurrir que dos fragmentos de código distintos generen nodos equivalentes en el *AST*. Cuando esto ocurre, la implementación actual de *Cuis University* no crea dos nodos, sino que crea una única instancia y la referencia más de una vez. Es por esta razón que la estructura no puede ser considerada un árbol. A modo ilustrativo, en el ejemplo de la Figura anterior se crea un único nodo para el literal "15", a pesar de que aparece dos veces en el código fuente. Este detalle técnico tiene sentido al tener en cuenta que estos ambientes descienden de la implementación original de *Smalltalk-80*, diseñada en la década de 1970, donde el ahorro de memoria era una prioridad que hoy en día tal vez no es tan relevante como en aquel entonces.

Muchos de los refactorings disponibles actualmente en *Cuis* basan su implementación en el análisis del *AST* de un método para computar los cambios necesarios. En una primera instancia, los algoritmos creaban nuevas instancias de subclases de `ParseNode` con las modificaciones y luego generaban el código refactorizado a partir de ellas. El problema con esta forma de implementarlos era que se perdía el formato original del código fuente, lo cual podía resultar molesto para los programadores. Por esta razón, actualmente los refactorings trabajan directamente con reemplazos de strings, utilizando el *AST* para encontrar los fragmentos que necesitan ser modificados.

Al trabajar sobre strings, es necesario identificar la posición de un nodo en el código fuente. El concepto que encapsula esta idea en *Cuis* es conocido como *source range*. En general, un *source range* no es más que una instancia de la clase `Interval` (o subclase de ella), donde el inicio del intervalo se corresponde con la primera posición del nodo en el código fuente y el fin con la última. Esta información es recolectada en tiempo de compilación para la mayoría de los nodos.

Como se mencionó anteriormente, el *AST* no es un árbol, y por esta razón puede ocurrir que un mismo nodo tenga más de un *source range*. La Figura 2 retoma el ejemplo de la Figura 1 y muestra los *source ranges* para cada `ParseNode`.

```
aMethod
```

```
^self doSomethingWith: 15 and: 15
```

Tipo de ParseNode	Código fuente	Source ranges
MethodNode	aMethod ^self doSomethingWith: 15 and: 15	No tiene. Esta información no es recolectada para instancias de MethodNode porque es trivial.
TemporariesDeclarationNode	| |	No tiene porque es un nodo implícito.
BlockNode	[^self doSomethingWith: 15 and: 15]	No tiene porque es un nodo implícito.
ReturnNode	^self doSomethingWith: 15 and: 15	(11 to: 43)
MessageNode	self doSomethingWith: 15 and: 15	(12 to: 43)
VariableNode	self	(12 to: 15)
SelectorNode	doSomethingWith:and:	No es reportado porque no aparece en este formato en el código fuente.
LiteralNode	15	(34 to: 35), (42 to: 43)

Fig. 2 - Ejemplo de *source ranges* de un *AST*

Un detalle implementativo relevante para el presente trabajo es la diferencia entre lo que se conoce como un *raw source range* y un *complete source range*. Originalmente, los rangos reportados por el *parser* a veces no incluían el nodo completo. Por ejemplo, el *raw source range* de un envío de mensaje no contenía la posición del receptor. Los *complete source ranges* solucionan esta clase de problemas y son muy útiles para la implementación de refactorings, por lo cual son los usados en la implementación provista. Todas las diferencias entre *raw* y *complete source ranges* pueden ser consultadas en [5].

Como se mencionó previamente, los rangos suelen ser instancias de `Interval`. En algunos casos, también se utiliza la subclase `SourceCodeInterval`, creada con el propósito de

reificar intervalos de código, particularmente para su uso en la implementación de refactorings [5]. Para lograr el objetivo del presente trabajo fue necesario ampliar las funcionalidades de esta clase. Algunas de las mejoras agregadas fueron:

- La capacidad de expandir un intervalo hasta el próximo caracter que no represente un espacio en blanco (por ejemplo, un *newline* o un *tab*).
- La capacidad de expandir un intervalo hasta el inicio del siguiente *statement* de código.
- La capacidad de expandir un intervalo hasta el final del *statement* actual.
- Detectar si un intervalo termina en el caracter de finalización de un *statement* (".").
- Detectar si el intervalo termina en el final del último *statement* del código fuente.
- Detectar si el intervalo termina en el final de la declaración de variables locales.

Otra funcionalidad requerida para la implementación de los refactorings fue el poder determinar si un método referencia o no a las pseudovariables *self* y/o *super*. Para lograrlo, fue necesario agregar métodos en varias subclases de `ParseNode` que analizan recursivamente el *AST* en búsqueda de referencias a las mismas.

Como se verá más adelante, el análisis del *AST* es uno de los aspectos más fundamentales de la implementación de los refactorings en *Smalltalk*, dado que la técnica permite desglosar el código fuente sintácticamente para computar los cambios necesarios.

2.3. Refactorings

Un refactoring puede definirse como un conjunto de operaciones que reestructuran un programa preservando su comportamiento, usualmente con el objetivo de ayudar en su diseño y evolución. Existen una serie de refactorings que son tan comunes que a lo largo del tiempo han sido nombrados y automatizados por las *IDEs*. Por lo general, estos suelen utilizarse como paso intermedio de cambios manuales más complejos. Ejemplos de estos son el *Extract Temporary* o el *Extract Method*, cuya función es convertir una expresión en una variable temporal o encapsularla en un método respectivamente.

Los refactorings implementados en el presente trabajo son conocidos como *Inline Temporary Variable* e *Inline Method*, y tienen el efecto inverso a los recién mencionados. Si bien la bibliografía clásica que acuñó el término "refactoring" [6] no los describe formalmente, los mismos se encuentran incluídos en la mayoría de las *IDEs* comerciales, por lo que puede definirse una idea general de su funcionamiento. Estas definiciones difieren ligeramente en algunos aspectos a las de algunos autores más recientes [3], y pueden encontrarse en las secciones *3.1.1* y *3.2.1* respectivamente.

2.3.1. Estado del arte

Actualmente, prácticamente todas las *IDEs* comerciales proveen implementaciones de algunos de los refactorings automáticos más útiles. Algunas tienen además funcionalidades agregadas para mejorar la experiencia de usuario, como puede ser mostrarle al usuario cómo se vería el código luego de los cambios a realizar (*preview*), o atajos del teclado para iniciarlos o revertirlos.

Como fue mencionado en la sección *2.1*, existen múltiples distribuciones de *Smalltalk*, y cada una de ellas provee implementaciones y features que varían ligeramente entre ellas. En este trabajo se tuvieron en cuenta para el análisis y comparación con *Cuis* los ambientes *Pharo*[5] y *Squeak*, ambos *open source*.

En primer lugar, a diferencia de *Cuis*, ni *Pharo* ni *Squeak* proveen atajos del teclado (*shortcuts*) para iniciar la mayoría de los refactorings (incluídos los desarrollados en este trabajo). Esto hace que el proceso de usarlos sea mucho más incómodo para el usuario. Por otro lado, *Pharo* es el único que soporta ver un *preview* del refactoring, es decir, ver cómo quedaría el código después de ejecutarlo antes de decidir si hacerlo. Además, también es el único que permite deshacerlos, lo cual resulta extremadamente útil.

Las herramientas de refactoring de *Pharo* y de *Squeak* están basadas en la implementación original del *Refactoring Browser*, que fue el primer proyecto en *Smalltalk* en abordar esta temática, pero que no incluía los refactorings implementados en este trabajo. Fue creado por primera vez para el entorno *VisualWorks*[6] y luego porteado a más plataformas, donde fue sufriendo modificaciones en cada una. Por esta razón, a diferencia de *Cuis*, tanto *Pharo* como *Squeak* funcionan creando nodos modificados del *AST* original y produciendo el nuevo código a partir de ellos, lo cual provoca la pérdida de formato mencionada en la sección anterior. Si bien esto puede resultar molesto para el programador, la decisión de no generar el código nuevo a partir de un *AST* le agrega una responsabilidad más a la implementación del refactoring dado que es necesario lidiar con aspectos del *formatting*.

Otra de las diferencias es que al estar basadas en el *Refactoring Browser*, tanto *Pharo* como *Squeak* trabajan sobre un *AST* únicamente creado para los refactorings, a diferencia de *Cuis*, que utiliza el mismo que se usa en la compilación. Esto provoca que haya clases duplicadas que tienen propósitos muy similares, complejizando el sistema.

Con respecto a los refactorings implementados en este trabajo, puede encontrarse una comparativa de lo que ofrece cada entorno sobre *Inline Temporary Variable* e *Inline Method* en las secciones *3.1.5* y *3.2.6* respectivamente.

[5] Entorno *Pharo*: https://pharo.org
[6] Entorno *VisualWorks*: https://www.cincomsmalltalk.com/main/

3. Refactorings

En *Cuis Smalltalk*, todos los refactorings están implementados siguiendo un cierto esquema, que si bien es algo laxo, funciona como una guía general para crear nuevos refactorings y facilitar su entendimiento. Las clases principales del sistema son:

- **`Refactoring` y sus subclases**. Estos objetos son los que contienen la lógica particular de cada refactoring y se encargan de analizar el código y realizar los cambios requeridos. En algunos casos también pueden validar precondiciones. El único método de instancia de la clase `Refactoring` es `Refactoring>>apply`, que debe ser implementado por sus subclases no abstractas y es el encargado de aplicar todas las modificaciones.
- **`RefactoringApplier` y sus subclases.** Los appliers son los encargados de pedir los parámetros relevantes para el refactoring al usuario a través de una interfaz gráfica. Además, se encargan de crear el objeto que realizará los cambios (alguna subclase de `Refactoring`), y opcionalmente pueden realizar validaciones previas a dicha creación.
- **`RefactoringPrecondition` y sus subclases.** Algunos refactorings comparten validaciones entre sí. Para evitar la repetición innecesaria de código, algunas precondiciones fueron reificadas en instancias de la clase `RefactoringPrecondition`.
- **`RefactoringError` y `RefactoringWarning`.** Estas se utilizan a la hora de manejar posibles errores durante la validación o ejecución de los refactorings.
- **`RefactoringMenues`.** Provee los menús con los refactorings disponibles y define atajos de teclado hacia los mismos.
- **`SmalltalkEditor`.** Esta clase representa un editor de código de Smalltalk, y suele ser el punto de entrada a los distintos appliers.

3.1. *Inline temporary variable*

3.1.1. Definición

Este refactoring consiste en reemplazar dentro un método las referencias a una variable local por su valor (que puede ser una expresión arbitrariamente compleja), eliminando también su declaración en caso de que deje de ser necesaria. Esta herramienta suele ser útil cuando a la variable se le asigna una expresión simple y luego el programador decide que es más legible utilizar directamente dicha expresión en vez de agregar una indirección.

La implementación propuesta en este trabajo permite iniciar el refactoring desde la declaración de la variable temporal o desde una referencia a ella. El primer caso sólo es válido si a la variable se le asigna una expresión una única vez, y fallará en caso de que esta precondición no se cumpla. Para realizar el refactoring, se requieren los siguientes parámetros de entrada:

- Nombre de la variable a refactorizar.
- Intervalo de código desde el cual se ejecuta el refactoring. Antes de crear el refactoring, se valida que sea una referencia a la variable.
- El método a refactorizar.

A continuación se presentan algunos ejemplos de cómo funciona el refactoring *Inline Temporary Variable* al utilizarse sobre la variable *t1*. En el primer caso, la variable tiene una sola asignación y el cambio puede iniciarse desde ella, desde su uso en la expresión de retorno, o desde su declaración con el mismo resultado:

Antes del refactoring

```
aMethod

	| t1 t2 |

	t1 := 20.
	t2 := 10 + 1.
	^t1.
```

Después del refactoring

```
aMethod

	| t2 |

	t2 := 10 + 1.
	^20.
```

Fig. 3 - Ejemplo 1 de *Inline Temporary Variable*

En el siguiente ejemplo, la variable es utilizada en la definición de *t2*, y es necesario agregar paréntesis al reemplazar la expresión para mantener la semántica del código original.

Antes del refactoring

```
aMethod

	| t1 t2 |

	t1 := 5 + 5.
	t2 := 4 * t1.
	^t2.
```

Después del refactoring

```
aMethod

	| t2 |
```

```
	t2 := 4 * (5 + 5).
	^t2.
```

Fig. 4 - Ejemplo 2 de *Inline Temporary Variable*

En este ejemplo la variable tiene dos asignaciones por lo que no es posible iniciar el refactoring desde la declaración, sino que es necesario hacerlo desde la primera asignación, desde su uso en la línea siguiente (con el mismo resultado) o desde la segunda asignación.

Antes del refactoring

```
aMethod

	| t1 t2 |

	t1 := 20.
	t2 := t1 + 1.
	t1 := 30.
	^t1.
```

Después del refactoring si se inicia desde la primera declaración

```
aMethod

	| t1 t2 |

	t2 := 20 + 1.
	t1 := 30.
	^t1.
```

Después del refactoring si se inicia desde la segunda declaración

```
aMethod

	| t1 t2 |

	t1 := 20.
	t2 := t1 + 1.
	^30.
```

Fig. 5 - Ejemplo 3 de Inline Temporary Variable

Si la variable se encuentra declarada pero no referenciada y el refactoring se inicializa desde dicha declaración, el resultado final es simplemente la eliminación de la misma. El siguiente ejemplo muestra este caso de uso.

Antes del refactoring

```
aMethod

	| t1 t2|

	t2 := 10.
	^t2 * 2.
```

Después del refactoring

```
aMethod

	| t2 |

	t2 := 10.
	^t2 * 2.
```

Fig. 6 - Ejemplo 4 de *Inline Temporary Variable*

3.1.2. Interfaz gráfica

Si bien los parámetros del refactoring incluyen un intervalo y un nombre de variable, estos son provistos en la práctica por la interfaz de usuario para facilitarle el uso de la herramienta al programador. Para acceder a la misma, sólo es necesario ubicar el cursor sobre la referencia a la variable, abrir el menú contextual y clickear la opción correspondiente como muestra la Figura 7. Otra alternativa es ubicar el cursor como fue indicado y usar el atajo *Ctrl+2* en *Windows* o *GNU/Linux*, o *Command+2* en *MacOS*. Para mejorar la experiencia de usuario, también fue considerado el caso donde una expresión es asignada a una variable temporal: en esta situación, aunque el cursor se encuentre en cualquier parte del statement, se aplicará el refactoring sobre la variable del lado izquierdo. Dado que no es necesario ningún otro input del usuario, el cambio en el código se refleja inmediatamente.

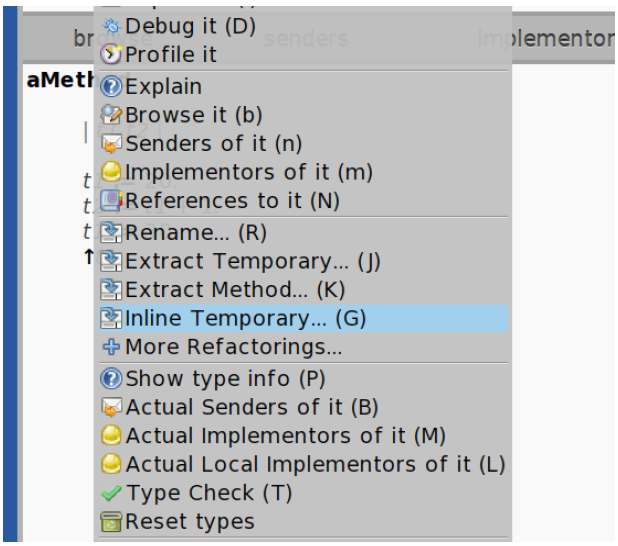


Fig. 7 - Menú desde el cual se inicia el refactoring *Inline Temporary Variable*

3.1.3. Implementación

La implementación provista se encuentra en la clase `InlineTemporaryVariable`. En esta, el valor que se reemplaza en las referencias es la expresión del lado derecho en la asignación previa más cercana a la línea desde donde se inicializa. Como muestra la Figura 5, el *scope* de los cambios comienza en dicha asignación y se extiende hasta la próxima, si existe, o hasta el fin del código si no.

Para poder ejecutar correctamente el refactoring en el lenguaje *Smalltalk*, es necesario que se cumplan algunas precondiciones sobre el código fuente original. Las implementaciones de la mayoría de los refactorings existentes en esta distribución validan que es posible realizar los cambios sobre un determinado fragmento de código antes de instanciar el objeto que se encargará de realizarlos. La implementación provista de *Inline Temporary Variable* realiza tres validaciones, pero lo hace en tiempo de ejecución por cuestiones de simplicidad.

La primera precondición es que no está permitido activar el refactoring desde un uso de una variable que nunca fue asignada, dado que no hay valor por el cual reemplazar las referencias. La segunda es que si el refactoring se inicia desde la declaración de una variable temporal, se requiere que haya a lo sumo una asignación a la misma. Esto se debe a que al inicializarse desde ahí, no queda determinado qué valor utilizar al hacer el inline, dado que hay más de uno posible, ni entre qué líneas debe aplicarse.

El tercer requerimiento es que la variable no sea utilizada dentro de un bloque y sea asignada posteriormente. Esto es necesario debido a que el bloque puede ser evaluado antes o después de dicha asignación, por lo cual podría ocurrir que cambie la semántica al remplazar la referencia por el valor previo más cercano. La Figura 8 muestra qué ocurriría si pudiera ejecutarse el refactoring en este caso de uso.

Antes del refactoring

```
aMethod

	| t1 b |

	t1 := 10.
	b := [t1 + 1].
	t1 := 20.
	^b value. "Resultado: 21"
```

Después del refactoring inicializado desde la primera asignación a *t1* (si pudiera ser ejecutado)

```
aMethod

	| t1 b |

	b := [10 + 1].
	t1 := 20.
	^b value. "Resultado: 11"
```

Fig. 8 - Ejemplo de *Inline Temporary Variable* con bloque entre asignaciones

Para instanciar un objeto de la clase `InlineTemporaryVariable` se utiliza el método de clase `InlineTemporaryVariable class>>named:atUsageInterval:inMethod:`. Los parámetros deben ser de tipo `String`, `Interval` y `CompiledMethod` respectivamente, y se corresponden con los mencionados en la definición del refactoring. A continuación se presenta la implementación de dicho método.

```
InlineTemporaryVariable class>>named: tempVarToInlineName atUsageInterval:
usageInterval inMethod: compiledMethodToRefactor

	| oldVariableNodeAndUsageInterval methodNode |
	methodNode := compiledMethodToRefactor methodNode.
	oldVariableNodeAndUsageInterval := self findTemporaryNamed:
tempVarToInlineName atUsage: usageInterval
		inMethodNode: methodNode.

	^self new
		initializeOldVariableNode: oldVariableNodeAndUsageInterval
first
		usage: oldVariableNodeAndUsageInterval second
```

```
            method: compiledMethodToRefactor
            methodNode: methodNode.
```

Fig. 9 - Implementación de `InlineTemporaryVariable class>>named:atUsageInterval:inMethod:`

En este punto, si en el código fuente no se encuentra una variable que se llame como `tempVarToInlineName` en el intervalo `usageInterval`, el refactoring falla con el mensaje de error *"Selected interval is not a temporary variable"*. A continuación se presenta la definición de la clase `InlineTemporaryVariable`.

Definición de `InlineTemporaryVariable`

```
Refactoring subclass: #InlineTemporaryVariable
      instanceVariableNames: 'variableToInline methodToRefactor
updatedSourceCode methodNode oldVariableNode usageToInline
sourceCodeChanges methodOrBlockNodeDeclaringTemporary
rangeOfNodeDeclaringTemporary assignmentToInlineRange'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Tools-Refactoring'
```

Comentario de clase

```
I am a refactoring that replaces references to a temporary variable for its
actual value.
Implementation notes:

- If the refactoring is initiated from the declaration of the variable,
then it must have at most one assignment.
- If there are no assignments, the only effect is the removal of the
declaration.
- If the refactoring is triggered from a usage, the value used is the one
from the previous last assignment and
the scope is limited to all usages up to the next assignment, if any, or
the end of the method.
```

Fig. 10 - Definición de la clase `InlineTemporaryVariable`

La siguiente tabla describe los colaboradores internos que se muestran en la figura anterior.

Variable de instancia	Tipo	Descripción
`variableToInline`	`String`	Nombre de la variable a reemplazar
`methodToRefactor`	`CompiledMethod`	Método sobre el cual se ejecuta el refactoring
`updatedSourceCode`	`String`	Variable donde se almacena el código resultante luego de aplicar el refactoring
`methodNode`	`MethodNode`	Nodo de método (raíz) de `methodToRefactor`
`oldVariableNode`	`TempVariableNode`	El nodo de variable correspondiente a `variableToInline`
`usageToInline`	`SourceCodeInterval`	Intervalo desde el cual se inicializa el refactoring
`sourceCodeChanges`	`SortedCollection`	Colección donde se almacenan los cambios de código a realizar en el último paso del refactoring
`methodOrBlockNodeDeclaringTemporary`	`MethodNode|BlockNode`	Nodo en el cual se declara la variable a refactorizar. Si está declarada al inicio del método entonces será un `MethodNode`. Si es local a un bloque, será un `BlockNode`
`rangeOfNodeDeclaringTemporary`	`SourceCodeInterval`	El intervalo de código que abarca `methodOrBlockNodeDeclaringTemporary`
`assignmentToInlineRange`	`SourceCodeInterval`	El intervalo de código donde se encuentra la asignación a la variable de la cual se extrae el valor a reemplazar en las referencias a la misma. Puede ser `nil` si no existe dicha asignación.

Fig. 11 - Tabla de variables de instancia de `InlineTemporaryVariable`

Como en todos los refactorings de *Cuis Smalltalk*, `InlineTemporaryVariable` sólo debe implementar el mensaje `#apply` de su clase padre. Este delega la mayoría del trabajo en métodos internos que se encargan de computar el nuevo código fuente y compilarlo. Todos los reemplazos de código necesarios son calculados y almacenados en la variable de instancia `sourceCodeChanges` como instancias de `Association` de `Interval` a `String`. Al final del proceso, el mensaje `#compileChanges` se encarga de aplicarlos todos a la vez sobre el código original y guardar el resultado en `updatedSourceCode` para ser compilado.

A continuación se incluye un fragmento de la implementación que muestra el cuerpo del método interno que realiza el inline de una variable hasta cierta posición en el código fuente. El primer parámetro es un `SourceCodeInterval` que contiene una asignación a la variable (de esta se toma valor que será reemplazado en posteriores referencias). El segundo es un índice que marca hasta dónde es necesario realizar cambios (la próxima asignación o el fin del método si no existe otra). Puede verse que en este punto se obtiene la expresión que se usará para hacer los reemplazos y se delega el cómputo de los cambios en otro método interno. La línea final se encarga de borrar la asignación refactorizada del código original.

```
InlineTemporaryVariable>>inlineAssignment: assignmentToInline upTo: anIndex

      | assignmentNode assignmentNodeValue expression expressionRange |

      assignmentNode := self assignmentNodeToInline: assignmentToInline.
      assignmentNodeValue := assignmentNode value.
      expressionRange := (methodNode completeSourceRangesOf:
assignmentNodeValue ifAbsent: [self shouldNotHappen])
            detect: [:range | assignmentToInline includesAllOf: range].
      expression := methodNode sourceText copyFrom: expressionRange first
to: expressionRange last.

      self inlineAssignmentNode: assignmentNode withExpression: expression
upTo: anIndex;
            removeAssignment: assignmentToInline.
```

Fig. 12 - Fragmento de la implementación de `InlineTemporaryVariable`

Buena parte de la complejidad de la implementación provista se encuentra en la lógica que determina si es necesario o no agregar paréntesis a la expresión al reemplazar una referencia a una variable. Esto depende tanto del valor a insertar como del punto de inserción. La Figura 13 muestra todos los casos en los cuales se insertan paréntesis al reemplazar las referencias a la variable *t*. Si ninguna de las reglas aplica, entonces no se agregan paréntesis.

Regla	Ejemplo de declaración	Ejemplo de referencia	Ejemplo de reemplazo

El valor es un mensaje binario y la referencia es el argumento de otro mensaje binario.	t := 3 * 5.	a := 1 + t.	a := 1 + (3 * 5)
El valor es un mensaje binario y la referencia es la recibidora de un mensaje unario	t := 3 * 5.	a := t cubed.	a := (3 * 5) cubed
El valor es un mensaje *keyword* y la referencia es la recibidora de un mensaje binario	t := self m2: 5.	a := 1 + t * 2	a := 1 + (self m2: 5) * 2
El valor es un mensaje *keyword* y la referencia es la recibidora de un mensaje unario	t := self m2: 5.	a := t cubed	a := (self m2: 5) cubed

Fig. 13 - Reglas de agregado de paréntesis para *InlineTemporaryVariable*

Por último, dado que en este refactoring algunas validaciones se hacen durante la ejecución, fue necesario cambiar ligeramente el método `#createAndValueHandlingExceptions:` de la clase `InlineTemporaryVariableApplier` para poder mostrar correctamente los mensajes de error al usuario. Dado que el refactoring no requiere realmente de input del usuario una vez que se inicia, el código del applier es muy sencillo porque su única función es la creación de una instancia de `InlineTemporaryVariable`.

3.1.4. Testing

Como se mencionó en la introducción de este trabajo, la implementación fue mayormente realizada utilizando la técnica *TDD*. Para lograrlo, se crearon un total de 31 casos de tests, de los cuales 26 son casos exitosos donde el refactoring puede ser aplicado y 5 son casos de error, donde se verifica que no se pueden realizar los cambios porque el código no cumple alguna de las precondiciones. Para el primer grupo, la confirmación de que el refactoring fue ejecutado correctamente se realiza comparando el nuevo código fuente compilado con un output esperado. En los casos de falla, se atrapan los errores y se checkea que contengan el mensaje deseado.

A continuación se lista el propósito de cada uno de los tests de casos exitosos, en el orden en el que fueron implementados.

1. Cuando se aplica sobre una variable sin usos, simplemente se borra su declaración.
2. Cuando se le asigna un valor a la variable pero luego no es referenciada, se borran tanto su declaración como la asignación.
3. El scope del refactor es entre asignaciones si hay más de una.
4. Las referencias posteriores a una asignación son reemplazadas por el valor asignado
5. No se agregan paréntesis cuando el valor de la variable es un literal y la referencia es la receptora de un mensaje unario.
6. No se agregan paréntesis cuando el valor de la variable es un literal y la referencia es el argumento de un mensaje binario.
7. Se agregan paréntesis cuando el valor de la variable es un mensaje binario y la referencia es el argumento de un mensaje binario.
8. No se agregan paréntesis cuando el valor de la variable es un mensaje binario y la referencia es la receptora de un mensaje binario.
9. Se agregan paréntesis cuando el valor de la variable es un mensaje binario y la referencia es la receptora de un mensaje unario.
10. Se agregan paréntesis cuando el valor de la variable es un mensaje binario y la referencia es parte de un mensaje donde el receptor es otra referencia a la variable.
11. No se agregan paréntesis cuando el valor de la variable es un mensaje binario y la referencia es el argumento de un mensaje keyword.
12. Se agregan paréntesis cuando el valor de la variable es un mensaje keyword y la referencia es la receptora de un mensaje binario.
13. No se agregan paréntesis cuando el valor de la variable es un mensaje keyword y la referencia es el argumento de un mensaje binario.
14. Se agregan paréntesis cuando el valor de la variable es un mensaje keyword y la referencia es parte de un mensaje donde el argumento es otra referencia a la variable.
15. No se agregan paréntesis extra cuando el valor o la referencia ya tienen.
16. Si hay dos asignaciones a la variable y el valor de la segunda contiene una autoreferencia, entonces al ejecutar el refactoring sobre la primera asignación, también se reemplaza con su valor la referencia en la segunda.
17. Si existe más de un bloque que declare una variable con el mismo nombre, y se ejecuta el refactoring en uno de ellos, los posteriores no son afectados.
18. Si existe más de un bloque que declare una variable con el mismo nombre, y se ejecuta el refactoring en uno de ellos, los anteriores no son afectados.
19. Aplicar el refactoring sobre una variable temporal declarada en un bloque anidado no afecta a otros bloques que declaren otra variable local con el mismo nombre.
20. Ejecutar el refactoring desde la declaración dentro de un bloque no afecta a bloques anteriores.
21. El refactoring solo borra la declaración de la variable afectada y deja las otras como estaban.

22. Se agregan paréntesis cuando el valor de la variable es un mensaje keyword y la referencia es el argumento de un mensaje keyword.
23. El refactoring puede realizarse cuando la variable es utilizada en un bloque luego de una asignación pero no existe ninguna otra.
24. El refactoring puede realizarse cuando hay una referencia a la misma al final del método, y no hay un caracter de finalización de *statement*.
25. El refactoring puede iniciarse desde una referencia posterior a la asignación relacionada.
26. Cuando hay una única asignación y ocurre dentro de un bloque, el scope del refactoring termina en el final de dicho bloque.

Los casos de uso de falla verifican que las precondiciones deben cumplirse para ejecutar el refactoring:

1. La variable seleccionada debe tener un nombre no vacío.
2. El intervalo de código con el cual se instancia el refactoring debe contener a una variable temporal con el nombre indicado en el código fuente del método seleccionado.
3. La variable no puede ser refactorizada si es utilizada dentro de un bloque y posteriormente se le asigna un valor.
4. Si el refactoring se inicia desde una declaración, debe existir a lo sumo una sola asignación a la variable. Nótese que puede ocurrir que la misma no esté referenciada en ningún otro lado, en cuyo caso no habrá ninguna asignación.
5. Si el refactoring se inicia desde una declaración y existe una referencia a la variable, debe existir exactamente una asignación a la misma. Si esta precondición no se cumple es porque el código previo al refactoring fallaría en runtime, dado que se está utilizando una variable sin valor.

3.1.5. Comparación entre entornos de *Inline Temporary Variable*

La Figura 14 presenta un listado de casos de uso y cómo se comporta el refactoring en cada distribución analizada. En general, el caso más frecuente es hacer inline sobre una variable que tiene una única asignación, lo cual está soportado en todos los entornos. Sin embargo, la implementación provista va más allá y es la única que puede manejar el caso donde hay dos o más, o incluso ninguna, lo cual puede resultar útil (y también suele ser soportado en *IDEs* comerciales).

Otras diferencias relevantes encontradas son que *Pharo* en algunos casos lanza advertencias pero permite continuar con el refactoring, lo cual suele terminar en cambios semánticos del código. Por su parte, *Squeak* es muy limitado y algo incómodo de usar, dado que es necesario seleccionar el *statement* de asignación entero para iniciar el refactoring, y de no hacerlo, se abre el debugger en vez de una advertencia.

Caso de uso	*Pharo(9.0)*	*Squeak(5.3)*	*Cuis University(v5093)*

Refactorizar método cuando hay más de una asignación a la variable	No soportado. Lanza advertencia y permite proseguir, pero falla al hacerlo	No soportado. Lanza advertencia y no permite proseguir	Soportado. Ejecuta el inline hasta la próxima asignación
Refactorizar una variable desde su declaración cuando es asignada posteriormente una única vez	Soportado	No soportado. El refactoring debe ser iniciado desde un *statement* de asignación	Soportado
Refactorizar una variable desde su declaración cuando no es asignada posteriormente	No soportado. Se abre el debugger	No soportado	Soportado
El refactoring mantiene el formato original del código fuente	No soportado	No soportado	Soportado
Refactorizar una variable declarada a nivel del método pero asignada únicamente dentro de un bloque	No soportado. Lanza advertencia, y si se prosigue, cambia la semántica del método	No soportado	Soportado. Se modifican únicamente las referencias dentro del bloque de la asignación
Agregado de paréntesis	Soportado. A veces los agrega aunque no sean necesarios	Soportado. A veces los agrega aunque no sean necesarios	Soportado. Sólo se agregan si son necesarios

Fig. 14 - Comparación de casos de uso de *Inline Temporary Variable* entre distintas distribuciones de *Smalltalk*

3.2. *Inline method*

3.2.1. Definición

Este refactoring consiste en reemplazar el envío de un mensaje por el cuerpo del método invocado. Dicho mensaje puede ser de la misma clase o de cualquier otra, puede tener opcionalmente argumentos y puede o no tener un valor de retorno explícito (en *Smalltalk*, cuando no hay un retorno explícito siempre se retorna `self` de forma implícita). Además, el programador puede elegir qué envíos del mensaje refactorizar, y si desea eliminar su declaración al finalizar los cambios. Todos estos factores, entre otros, alteran el funcionamiento

del refactoring, pero en todos los casos el objetivo es preservar la semántica del código original.

Si bien todos los casos de uso se encuentran descritos en la sección *3.2.5*, a continuación se presenta un subconjunto de ellos que muestra las partes más importantes del comportamiento de la implementación propuesta de *Inline Method*. Se asume que el refactoring es aplicado siempre sobre el único mensaje envíado en el método `aSender`.

El primer ejemplo consiste en uno de los casos más sencillos, donde el método invocado pertenece a la misma clase y no tiene ni argumentos ni valor de retorno.

Antes del refactoring

```
aSender

      self aMethodToInline.
      ^true
```

```
aMethodToInline

      self doSomething.
```

Después del refactoring

```
aSender

      self doSomething.
      ^true
```

Fig. 15 - Ejemplo 1 de *Inline Method*

Cuando el método tiene parámetros, es necesario reemplazarlos por los valores utilizados en la invocación que está siendo refactorizada. El siguiente ejemplo muestra este caso.

Antes del refactoring

```
aSender

      self doWithString: 'hello' andInt: 10.
```

```
doWithString: aString andInt: anInt

      self doWorkWith: aString and: anInt.
```

Después del refactoring

```
aSender
```

```
      self doWorkWith: 'hello' and: 10.
```

Fig. 16 - Ejemplo 2 de *Inline Method*

El refactoring puede volverse más complejo cuando el método sobre el cual se hace el inline tiene un valor de retorno. El mismo puede ser directamente utilizado o no en la colaboración, e incluso podría ser asignado a una variable.

A continuación se presentan otros dos casos. En el primero el valor de retorno es ignorado, mientras que en el segundo se guarda en un colaborador local de la clase que invoca al método.

Antes del refactoring

```
aSender

      self doWithString: 'hello' andInt: 10.
      ^'Done'
```

```
doWithString: aString andInt: anInt

      self doWorkWith: aString and: anInt.
      ^true
```

Después del refactoring

```
aSender

      self doWorkWith: 'hello' and: 10.
      ^'Done'
```

Fig. 17 - Ejemplo 3 de *Inline Method*

Antes del refactoring

```
aSender

      | res |

      res := self doWithString: 'hello' andInt: 10.
      ^res.
```

```
doWithString: aString andInt: anInt

      self doWorkWith: aString and: anInt.
      ^true
```

Después del refactoring

```
aSender

      | res |

      self doWorkWith: 'hello' and: 10.
      res := true.
      ^res.
```

Fig. 18 - Ejemplo 4 de *Inline Method*

En los casos presentados hasta el momento, el mensaje sobre el cual se hace inline no utilizaba variables locales. Si las hay, es necesario declararlas con algún nombre en el método invocador. Por cuestiones de legibilidad, en general son definidas con el mismo nombre que tenían antes, pero podría ocurrir que este ya se encuentre en uso por alguna otra variable o parámetro en el nuevo contexto. Cuando esto ocurre, es necesario definir algún algoritmo de resolución de nombres. En la sección de *3.2.3* se explica el utilizado en este trabajo.

Como se ha mencionado anteriormente, el lenguaje *Smalltalk* incluye el concepto de *closure* o *bloque*. Estos pueden tener asociadas variables temporales que sólo son visibles dentro de los mismos. Por esta razón, si se realiza un inline de un mensaje con variables locales, a la hora de declararlas en el nuevo contexto hay que elegir si hacerlo al nivel del método o nivel del bloque. La implementación propuesta utiliza esta última forma, que es ilustrada en el siguiente ejemplo.

Antes del refactoring

```
aSender

      | methodVar |

      methodVar := 4.
      ^[
            | blockVar |
            blockVar := 'Block variable'.
            self doWithString: blockVar andInt: 20.
      ] value + methodVar.
```

```
doWithString: aString andInt: anInt

      | myVar |

      myVar := 5.
      self doWorkWith: aString and: anInt.
      ^myVar
```

Después del refactoring

```
aSender

      | methodVar |

      methodVar := 4.
      ^[
            | blockVar myVar |
            blockVar := 'Block variable'.
            myVar := 5.
            self doWorkWith: blockVar and: 20.
            myVar.
      ] value + methodVar.
```

Fig. 19 - Ejemplo 5 de *Inline Method*

Este último ejemplo también muestra otro detalle sobre el comportamiento de *Inline Method*. Dado que el mensaje sobre el cual se aplica el refactoring funciona como valor de retorno del bloque por ser su última línea, es necesario que el resultado de `doWithString:andInt:` (`myVar` en el ejemplo) pase a ser la línea final. Vale destacar que no debe contener el caracter de retorno `^`, dado que en la semántica original el bloque no es un *full closure*.

Similar al caso de *Inline Temporary Variable*, a veces realizar un inline requiere agregar paréntesis para mantener la semántica. El siguiente ejemplo muestra uno de estos casos.

Antes del refactoring

```
aSender

      ^3 * self computeValue
```

```
computeValue

      ^5 + 10
```

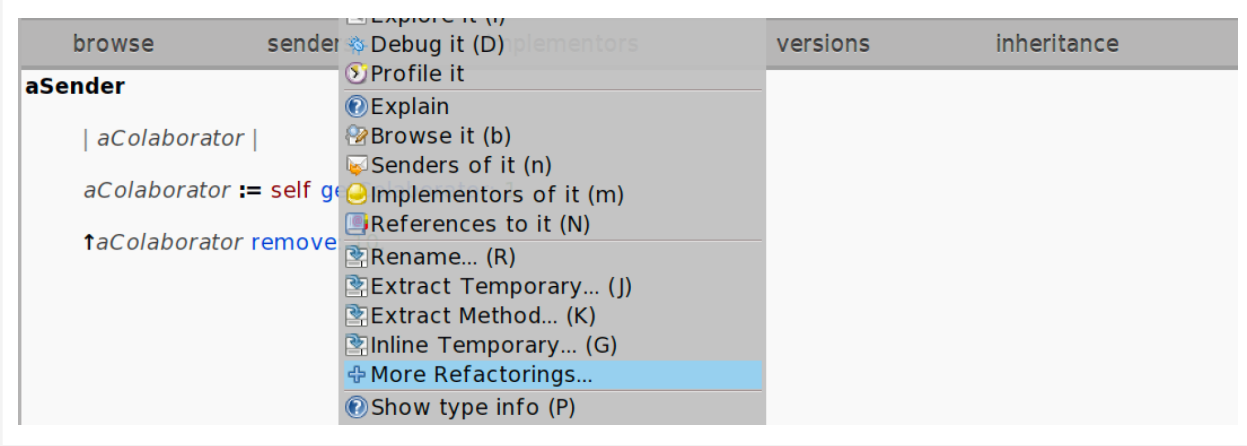
Después del refactoring
aSender ^3 * (5 + 10)

Fig. 20 - Ejemplo 6 de *Inline Method*

La implementación provista permite al usuario refactorizar más de una colaboración en una sola ejecución. Para esto, el programador deberá seleccionar a través de la interfaz gráfica todos los envíos del mensaje que quiere reemplazar y qué *implementor* utilizar.

3.2.2. Interfaz gráfica

Este refactoring es más complejo que el *Inline Temporary Variable*, y provee al programador varias opciones de configuración a la hora de ejecutarlo. Para empezar, existen dos flujos ligeramente distintos. El primero puede iniciarse accediendo al menú de refactorings directamente desde un envío del mensaje a refactorizar utilizando el click derecho sobre el mismo y presionando la opción "*More refactorings*" seguido de "*Inline method*", como muestra la Figura 21, o utilizando el shortcut *Ctrl+3* en *Windows* o *GNU/Linux*, o *Command+3* en *MacOS*.



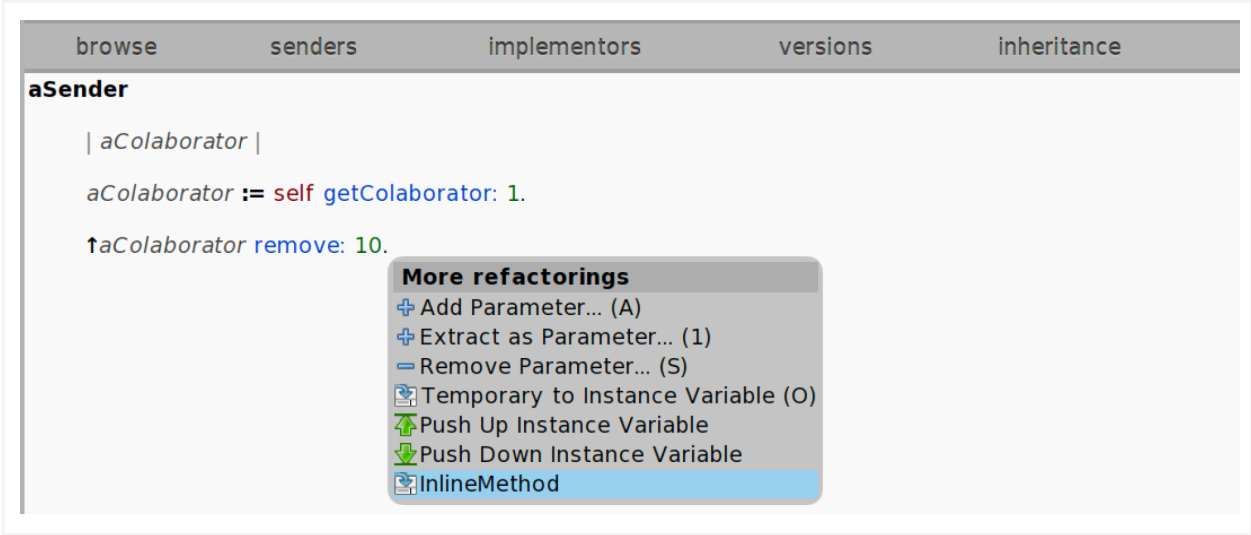


Fig. 21 - Menú desde el cual se inicia el refactoring *Inline Method* en el editor de código

A continuación, se abrirán una serie de menús que permiten configurar ciertos aspectos del refactoring. La implementación provista permite al usuario decidir si realizarlo únicamente sobre la colaboración desde la cual fue inicializado, o si prefiere ejecutarlo en varios lugares, en cuyo caso deberá elegir en cuáles. También se ofrece la opción de mantener o eliminar el método que será inlineado. Las siguiente figuras muestran las ventanas correspondientes a estas configuraciones.

Fig. 22 - Menú que permite al usuario elegir si refactorizar sólo una colaboración o varias

Fig. 23 - Menú que permite al usuario elegir si eliminar o no el *implementor* al finalizar el inline

Al tratarse *Smalltalk* de un lenguaje tipado dinámicamente, en principio no existe forma de saber exactamente cuál es el método que el programador quiere refactorizar. Por ejemplo, en la Figura 21, el tipo de `aColaborator` es desconocido fuera del runtime. Si existe más de una clase que defina el mensaje `remove:`, entonces no es posible determinar con certeza a qué método se refiere esa colaboración. Por esta razón, el programador es el responsable de seleccionar qué clase contiene la implementación que quiere utilizar en el reemplazo. A partir de aquí, el o los métodos donde se envía el mensaje sobre el cual se hace el inline serán llamados *senders*, mientras que el método elegido por el usuario será el *implementor*. Vale la pena destacar que el refactoring puede modificar múltiples *senders* al mismo tiempo a partir de la selección de un único *implementor*.

Para saber qué opciones de *implementors* y *senders* mostrar en la interfaz, primero el usuario debe seleccionar el scope del refactoring, como muestra la Figura 24. Por ejemplo, al seleccionar la opción "*In Class*", se buscará el mensaje en la clase actual y todos los envíos del mismo que se encuentren en métodos de dicha clase.

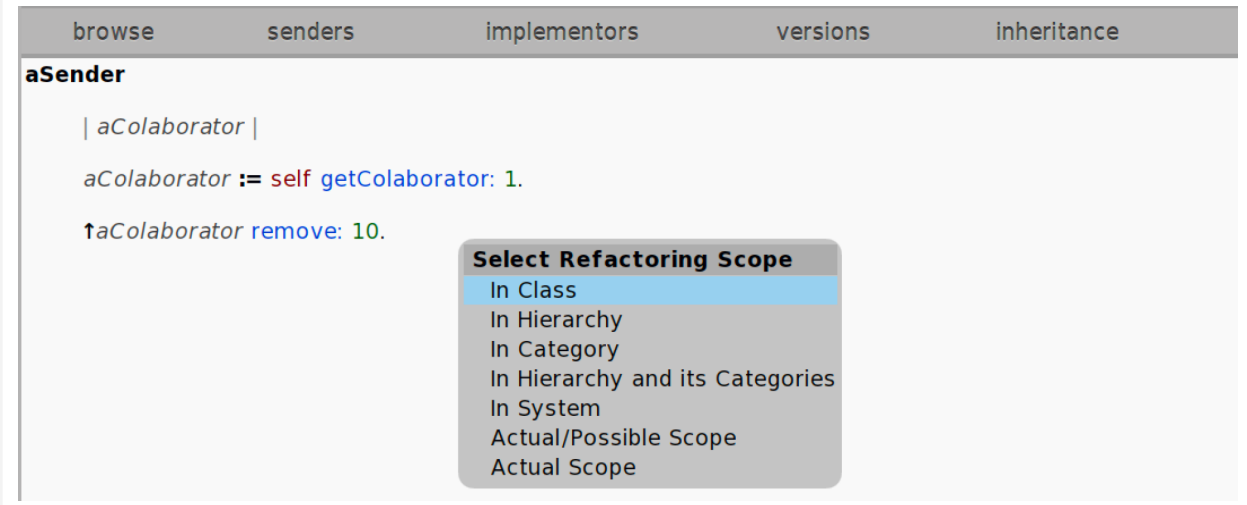


Fig. 24 - Menú que permite al usuario elegir el scope donde buscar *implementors* y *senders*

Las últimas dos opciones utilizan el sistema de *LiveTyping* incluído en *Cuis University* para presentarle al usuario opciones más precisas, dado que se utiliza la información de tipo del recibidor para identificar exactamente a qué *implementors* puede referirse esa colaboración, y qué otros envíos de mensaje pueden estar relacionados. Esta funcionalidad se trata con más detalle en la sección *3.2.4*.

Luego de elegir si cambiar o no más de un envío de mensaje, si no existen clases que respondan al mensaje a inlinear, el refactoring muestra un mensaje de error y finaliza sin hacer cambios. Si por el contrario las hay, se abrirá una ventana similar a la Figura 25 donde el programador deberá elegir qué *implementor* utilizar (en base al scope previamente seleccionado).

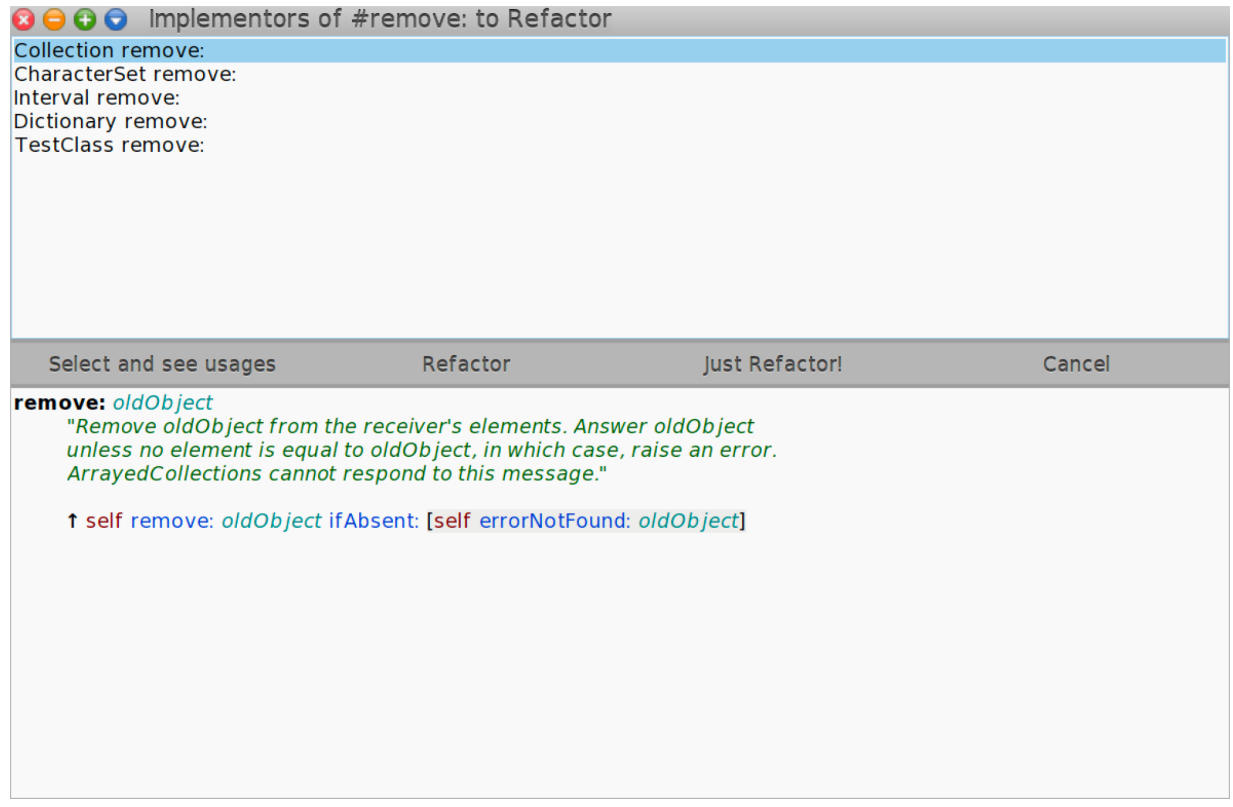


Fig. 25 - Ventana de selección del *implementor*

La ventana se divide en dos secciones separadas por una fila de botones. La sección superior lista los posibles *implementors* para el mensaje elegido. La sección inferior muestra el código fuente del método seleccionado actualmente en la parte superior. Los botones proveen las siguientes funciones:

- **Select and see usages**: selecciona el *implementor* actualmente resaltado y abre la ventana de selección de envíos de mensaje (Figura 26).

- **Refactor**: selecciona el *implementor* actualmente resaltado y ejecuta el refactoring sobre todos los posibles *senders* directamente, sin pasar por la selección de envíos de mensaje. Al finalizar, muestra una ventana de resumen con todos los cambios realizados. Esta opción es útil cuando el programador está seguro de cuáles son los envíos de mensaje posibles y quiere refactorizarlos todos.
- **Just refactor!**: igual que el anterior, pero no muestra los cambios al terminar.
- **Cancel**: cierra la ventana y cancela el refactoring.

Al presionar el botón "*Select and see usages*", se abrirá una ventana similar a la Figura 26, donde el usuario podrá elegir qué envíos del mensaje serán incluídos en el refactoring. Similar a la interfaz previa, en la parte superior se muestran los envíos presentes en cada clase en el formato `Clase>>#selector{messageNode}(intervalo en el código fuente)`. En la sección inferior se muestra el código asociado y la colaboración a refactorizar resaltada.

Por default, se incluyen todos los que están en la lista, y se utiliza el botón "*Remove*" para excluirlos. También se incluye la opción de volver a la pantalla anterior y cambiar el *implementor* mediante el botón "*See implementors*". El resto de los botones proveen la misma funcionalidad descrita anteriormente.

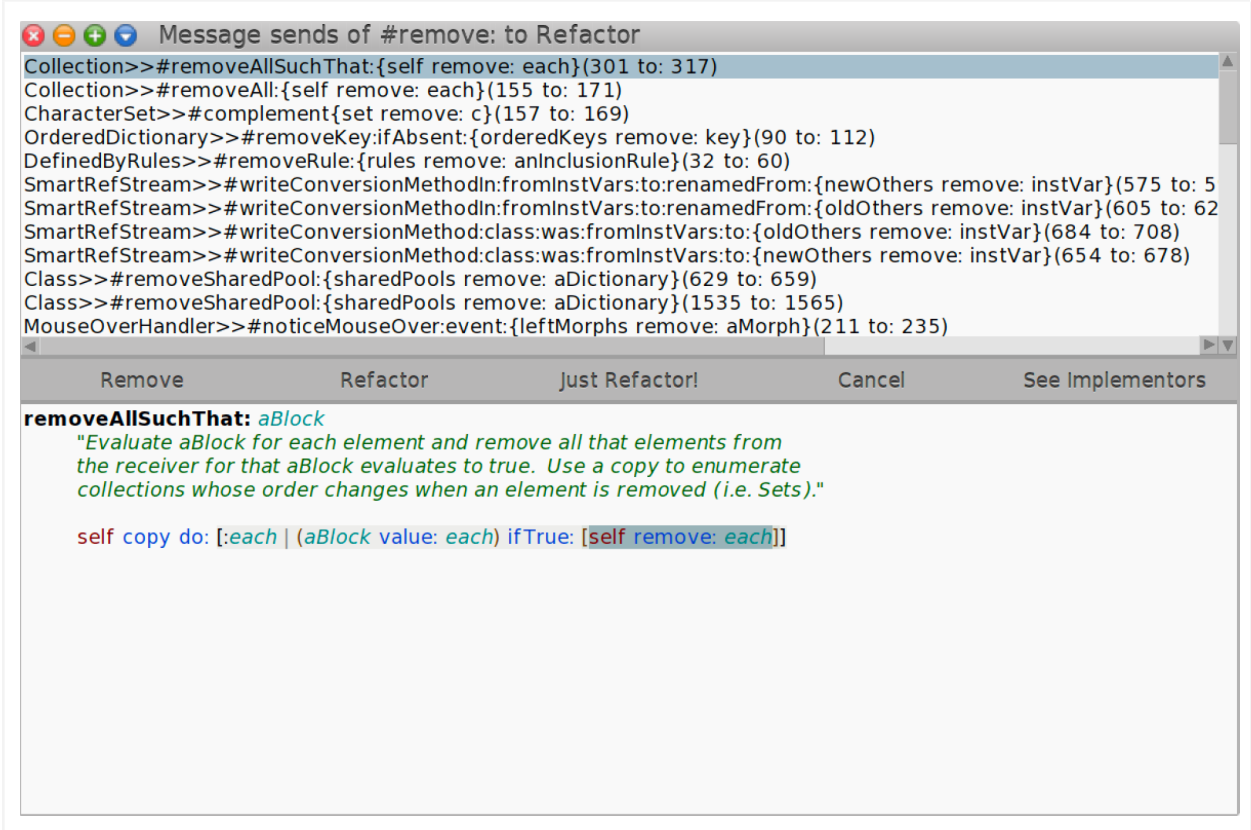


Fig. 26 - Ventana de selección de envíos de mensaje

Como se mencionó en secciones anteriores, la implementación no soporta refactorizar envíos de mensaje en cascada. Sin embargo, estos serán mostrados en la interfaz para que el usuario esté al tanto de que existen. Si alguno no es removido, el refactoring fallará con un mensaje de error tal como lo muestra la siguiente Figura.

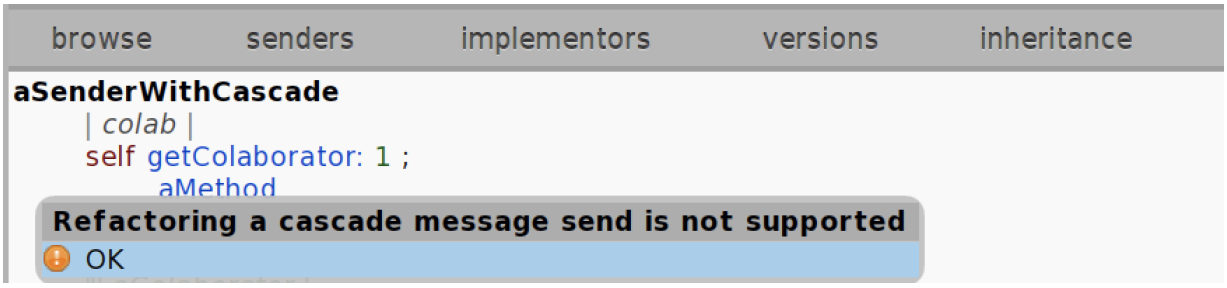


Fig. 27 - Mensaje de error al intentar refactorizar colaboraciones en cascada

Al utilizar el botón "*Refactor*", si todo sale exitosamente, se abrirá la ventana de cambios realizados, como muestra la Figura 28. En la sección superior se listan todos los selectores afectados, que pueden o bien haber sufrido cambios o haber sido borrados (este es el caso del *implementor* cuando el usuario elige la opción de eliminarlo al finalizar). En la sección inferior se muestra el resultado final de aplicar los cambios para el mensaje seleccionado actualmente.

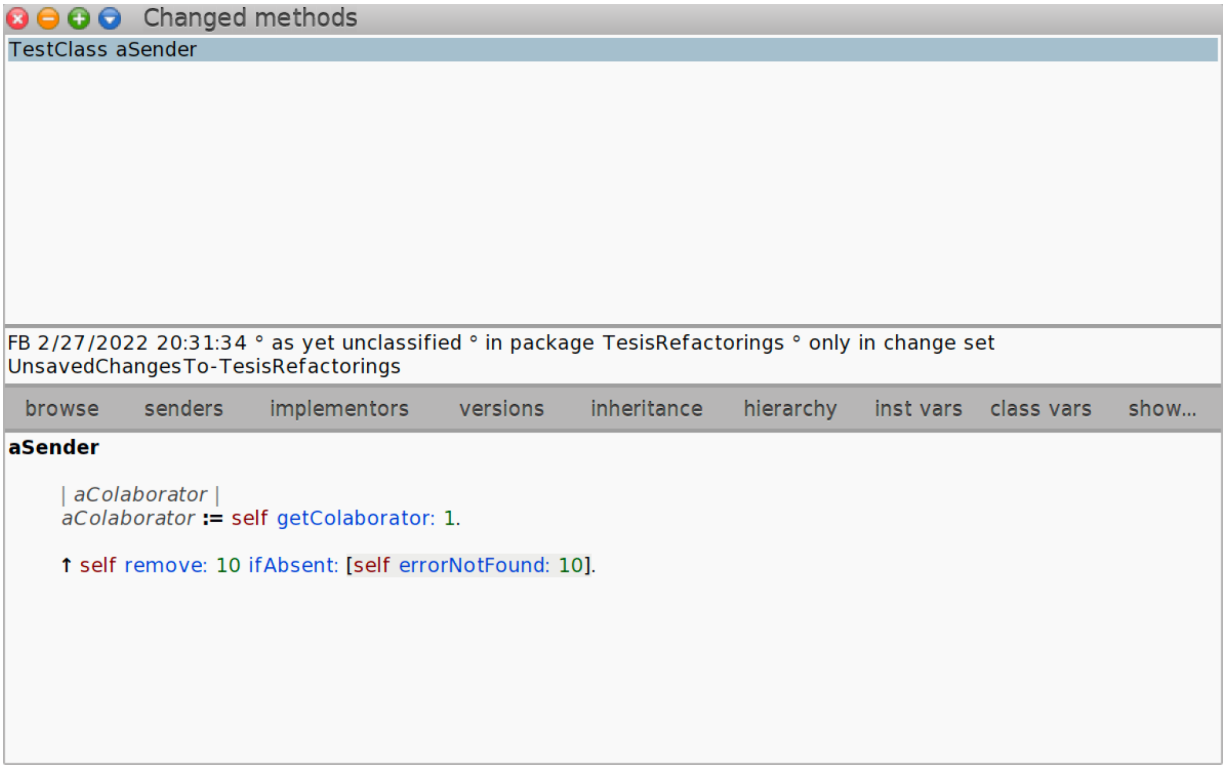


Fig. 28 - Ventana de cambios realizados

Como se mencionó anteriormente, el programador puede elegir refactorizar únicamente el envío de mensaje seleccionado al iniciar el refactoring (Figura 23). Si elige hacerlo, no es necesario mostrar la pantalla de selección de mensajes (Figura 26), por lo cual también cambian ligeramente los botones de la ventana de selección de *implementor* (Figura 25), que pasa a verse como la Figura 29.

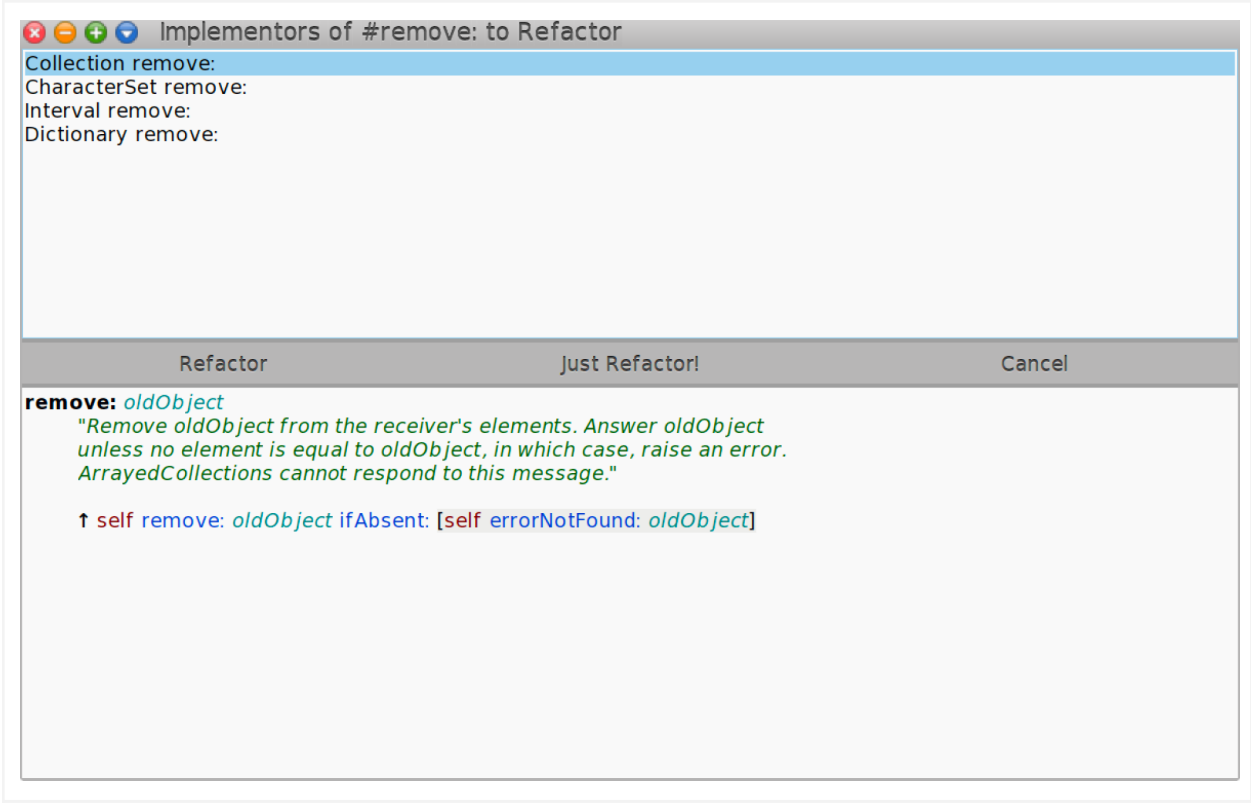


Fig. 29 - Ventana de selección de *implementors* sin selección de envíos de mensaje

Si al ejecutar el refactoring falla alguna validación o se produce un error, se le presentará un mensaje al usuario con una descripción del mismo. La siguiente Figura muestra un ejemplo de este caso, cuando se intenta hacerle inline a un método que accede a variables de instancia desde un sender que no tiene acceso a las mismas.

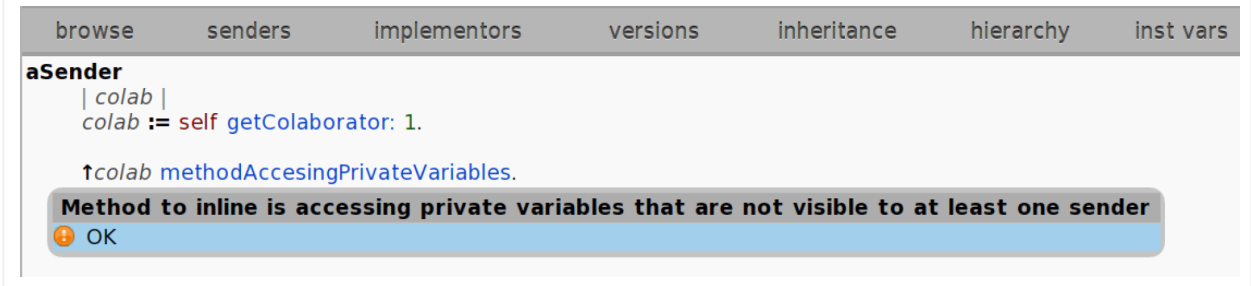


Fig. 30 - Ejemplo de mensaje de error mostrado al usuario cuando el refactoring falla

Como se indicó al principio de esta sección, existe también un segundo flujo para este refactoring, que puede ser iniciado desde dos lugares distintos con el mismo resultado. El primero es desde la ventana de selectores de una clase usando el click derecho y eligiendo la opción "*Refactorings*", seguida de "*Inline method*", como muestra la Figura 31. El segundo es desde el nombre del mensaje en el editor de código de la sección inferior (Figura 32)

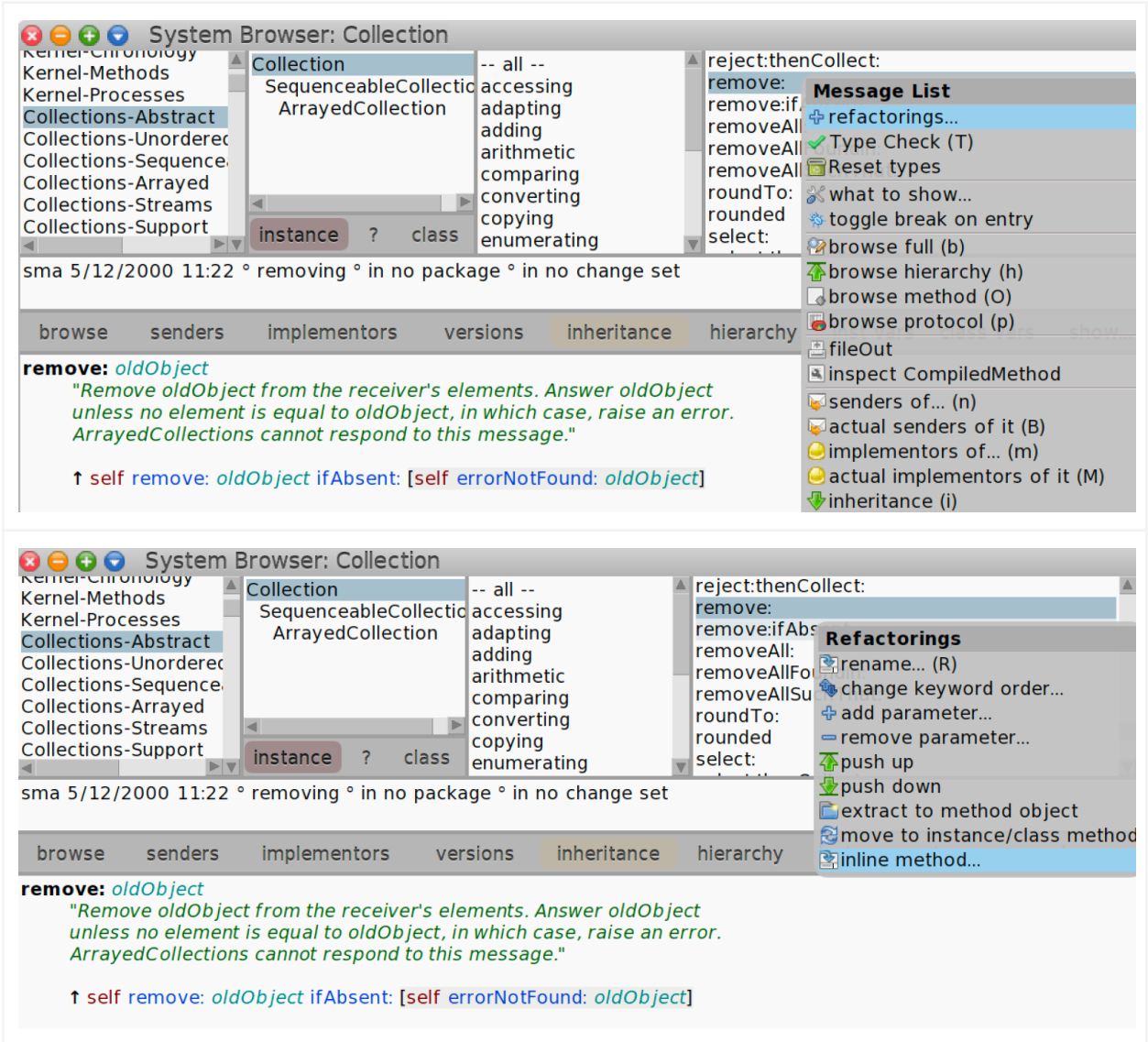


Fig. 31 - Menú desde el cual se inicia el refactoring *Inline Method* desde la lista de mensajes

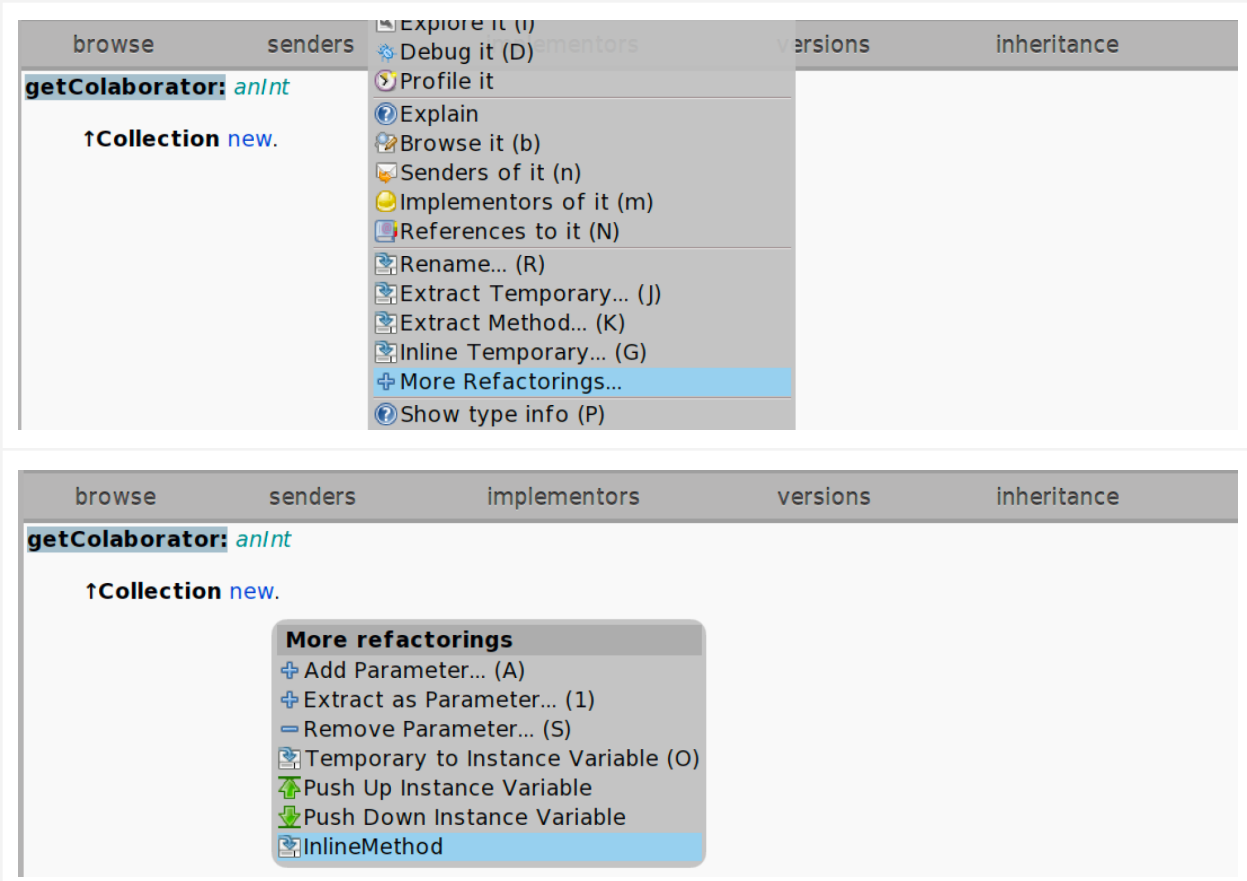


Fig. 32 - Menú desde el cual se inicia el refactoring *Inline Method* desde el nombre del mensaje en el editor de código

El flujo de configuración es bastante similar al que ya ha sido descrito. La diferencia más importante es que al iniciar el refactoring de alguna de estas formas, ya queda determinado qué clase es la que implementa el método a inlinear. Por esta razón, no es necesario presentarle al usuario la ventana de selección de *implementor*, y puede abrirse directamente el menú de envíos de mensaje. En este caso, este último tiene el mismo funcionamiento que el presentado en la Figura 26, con la diferencia de que no incluye el botón "*See implementors*", dado que no hace falta. La Figura 33 muestra un ejemplo de esta ventana al iniciar *Inline Method* sobre el mensaje `getColaborator:`.

Nótese que el menú de la Figura 22 tampoco tiene sentido en este flujo, por lo que el funcionamiento es equivalente a que el programador hubiera elegido la opción "*Select messages to inline".*

Fig. 33 - Ventana de selección de envíos de mensaje sin botón de *implementors*

Por último, también podría ocurrir que el usuario utilice el refactoring sobre un mensaje que no tiene *senders*. Como parte del flujo normal, el programador debe decidir si eliminar el método *implementor* o no del sistema. En este caso, si decide no hacerlo, ejecutar el refactoring no tiene efecto alguno, y por esta razón se le ofrece esta información al programador y se le ofrece nuevamente la opción de borrar el método. La siguiente figura muestra un ejemplo de esta situación, donde el mensaje `aSender` no tiene usos en el scope.

Fig. 34 - Opción mostrada cuando el mensaje no tiene *senders* y el programador ha decidido no eliminar el *implementor*

3.2.3. Implementación

La lógica principal de la implementación provista se encuentra distribuída entre las clases `InlineMethod` e `InlineMethodApplier`. La primera se encarga de computar y realizar los cambios necesarios a partir de un cierto *implementor* y una colección de envíos de mensaje, mientras que la segunda maneja la lógica de ventanas de configuración previas a la ejecución, y crea una instancia de `InlineMethod` en base a las opciones elegidas.

La implementación de este trabajo requiere que el código fuente cumpla con las siguientes precondiciones para poder ejecutarse:

1. Si el método *implementor* contiene referencias a variables de instancia, entonces todos los envíos de mensaje a refactorizar deben pertenecer a métodos de la misma clase. Esto es necesario porque el resto de las clases no tienen visibilidad de dichas variables, por lo cual no pueden ser referenciadas.
2. No está permitido usar *implementors* que tienen múltiples retornos (ya sean implícitos o explícitos). Como se verá en la sección *3.2.6*, mantener el comportamiento en estos casos puede volverse muy complejo y hasta las *IDEs* más utilizadas en la industria pueden fallar al intentarlo, por lo cual se decidió no contemplarlos en el scope de este trabajo.
3. No se puede utilizar el refactoring sobre un envío de mensaje en cascada. Dado que no es un uso muy frecuente, no fue implementado en esta primera iteración del refactoring, pero podría ser soportado en un futuro.
4. Si el método *implementor* referencia a la pseudovariable *self*, entonces cada envío de mensaje a refactorizar debe cumplir que o bien pertenece a un método en la misma clase que el *implementor*, o el receptor del mensaje es una variable. Si se cumple el primer caso, entonces *self* no necesita ser reemplazado. En el segundo, *self* se reemplaza con el nombre de dicha variable.
5. No está permitido usar *implementors* que referencien a la pseudovariable *super*. Esto es necesario dado que no hay forma de reemplazar la referencia por algo que mantenga la semántica del código.

Una de las partes más complejas de la implementación se encuentra relacionada con la ventana de configuración que permite al programador seleccionar qué envíos de mensaje deben ser refactorizados (Figura 26). Hasta el momento, los refactorings implementados en *Cuis University* solo necesitaban mostrar ventanas de selección de mensajes (la Figura 35 muestra un ejemplo), pero no de envíos de mensaje particulares. La mayoría de estos eran subclases de `ChangeSelectorApplier`.

Este sutil cambio implicó la necesidad de crear una nueva clase llamada `MessageNodeReference`, similar a la preexistente `MethodReference`, para representar un envío de mensaje particular, que incluyera propiedades tales como el método donde se encuentra y los *source ranges* del mismo. Esta nueva abstracción juega un rol central tanto en el applier como en el refactoring, dado que es el modelo sobre el cual se basa la ventana de selección de envíos de mensaje y la entidad utilizada a lo largo de todos los métodos internos de `InlineMethod`.

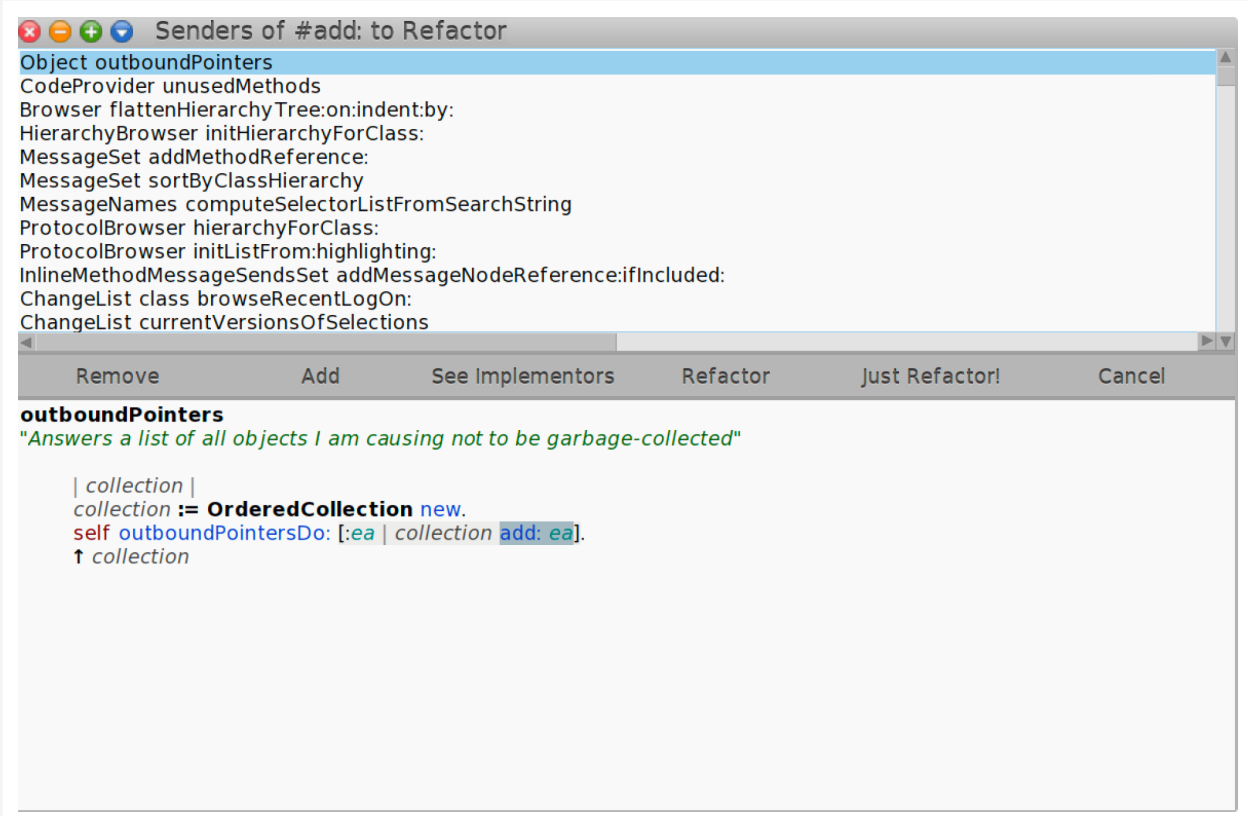


Fig. 35 - Ventana de selección de *senders* del refactoring *Rename Selector* (subclase de *ChangeSelector*)

El applier de *Inline Method* comparte mucha lógica relacionada a la selección de *implementors* y *senders* con `ChangeSelectorApplier`, pero se decidió que no heredaría de ésta dado que el refactoring no altera ningún selector. Debido a esto, en la primera implementación se duplicó mucho código y se deterioró la mantenibilidad del mismo, pero posteriormente se creó una superclase de `ChangeSelectorApplier` e `InlineMethodApplier` llamada `SelectableImplementorsAndSendersApplier`, que encapsula gran parte de la lógica compartida.

Otro aspecto similar entre ambos appliers es la búsqueda inicial de los *implementors* y *senders* en base al scope seleccionado por el programador. Previo al trabajo realizado, esta lógica se encontraba en la clase `ChangeSelector`. Esta funcionalidad fue trasladada a una nueva jerarquía de clases capaces de buscar *implementors*, *senders* (instancias de `MethodReference`) o envíos de mensaje (instancias de `MessageNodeReference`) de un selector en particular.

Existen dos formas de obtener obtener una instancia de `InlineMethodApplier`. La primera es utilizando el mensaje `InlineMethodApplier class>> initializeOn:for:in:`, que toma por parámetro un `Browser`, un `Symbol` con el nombre del selector a inlinear y la clase que lo implementa. Este constructor inicializa el flujo descrito donde el *implementor* ya es conocido. La otra forma es a través de `InlineMethodApplier class>> initializeOn:forMessageSend:`, que toma también un `Browser` y una instancia de

`MessageNodeReference`, que apunta al mensaje desde el cual se está instanciando el refactoring. Esta última es la utilizada por la clase `SmalltalkEditor` cuando se inicia el refactoring desde el editor de código. A continuación se presenta la declaración de la clase `InlineMethodApplier` y una tabla con la descripción de cada colaborador interno de la misma (la mayoría heredados de su superclase).

Definición de `InlineMethodApplier`

```
SelectableImplementorsAndSendersApplier subclass: #InlineMethodApplier
      instanceVariableNames: 'shouldRemoveImplementor triggeringMessageSend
shouldInlineTriggeringMessageSendOnly'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Tools-Refactoring'
```

Comentario de clase

```
I can apply the InlineMethod refactoring to a message. I am in charge of
handling the configuration windows and instantiating an instance of
InlineMethod.
Implementation notes:

- A single implementor will have to be chosen.
- More than one message send can be refactored at the same time.
- An option to delete the message from the implementor is provided.
```

Fig. 36 - Definición de la clase `InlineMethodApplier`

Variable de instancia	Tipo	Descripción
`selectorToRefactor`	`Symbol`	Nombre del selector que se desea refactorizar
`scopeChoice`	`SmallInteger`	Número que representa el scope seleccionado por el usuario
`implementors`	`Collection`	Colección de los implementors del mensaje `selectorToRefactor`. La ventana de selección de *implementors* modifica esta variable en base a las acciones del usuario

`senders`	`Collection`	Colección de los envíos posibles del mensaje `selectorToRefactor`. La ventana de selección de envíos de mensaje modifica esta variable en base a las acciones del usuario
`selectedClass`	`Class`	Dependiendo del flujo, esta variable puede ser la clase que contiene el mensaje a refactorizar, o la clase que contiene el envío de dicho mensaje
`wizardStepWindow`	Subclase de `InlineMethodWizardStepWindow`	Ventana actualmente abierta de selección de *implementors* o envíos de mensaje
`shouldShowChanges`	`Boolean`	Determina si mostrar o no el resumen de cambios al finalizar el refactoring
`browser`	`Browser`	Instancia de `Browser` desde la cual se inició el refactoring
`shouldRemoveImplementor`	`Boolean`	Determina si el mensaje refactorizado debería ser eliminado al finalizar
`triggeringMessageSend`	`MessageNodeReference`	Envío de mensaje desde el cual se inició el refactoring. Puede ser `nil` dependiendo del flujo
`shouldInlineTriggeringMessageSendOnly`	`Boolean`	Registro de si el usuario quiere refactorizar sólo el mensaje desde el cual se inició el refactoring o no

Fig. 37 - Tabla de variables de instancia de `InlineMethodApplier`. Todas salvo las últimas tres son heredadas

Para la correcta implementación de todas las ventanas de selección, fue necesario crear varias clases nuevas con lógica compartida. La superclase de todas ellas es `SelectableImplementorsAndSenderWizardStepWindow`, que contiene una referencia al applier asociado. La idea principal de las ventanas es modificar las variables de instancia `implementors` y `senders` de su `InlineMethodApplier` a medida que el usuario interactúa con ellas. A modo de ejemplo, al iniciar el refactoring, se cargan todos los *implementors* posibles en

la variable, y cuando el usuario selecciona uno, se guarda una colección con ese único elemento elegido, que después será pasado como parámetro a una instancia de `InlineMethod`.

Como fue descrito anteriormente, dependiendo del flujo, algunas ventanas son muy similares a otras pero con más o menos botones. Debido a algunas limitaciones técnicas, esto debió ser modelado utilizando jerarquías de clases, como muestra la siguiente figura.

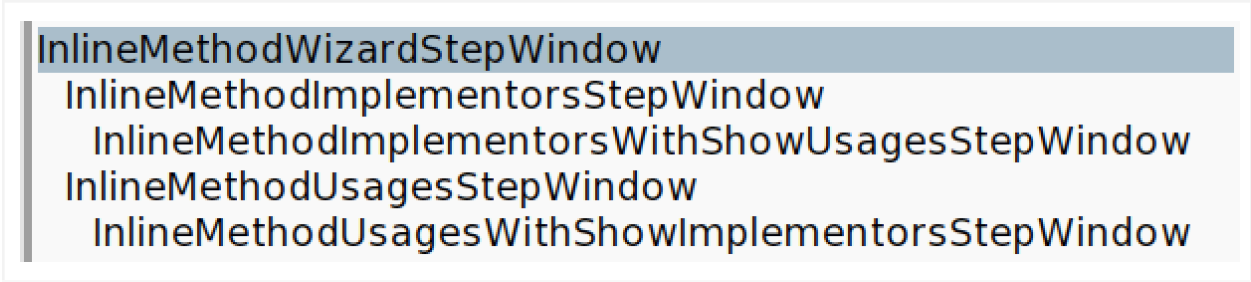


Fig. 38 - Jerarquía de clases responsables de la interfaz del refactoring *Inline Method*

El mensaje principal del applier es `#value`, y se encuentra definido en su superclase. Aquí se presenta el código fuente de la implementación provista para ilustrar partes relevantes de la lógica descrita previamente.

```
SelectableImplementorsAndSendersApplier>>value

	requestExitBlock := [ ^self ].

	self requestRefactoringParametersHandlingRefactoringExceptions.

	self
		ifHasNoSendersAndOneImplementor: [ :anImplementor |
			self
createAndApplyRefactoringWhenNoSendersAndOneImplementor: anImplementor
		]
		ifNot: [
			self askScope.
			self initializeImplementorsAndSenders.
			self openRefactoringWizard.
		]
```

Fig. 39 - Implementación de `InlineMethodApplier>>value`

A continuación se presenta la declaración de la clase `InlineMethod` y otra tabla con la descripción de sus colaboradores internos.

Definición de `InlineMethod`

```
Refactoring subclass: #InlineMethod
	instanceVariableNames: 'methodToInline messageSendsToInline
updatedSendersCode methodNodeToInline replacementsBySender
temporariesDeclarationsByNode temporariesToDeclareByInsertionPoint
implementorCompleteSourceRanges removeMethod'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Tools-Refactoring'
```

Comentario de clase

```
I am a refactoring that replaces message sends in the sender with the body
of the target method , replacing the parameters as needed.
Implementation notes:

- I can refactor multiple message sends at the same time.
- I can delete the target method.
```

Fig. 40 - Definición de la clase `InlineMethod`

Variable de instancia	Tipo	Descripción
`methodToInline`	`CompiledMethod`	Método que se desea refactorizar
`messageSendsToInline`	`Collection`	Colección de `MessageNodeReference` que se desean inlinear
`updatedSendersCode`	`Dictionary`	Diccionario que almacena el código actualizado luego de los cambios para cada método afectado por el refactor
`methodNodeToInline`	`MethodNode`	El `MethodNode` correspondiente a `methodToInline`
`replacementsByMessageSend`	`Dictionary`	Guarda los reemplazos de código que deben hacerse relacionados a cada envío de mensaje que se refactorizará

`temporariesDeclarationsByNode`	Dictionary	Contiene las variables temporales (preexistentes y agregadas por el refactoring si es necesario) para un cierto `MethodNode` o `BlockNode`
`temporariesToDeclareByInsertionPoint`	`Dictionary`	Guarda las variables temporales que hay que agregar para cada *sender* y en qué índice de su código fuente
`implementorCompleteSourceRanges`	`Dictionary`	Almacena los *complete source ranges* del método *implementor* para no recalcularlos repetidas veces
`removeMethod`	`Boolean`	Determina si el mensaje refactorizado debería ser eliminado al finalizar

Fig. 41 - Tabla de variables de instancia de `InlineMethod`

Dado que se trata de un refactoring complejo, existen varios pasos a seguir (algunos varían de acuerdo al flujo) y detalles de implementación. A modo ilustrativo, si se analiza uno de los casos de uso más normales, donde el refactoring se inicia desde el envío de un mensaje y su valor de retorno es almacenado en una variable local, pueden determinarse a grandes rasgos los siguientes pasos:

1. Para cada uno de los envíos que se desean refactorizar, aplicar los pasos 2-10.
2. Obtener el `CodeNode` más cercano que contiene al envío (también llamado *enclosing block*). Este puede ser un `BlockNode` si la colaboración se encuentra dentro de un *closure*, o el `MethodNode` del *sender*.
3. Obtener los *statements* del método *implementor*.
4. Guardar los reemplazos de parámetros necesarios para este envío de mensaje en la variable de instancia `replacementsByMessageSend`. El tipo de las entradas almacenadas es `Association` de `SourceRange` a `String`, donde el rango corresponde a un uso del parámetro en el método *implementor* y el string es el valor pasado como parámetro en la colaboración, en formato string. También se guardan los reemplazos de la pseudovariable *self* de ser necesarios. La Figura 42 muestra el fragmento de código relevante.
5. Almacenar en `temporariesDeclarationsByNode` las variables existentes y las que es necesario declarar en el *enclosing block*. Nótese que puede ser necesario renombrar variables si estas ya existen en el scope. También puede ocurrir que el implementor no utilice variables locales, en cuyo caso los pasos 5-7 no son necesarios.

6. Guardar el punto de inserción de estas nuevas variables para el *sender* siendo refactorizado.
7. Registrar en `replacementsByMessageSend` los reemplazos relacionados a variables locales, dado que pueden haber cambiado de nombre. Por ejemplo, en el *implementor* existe una referencia a la variable temporal `t`, pero en el nuevo scope esta se llama `t1`.
8. Para cada *statement* del *implementor*, realizar los reemplazos requeridos en base a `replacementsByMessageSend`.
9. Crear un *statement* nuevo donde se asigna el valor de retorno del *implementor* con los reemplazos del paso anterior a la variable que almacenaba el resultado del envío del mensaje al refactoring.
10. Guardar en `updatedSendersCode` una entrada que reemplace el *statement* del *sender* que asigna a la variable el resultado del envío del mensaje por todos los nuevos statements de los dos pasos anteriores.
11. Guardar en `updatedSendersCode` una entrada que declare todas las variables temporales necesarias para cada *sender*.
12. Opcionalmente, borrar el *implementor*.
13. A partir de `updatedSendersCode`, ensamblar el nuevo código fuente para sender y compilarlo.
14. Retornar la lista de mensajes afectados por el refactoring.

```
InlineMethod>>calculateReplacementsFrom: aMessageNodeReference usingRanges:
senderRanges

	"Return a Dict of (sourceRange -> string) where the source range
belongs to a parameter usage
	or self reference in the implementor method and the string is the
variable name passed as
	that used parameter in the message node, or the receiver of the
message to inline for self references
	E.g.

	m1: aParam

	^aParam

	---
	m2
		^m1:2

	then the Dict would have one entry: (<rangeOfAParamUsage> -> 2)"
	| replacements |
```

```
      replacements := self getReplacementsMapForMessageSend:
aMessageNodeReference.
      methodToInline methodNode arguments withIndexDo: [:anArgumentNode
:argIndex | | passedArgument |
            passedArgument := aMessageNodeReference messageNode arguments
at: argIndex.
            (self findRangesOf: anArgumentNode in:
implementorCompleteSourceRanges) do: [:aRange |
                  replacements at: aRange put: (self sourceCodeOfNode:
passedArgument ofSender: aMessageNodeReference compiledMethod using:
senderRanges).]

      ].

      methodToInline methodNode nodesDo: [:aParseNode | (aParseNode
isVariableNode and: [aParseNode referencesSelf]) ifTrue: [
            (self findRangesOf: aParseNode in:
implementorCompleteSourceRanges) do: [:rangeOfSelfReference |
                  "Given previous validations, we can be sure that the
receiver is a variable node"
                  replacements at: rangeOfSelfReference put:
aMessageNodeReference messageNode receiver name.
            ]
      ]].
      ^replacements.
```

Fig. 42 - Implementación del cómputo de reemplazos relacionados a parámetros

Como se mencionó anteriormente, varios de estos pasos dependen del caso de uso, pero dejan en claro la complejidad del código encapsulado en `InlineMethod`. Para entender por qué se tomaron algunas decisiones de diseño es necesario resaltar varias de las dificultades que presenta el problema.

Dado que el refactoring permite inlinear más de un envío de mensaje, podría ocurrir que en un mismo método haya que refactorizar dos o más colaboraciones. Esto hace que sea necesario acumular la información de todos los cambios para un cierto *sender* y construir el nuevo código fuente al final de procesar todos los envíos de mensaje. Por ejemplo, si el implementor utiliza una variable local `t`, y se refactorizan dos colaboraciones, es necesario declarar dos variables `t` y `t1` en el nuevo scope para que su valor no se pise. Esto no podría lograrse si al refactorizar el segundo envío de mensaje no se tuviera en cuenta que ya fue necesario crear una variable llamada `t` para el primero.

Además, debió determinarse un algoritmo que pudiera nombrar variables nuevas unívocamente. La opción elegida es la más sencilla y usada por cuestiones de legibilidad en

todas las implementaciones comerciales, y consiste en reutilizar el nombre con el que aparece en el método *implementor*. En caso de no ser posible porque ya se encuentra en uso, se le agrega un "1" al final. Si ese nombre tampoco se encuentra disponible, el número se va incrementando de a uno en uno hasta que se encuentre un nombre válido para la nueva variable. A modo de ejemplo, si tanto el *sender* como el *implementor* declaran la variable temporal `var`, se intentará nombrar la nueva variable en el *sender* como `var1`, luego `var2`, y así sucesivamente hasta encontrar un nombre libre.

Otra cara de la complejidad es determinar cuál es el caso de uso a refactorizar. Por ejemplo, determinar si el envío del mensaje es el valor de retorno implícito de un bloque, o si es la última línea un *full closure*, etc. Todas estas posibilidades resultan en pasos similares pero ligeramente distintos, como por ejemplo agregar o no un caracter de retorno o paréntesis a una expresión. La regla utilizada para este último punto es que se agregan paréntesis si el *return* del *implementor* es un `MessageNode` y el envío de mensaje refactorizado es parte de otro `MessageNode`, como muestra la Figura 20.

3.2.4. Integración con *LiveTyping*

Como fue mencionado brevemente en la sección *3.2.2*, este refactoring interactúa con el sistema de *LiveTyping* implementado en *Cuis University* para intentar mejorar la experiencia del usuario. En este caso, es utilizado en la implementación de las opciones "*Actual Scope"* y "*Actual/Possible Scope"* de la Figura 24. Estos *scopes* utilizan la información recopilada por *LiveTyping* para intentar identificar los posibles tipos del receptor de la colaboración sobre la cual se inicia el refactoring y presentarle al programador opciones de *implementors* y envíos de mensaje más precisas. Vale aclarar que dichos *scopes* sólo aparecen cuando el paquete de *LiveTyping* se encuentra instalado en la imagen, pero en caso de no estarlo, el resto del refactoring es capaz de seguir funcionando correctamente sin proveer esta funcionalidad, dado que no altera en absoluto la lógica descrita de la clase `InlineMethod`.

Debido a la implementación actual de *LiveTyping*, en algunos casos no se tiene información de tipado de algunas variables. Por ejemplo, todavía no se encuentra implementada la recolección de datos para las variables locales de un bloque. Además, el sistema recolecta los tipos en runtime, por lo que es necesario que el código que se desea refactorizar haya corrido al menos una vez para tener la información necesaria.

La diferencia entre las opciones "*Actual Scope"* y "*Actual/Possible Scope"* es que la segunda incluye opciones sobre las que no se tiene información, mientras que la primera las excluye. Vale aclarar que en la interfaz, al seleccionar *"Actual Scope"* también se muestran opciones del *"Possible Scope"* para que el usuario esté al tanto de que también existen, pero son ignoradas al proseguir con el refactoring. Si el usuario elige la primera opción y el sistema no tiene información sobre el receptor, se muestra un mensaje de error, como puede verse en la Figura 43.

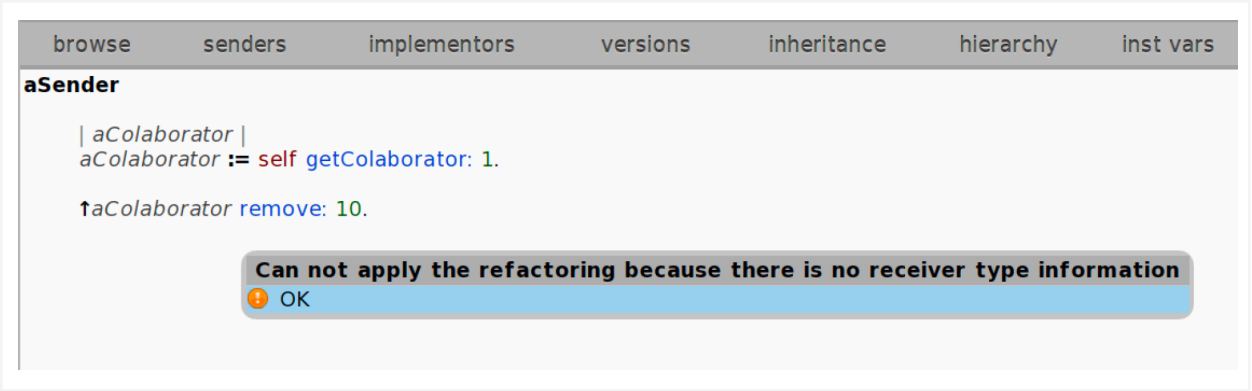


Fig. 43 - Mensaje de error al utilizar "*Actual Scope*" cuando no hay información de tipos

Para facilitar la explicación de esta funcionalidad, se presentan a continuación una serie de clases y mensajes que serán utilizados en ejemplos subsiguientes.

```
TestClass1>>sharedMethod

      ^'Method from TestClass1'.
```

```
TestClass2>>sharedMethod

      ^'Method from TestClass2'.
```

```
TestInline>>aSender
      | colab |
      colab := self getRandomColaborator.
      ^colab sharedMethod.
```

```
TestInline>>getRandomColaborator
      ^(Random new nextInteger: 10) even
          ifTrue: [^TestClass1 new]
          ifFalse: [^TestClass2 new].
```

Fig. 44 - Definición de clases para ejemplificar *LiveTyping* en la selección de *implementors* de `InlineMethod`

Si se asume que el código de `TestInline>>aSender` nunca fue corrido, entonces no hay información de tipo sobre la variable `colab`. Al correr por primera vez, se ejecuta `TestInline>>getRandomColaborator`, que puede devolver una instancia de `TestClass1` o de `TestClass2`. Si se inicia el refactoring sobre `sharedMethod` en `TestInline>>aSender` y se elige como *scope* alguna de las opciones de *LiveTyping*, sólo se presentará como opción de *implementor* la clase que haya devuelto `TestInline>>getRandomColaborator` al haber corrido. Con sucesivas ejecuciones, es muy probable que la clase retornada sea distinta

(debido a la aleatoriedad del método), por lo cual los posibles tipos de `colab` pasarán a ser `TestClass1` y `TestClass2`. Cuando eso ocurra, si se vuelve a ejecutar `InlineMethod` sobre `sharedMethod`, se mostrarán ambas clases como posibles *implementors*. El caso más frecuente y útil de esta funcionalidad se da cuando el receptor del mensaje siempre es del mismo tipo, pero existen muchas clases que implementan dicho mensaje. En esta situación, *LiveTyping* reduce la selección del *implementor* a una única opción posible.

Como se mencionó previamente, *LiveTyping* también es útil a la hora de elegir cuáles envíos de mensajes pueden ser refactorizados. Es posible distinguir dos tipos: los "seguros" son aquellos sobre los cuales se tiene certeza de que la colaboración se refiere a un selector determinado, y los "posibles", donde este podría ser el caso, pero no puede afirmarse. Se utilizarán las siguientes definiciones para explicar con más detalle la funcionalidad.

```
TestInline>>methodToInline

	^10.
```

```
TestInline>>sendToSelf

	^self methodToInline.
```

```
TestInline>>sendToUnknown

	^[ | a |

	a methodToInline. ] value.
```

```
TestInline>>sendToWrongType

	^1 methodToInline.
```

Fig. 45 - Definición de mensajes para ejemplificar *LiveTyping* en la selección de envíos de mensaje de `InlineMethod`

La siguiente tabla describe qué ocurriría al intentar iniciar el refactoring sobre los envíos del mensaje `methodToInline` de la Figura anterior y seleccionar la opción para elegir qué colaboraciones deben ser tenidas en cuenta. Nótese que en casos reales, podría ocurrir que un mismo método envíe dos o más veces el mismo mensaje, por lo cual podría haber más de un envío posible o seguro para un mismo método, y pueden ser seleccionados individualmente. Por cuestiones de legibilidad, la Figura 46 denota los envíos como referencias a métodos, pero en realidad son referencias a envíos del mensaje dentro de un método.

Mensaje *sender*	Descripción	Senders seguros	Senders posibles

`sendToSelf`	El mensaje se envía a self, por lo cual se detecta que el selector es `TestInline>>methodToInline`. *LiveTyping* calcula los envíos del mensaje seguros y posibles. El envío en `sendToWrongType` es ignorado porque `SmallInteger` no implementa `#methodToInline`.	`sendToSelf`	`sendToUnknown`
`sendToUnknown`	Falla porque no hay información de tipos de la variable de bloque `a`.	No hay.	No hay.
`sendToWrongType`	Se detecta que el mensaje no es implementado por el receptor (`SmallInteger`) y se le informa al programador. Sin embargo, se muestra el envío desde el cual se inició el refactoring entre los senders seguros.	`sendToWrongType`	`sendToUnknown`

Fig. 46 - Ejemplo de clasificación de *senders* seguros y posibles en `InlineMethod` con *LiveTyping*

La forma de interacción con el módulo es similar a la de otros refactorings implementados en *Cuis*, pero adaptada para funcionar con instancias de `MessageNodeReference` en vez de `MethodReference` en la ventana de selección de envíos de mensaje. Para facilitar el entendimiento de la implementación, se mencionó previamente que el applier era una instancia de `InlineMethodApplier`. En realidad, cuando se utiliza *LiveTyping*, se obtiene una instancia de `InlineMethodApplierWithActualScope` y a partir de esta una de `InlineMethod`. La primera es una subclase de `InlineMethodApplier` cuya única función es agregar las dos opciones de scope ya mencionadas e instanciar el refactoring.

3.2.5. Testing

De forma similar al *Inline Temporary Variable*, la implementación de *Inline Method* también fue llevada a cabo utilizando *TDD*. En este caso, se escribieron un total de 32 tests, de los cuales 26 son de casos exitosos (donde el refactoring puede ser aplicado) y 6 de casos de error, donde no se cumplen las precondiciones. A continuación se lista el propósito de cada uno de los tests de casos exitosos, en el orden en el que fueron implementados. Nótese también

que cada bug descubierto a lo largo de la implementación llevó a la creación de un nuevo caso de test que lo cubriera.

1. Hacer inline de un método sin parámetros y que no tiene valor de retorno explícito.
2. Hacer inline de un método con parámetros y que no tiene valor de retorno explícito.
3. Hacer inline de un método que solamente retorna un literal.
4. Hacer inline de un método con varios statements y un valor de retorno explícito que es asignado en el *sender* a una variable.
5. Hacer inline de un método cuyo valor de retorno es ignorado por el *sender*.
6. Hacer inline de un método que declara variables temporales.
7. Hacer inline de un método que declara variables temporales con el mismo nombre que variables ya utilizadas en el *sender*.
8. Al hacer inline de un método que utiliza variables temporales dentro de un bloque del *sender*, las mismas son declaradas en el scope del *closure* y no del método.
9. Hacer inline de un método que es el retorno de un *full closure* mantiene el símbolo de retorno.
10. Hacer inline de un método de instancia de otra clase.
11. Hacer inline de un método de clase de otra clase.
12. Hacer inline sobre dos envíos del mensaje en un mismo *sender*.
13. Hacer inline de un método y eliminar el mensaje del sistema.
14. Hacer inline de un método que tiene una sola línea y retorna un literal que es ignorado.
15. Hacer inline de un método que se encuentra en la última línea de un *closure*.
16. Hacer inline de un método mantiene la indentación correctamente en algunos casos particulares.
17. Hacer inline de un método que es parte de un statement de retorno del *sender*.
18. Hacer inline de un método agrega paréntesis de ser necesario.
19. Hacer inline de un método en varios *senders*.
20. Hacer inline de un método que accede a variables privadas en un *sender* de la misma clase.
21. Hacer inline de un método de otra clase que referencia a la pseudovariable *self* cuando el receptor del envío del mensaje es una variable.
22. Hacer inline de un método de la misma clase con un valor de retorno explícito cuando el *sender* retorna *self* implícitamente.
23. Hacer inline de un método que tiene una única línea donde se retorna el resultado de otro método, que no es asignado en el *sender*.
24. Hacer inline de un método cuyo valor de retorno es utilizado en la inicialización de un array.
25. Hacer inline de un método cuyo valor de retorno es utilizado en la inicialización de un array y requiere paréntesis.

26. Hacer inline de un método que termina en el caracter de finalización de un *statement* (el ".") y cuyo resultado es usado como receptor de un mensaje en el *sender*.

Los casos de uso de falla verifican que las precondiciones deben cumplirse para ejecutar el refactoring:

1. No se puede hacer inline de un método que accede a variables privadas si alguno de los senders no tiene acceso a las mismas (por no pertenecer a la misma clase que el *implementor*).
2. No se puede hacer inline de un método que tiene múltiples valores de retorno posibles.
3. No se puede hacer inline de un método que tiene un valor de retorno explícito y uno implícito.
4. No es posible refactorizar una colaboración en cascada.
5. No se puede refactorizar un método que referencia a la pseudovariable *super*.
6. No se puede refactorizar un método que referencia a la pseudovariable *self* cuando el receptor del envío del mensaje no es una variable.

3.2.6. Comparación entre entornos de *Inline Method*

La Figura 49 presenta un listado de casos de uso y cómo se comporta el refactoring en cada distribución analizada. El uso más común es refactorizar un mensaje propio de la clase, y está cubierto por todas las implementaciones. Sin embargo, una de las diferencias más importantes es que *Cuis* permite refactorizar colaboraciones donde el receptor no es *self*, dando la opción de seleccionar el *implementor* a utilizar.

Otra de las ventajas de la implementación provista es que le permite al programador elegir exactamente cuáles colaboraciones deben ser refactorizadas, a través de todos los *senders* posibles en el scope seleccionado. En este aspecto, la funcionalidad de *Pharo* es poco intuitiva, dado que la selección de colaboraciones dentro de un mismo método funciona de forma incremental. A modo de ejemplo, si hay dos envíos del mensaje dentro del mismo método, se puede elegir refactorizar sólo el primero, o tanto el primero como el segundo, pero no sólo el segundo. No está claro si esto es intencional o si se trata de un error de implementación.

Otra de las diferencias relevantes es la posibilidad de eliminar el método refactorizado del sistema al concluir el refactoring. Tanto *Pharo* como *Squeak* solo permiten esta opción al iniciarlo desde la lista de mensajes de la clase, mientras que *Cuis* permite hacerlo también desde el editor de código. Además, *Squeak* no le permite al usuario elegir, sino que lo hace automáticamente.

Una de las limitaciones de *Cuis* es que no es posible hacer inline sobre un método que contiene múltiples valores de retorno. Por su parte, tanto *Pharo* como *Squeak* funcionan de la misma forma, siendo capaces de reescribir el código en algunos casos para poder ejecutar exitosamente el refactoring. La Figura 47 muestra un ejemplo de este caso de uso. Cuando un fragmento de código más complejo como el de la Figura 48 no puede ser reescrito, ambas implementaciones lanzan una advertencia y no permiten continuar con el refactoring.

Antes del refactoring

```
aSender
      | b |
      b := self aMethodToInline.
      ^ 50
```

```
aMethodToInline
      1 > 0 ifTrue: [^true]
            ifFalse: [^false].
```

Después del refactoring sobre *m1*

```
aSender
      | b |
      b := 1 > 0
            ifTrue: [ true ]
            ifFalse: [ false ].
      ^ 50
```

Fig. 47 - Funcionamiento de Pharo y Squeak de *Inline Method* en un caso donde el método tiene múltiples valores de retorno

Antes del refactoring

```
aSender
      | b |
      b := self aMethodToInline.
      ^ 50
```

```
aMethodToInline

      1 > 0 ifTrue: [^true].
      self anotherMethod ifTrue: [^self
anotherMethod] ifFalse:[^true].
```

Fig. 48 - Ejemplo de método con múltiples retornos no refactorizable con *Inline Method* en Pharo ni Squeak

Por último, la implementación de *Cuis* no soporta actualmente la refactorización de colaboraciones en cascada, mientras que esto es soportado total y parcialmente por *Pharo* y *Squeak* respectivamente. Se tomó la decisión de no incluir este feature en el presente trabajo dado que no es un caso de uso frecuente.

Caso de uso	*Pharo(9.0)*	*Squeak(5.3)*	*Cuis University(v5093)*
Iniciar el refactoring desde el editor de código	Soportado	Soportado	Soportado
Iniciar el refactoring	Soportado	Soportado	Soportado

desde la lista de mensajes de la clase			
Refactorizar mensajes con parámetros	Soportado	Soportado	Soportado
Refactorizar un método que declara variables temporales ya utilizadas en el *sender*	Soportado	Soportado	Soportado
Refactorizar más de una colaboración desde el editor de código	No soportado	No soportado	Soportado
Refactorizar más de una colaboración desde la lista de mensajes de la clase	Parcialmente soportado. La selección de colaboraciones funciona de forma incremental dentro de un mismo *sender*	Parcialmente soportado. Se aplica a todas las colaboraciones, no se puede elegir a cuáles	Soportado. Permite elegir exactamente cuáles refactorizar
Refactorizar una colaboración donde el receptor no es *self*	No soportado	No soportado	Soportado
Refactorizar una colaboración donde el *implementor* tiene múltiples valores de retorno posibles	Parcialmente soportado. En algunos casos puede reescribirse el código	Parcialmente soportado. En algunos casos puede reescribirse el código	No soportado
Refactorizar un mensaje teniendo el cursor sobre alguna parte del método	Soportado	No soportado. Es necesario seleccionar la expresión entera	Soportado
Eliminar el método *implementor* al finalizar el refactoring	Parcialmente soportado. Permite elegir si hacerlo o no sólo al hacerlo desde la lista de mensajes	Parcialmente soportado. Al hacerlo desde la lista de mensajes, se fuerza el borrado si no hay más *senders* en la clase	Soportado. Permite elegir si hacerlo o no tanto desde el editor como desde la lista de mensajes
Refactorizar una colaboración en	Soportado	Parcialmente soportado. Sólo se	No soportado. Se muestra un mensaje

cascada		puede sobre el último mensaje	de error

Fig. 49 - Comparación de casos de uso de *Inline Method* entre distintas distribuciones de Smalltalk

4. Conclusiones y trabajo futuro

4.1. Conclusiones

Los objetivos de este trabajo fueron cumplidos satisfactoriamente. Ambos refactorings implementados funcionan lo suficientemente bien como para poder ser utilizados frecuentemente por los usuarios de *Cuis University*. Al ya proveer esta plataforma un esquema de organización de cómo deben implementarse refactorings automáticos, fue relativamente sencillo integrar la nueva funcionalidad. Gracias al trabajo previo sobre esta temática, no fue necesario modificar clases relacionadas al parser, lo cual simplificó mucho la tarea.

Uno de los desafíos más interesantes a la hora de implementar refactorings es la de analizar si es posible realizarlo o no. En lenguajes de tipado dinámico como *Smalltalk*, la dificultad es mucho mayor porque no se conoce el tipo del receptor del mensaje. Las implementaciones provistas en el presente trabajo lograron integrarse con el componente de *LiveTyping* de *Cuis* para poder proveer al programador una mejor experiencia y mayor flexibilidad con respecto al estándar de otros entornos.

Como adiciones relevantes a la distribución pueden mencionarse la creación de una nueva abstracción para representar el envío de un mensaje en un método en particular y con información sobre los *source ranges* llamada `MessageNodeReference`. Además, se incluyeron mejoras en otras clases ya presentes del sistema tales como `SourceCodeInterval`.

Con respecto a la metodología, fue posible implementar la totalidad del código de `InlineTemporaryVariable` e `InlineMethod` usando la estrategia de *TDD*. Eso permitió tener un nivel muy alto de seguridad en cada paso de la implementación y ayudó a descubrir problemas que surgían en el paso de refactorización de la solución. Por otro lado, si bien al principio fue necesario incurrir en un costo de tiempo de *setup* de los tests, una vez que eso fue resuelto, escribir uno nuevo se convirtió en una tarea fácil y rápida. Vale aclarar que el código relacionado a los appliers y a la interfaz gráfica no tiene tests automáticos, sino que fue probado manualmente.

Las primeras versiones del código han sido integradas a modo de prueba en algunas versiones de *Cuis* para ser distribuídas a partes de la comunidad para poder obtener *feedback* sobre los refactorings. Los mismos fueron utilizados durante el primer cuatrimestre de 2022 en la materia "Ingeniería de Software 1" de la Universidad de Buenos Aires, y de esta forma se descubrieron algunos casos no contemplados por los tests que ya han podido ser arreglados en las últimas versiones.

4.2. Trabajo futuro

Como fue mencionado anteriormente, una de las desventajas de que los refactorings trabajen directamente manipulando strings es que el formateo del código fuente queda a cargo de la implementación del refactoring. Esto se evidencia en algunos casos al invocar cualquiera de los implementados en este trabajo. Una posible mejora para ambos sería asegurarse de que los niveles de indentación se mantienen.

4.2.1. Mejoras a *Inline Temporary Variable*

La implementación provista de este refactoring ya cumple con la mayoría de los features que suelen incluir las *IDEs* comerciales y es más versátil que las de *Pharo* y *Squeak*. Algunas otras mejoras posibles relacionadas a este son:

1. Permitirle al usuario elegir exactamente en qué referencias hacer el inline, conservando la variable en caso de no refactorizar todas. Vale la pena notar que en algunos casos esta funcionalidad podría cambiar la semántica del código, por lo cual ya no podría considerarse un refactoring propiamente dicho.
2. Detectar casos particulares donde el refactoring puede cambiar la semántica y lanzar una advertencia. La Figura 50 muestra uno de ellos, donde todas las implementaciones analizadas (incluso las comerciales) cambian el resultado del código al aplicar el inline sobre `aCollection`.

Antes del refactoring

```
aSender

	| aCollection |

	aCollection := Collection new.
	aCollection add: 3.
	^aCollection. "Returns a collection with the number 3"
```

Después del refactoring

```
aSender

	Collection new add: 3.
	^Collection new. "Returns an empty collection"
```

Fig. 50 - Ejemplo donde *Inline Temporary Variable* cambia la semántica del código

4.2.2. Mejoras a *Inline Method*

Al comparar la implementación de *Cuis* con las de *Pharo* y *Squeak*, queda en claro que la desventaja más grande es que no está soportado hacer inline de un método que tiene múltiples valores de retorno (implícitos o explícitos). Una mejora posible sería realizar un análisis del *AST* en estos casos y reescribirlos de ser posible, de forma similar a lo que hacen las otras distribuciones.

Otra de las limitaciones es que no está permitido ejecutar el refactoring sobre colaboraciones en cascada. Dada la estructuración del código y la batería de tests sobre el funcionamiento del mismo, implementar esta funcionalidad no debería ser muy complejo, por lo cual podría ser añadida en versiones futuras.

5. Referencias

[1] Aho, Alfred V., and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*. Prentice Hall, Inc., 1972.

[2] Beck, Kent. *Test Driven Development*. Addison-Wesley, 2003.

[3] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019.

[4] Garbezza, Nahuel. *Mejorando el ambiente de programación Cuis Smalltalk con refactorings esenciales*. Disponible en https://docs.google.com/document/d/1PGccW77lAUdPZBONx4MZ8lRB7aZHskbzDVtqLf3pTZo/edit#heading=h.2q6aa5sdiv6u.

[5] Goldberg, Adele, and David Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley, 1983.

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

[7] Wilkinson, Hernán. *VM Support for Live Typing: Automatic Type Annotation for Dynamically Typed Languages*. 2019.