Departamento de Ciencia y Tecnología
Licenciatura en Informática
Seminario Final

“Mejorando el ambiente de programación Cuis Smalltalk con refactorings esenciales”

Nahuel Garbezza

Director: Lic. Máximo Prieto
Codirector: Lic. Hernán Wilkinson (UBA)

A Rocío, mi compañera de vida, por acompañarme en este proceso y brindarme el apoyo incondicional que necesité en esos momentos difíciles.

A mis directores, Máximo y Hernán, por su constante guía, su interés genuino por este trabajo, inmensa sabiduría en la temática y predisposición para resolver consultas.

Al grupo de tesistas con el que compartimos este largo camino que implican los trabajos finales de carrera, en especial a Matías y Fernando que recorrieron este camino antes que yo, y me mostraron que era posible. ¡Mejor acompañado que solo!

A Hernán (sí, ¡otra vez!) y a 10Pines, mi hogar en lo profesional, por el tiempo y espacio cedido para nuestras reuniones de seguimiento, claves para avanzar a paso firme.

A colegas y alumnas/os de UNQ y UCA, a quienes les he mostrado mi progreso y compartido las versiones preliminares de mis desarrollos, y así obtener feedback muy valioso.

A Juan Vuletich, por liderar Cuis Smalltalk, un proyecto argentino y de código abierto. Un ejemplo de buen software, sin duda el software más bonito con el que he trabajado, y en el que que me da un enorme placer contribuir a través de este trabajo.

A la Comunidad de Programación Informática de la UNQ, de la que soy muy afortunado de pertenecer desde sus inicios y me brindó muchísimo, más de lo que he podido devolverle.

A ustedes, ¡¡MUCHAS GRACIAS!!

Tabla de contenidos

1. Introducción 5

2. Contexto, problemas y motivaciones 6
    2.1. Definiciones 6
        2.1.1. Smalltalk 6
        2.1.2. Refactorings 7
        2.1.3. Parseo y generación de código 9
        2.1.4. Elementos de Cuis 10
        2.1.5. Metodología 12
    2.2. Estado del arte: Refactorings existentes en diferentes distribuciones de Smalltalk 12
        2.2.1. Comparativa de *Extract Method* 13
        2.2.2. Comparativa de *Extract Variable* 16
        2.2.3. Resumen de la comparativa 18
    2.3. Estado del arte: Refactorings en Cuis Smalltalk 19
    2.4. Objetivos 19

3. Refactoring: *Extract Method* 21
    3.1. Funcionalidad 21
    3.2. Interfaz de usuario 22
    3.3. Implementación 26
    3.4. Testeo 31

4. Refactoring: *Extract Variable* 35
    4.1. Funcionalidad 35
    4.2. Interfaz de usuario 35
    4.3. Implementación 38
    4.4. Testeo 42

5. Cambios y mejoras necesarias para la implementación de refactorings 45
    5.1. Modelo de refactorings 45
        5.1.1. Comentarios de clases 45
        5.1.2. Nuevas precondiciones 45
    5.2. Agregados a Cuis Smalltalk 46
        5.2.1. *Source ranges* completos 46
        5.2.2. *Source ranges* para *brace arrays* 47
        5.2.3. *Source ranges* para mensajes en cascada 47
        5.2.4. Nodos nuevos del AST para declaración de variables temporales, con sus respectivos *source ranges* 48
        5.2.5. Reificación de intervalos de código 50
        5.2.6. Cambios y mejoras auxiliares 50

6. Metodología y conceptos aplicados 51

6.1. Forma de trabajo 51
6.2. Publicación 53
6.2.1. Versión beta o no oficial 53
6.2.2. Versión final para usuarios 53
6.2.3. Licencia 53
6.3. Conceptos aplicados 53
6.3.1. Patrones y buenas prácticas de diseño 53
6.3.2. Test-Driven Development 54
6.3.3. Testing unitario vs. testing funcional 55
6.3.4. Producto Mínimo Viable 55
6.3.5. eXtreme Programming (XP) 55
6.3.6. Metaprogramación 56
6.3.7. Parseo y generación de código Smalltalk 56
7. Conclusiones y Trabajo Futuro 57
7.1. Conclusiones 57
7.2. Trabajo Futuro 58
7.2.1. Mejoras al *Extract Method* 58
7.2.2. Mejoras al *Extract Variable* 59
7.2.3. Mejoras al proyecto de refactorings 60
7.2.4. Mejoras a Cuis Smalltalk 60
8. Referencias 62
9. Anexo: contribuciones realizadas a Cuis 64

1. Introducción

Los refactorings automatizados (transformaciones de un programa sin variar su comportamiento) son parte de la mayoría de los ambientes de programación y son necesarios para trabajar con eficiencia y eliminar la probabilidad de errores triviales, dejando las tareas más complejas para quien escribe el programa.

En el caso de Smalltalk, el primer lenguaje en tener refactorings automatizados como parte de un ambiente de programación, existe una herramienta llamada Refactoring Browser, que fue implementada en varias distribuciones. Refactoring Browser posee dos importantes desventajas: tiene su propia representación del árbol de sintaxis abstracto (adicional a las que ya tenga cada distribución), y los refactorings no son capaces de preservar el formato del código existente.

Cuis Smalltalk es una distribución de Smalltalk de código abierto, multiplataforma y orientada a la simplicidad. Cuis posee una nueva implementación de refactorings en la que se apunta a resolver los problemas históricos del Refactoring Browser, y así poder tener una mayor variedad de refactorings, con implementaciones robustas y que sean capaces de utilizarse de manera versátil para lograr refactorings más avanzados.

En este trabajo se presenta el desarrollo de dos refactorings: *Extract Method* y *Extract Variable*, para Cuis Smalltalk. Dichos refactorings son al día de hoy parte de la distribución y utilizados frecuentemente por toda la comunidad, tanto en la academia como en la industria. Para poder implementar los nuevos refactorings fueron necesarios cambios adicionales a Cuis (principalmente al parser) los cuales fueron cuidadosamente introducidos para contribuir positivamente a la calidad del código.

2. Contexto, problemas y motivaciones

2.1. Definiciones

2.1.1. Smalltalk

Smalltalk (Goldberg y Robson, 1983; Kay 1993) es el ambiente de programación con objetos vivos, reflexivo y metacircular diseñado por Alan Kay y Dan Ingalls; que dio inicio a la orientación a objetos; y del cual existen diferentes dialectos o distribuciones en la actualidad, algunas de ellas comerciales y otras de código abierto. También se suele referir usando la palabra Smalltalk sólo al lenguaje de programación utilizado en el ambiente.

Squeak (Nierstrasz, Ducasse y Pollet 2009)[1] es una distribución de Smalltalk de código abierto, multiplataforma, creada en el año 1996 con el propósito de continuar con la experiencia de Smalltalk '80. Posee una comunidad muy activa y es utilizado principalmente en investigación y en educación.

Morphic[2] es la implementación de elementos gráficos que fue diseñada para Squeak y sus derivados (incluido Cuis).

Un *browser* es la herramienta que todo ambiente Smalltalk posee para la visualización y edición de código. Desde allí mismo se realizan los refactorings y cualquier tarea que tenga que ver con la administración del código.

Cuis Smalltalk[3] (de aquí en adelante, Cuis) es una distribución de Smalltalk de código abierto y multiplataforma derivada de Squeak, con una diferencia importante en la filosofía de desarrollo: Cuis está orientado a la simplicidad, en el sentido que cualquier parte de este ambiente debería ser fácil de comprender y extender; se busca también un ambiente pequeño (en cantidad de código) pero poderoso (sobre el cual se puedan desarrollar bibliotecas para diferentes propósitos). La distribución fue creada por el desarrollador argentino Juan Manuel Vuletich, quien lidera el proyecto desde entonces. Cuis se destaca por su velocidad y estabilidad, y posee una comunidad de decenas de programadores/as a lo largo del mundo, varios de los cuales realizan contribuciones con frecuencia.

Cuis University[4] (Wilkinson, Prieto y Garbezza 2018) es una distribución de Smalltalk basada en Cuis, creada por Hernán Wilkinson y Máximo Prieto, diseñada especialmente para el aprendizaje de programación orientada a objetos. Cuis University agrega a Cuis elementos que aportan a la didáctica y herramientas preinstaladas para facilitar el trabajo a lo largo de un curso, sin tener que instalar agregados. Los elementos incluidos en Cuis University son:

---

[1] Squeak Smalltalk: https://squeak.org/
[2] Morphic: http://wiki.squeak.org/squeak/morphic
[3] Cuis Smalltalk (J. Vuletich y contribuyentes): http://www.cuis-smalltalk.org/
[4] Cuis University (H. Wilkinson, M. Prieto y contribuyentes): https://cuisuniversity.org/

- Denotative Objects[5], una herramienta para trabajar con objetos concretos en lugar de clases e instancias.
- Live Typing (Wilkinson 2019), un sistema de anotación de tipos en tiempo de ejecución con el fin de mejorar las herramientas de navegación y refactoring de código.
- Aconcagua (Wilkinson, Prieto y Romeo 2005a), biblioteca de medidas y unidades.
- Chaltén (Wilkinson, Prieto y Romeo 2005b), biblioteca de manejo de fechas e intervalos de tiempo en general.
- TDDGuru, una herramienta que analiza cambios en la imagen de Cuis para detectar si se realizó o no Test-Driven Development, y cómo fue paso a paso la utilización de la técnica.
- Stack limit, una herramienta que previene problemas de recursión infinita dentro de la imagen de Cuis.

2.1.2. Refactorings

Se denomina *refactoring* (Odpyke, 1992; Fowler 1999) a cualquier proceso, manual o automático, en el que se cambia la implementación de una pieza de software con el fin de obtener un resultado mejor (en términos de requerimientos no funcionales como legibilidad, robustez o performance), sin alterar su comportamiento.

Normalmente cuando se menciona la palabra *refactoring* hacemos referencia a refactorings automáticos y conocidos. Automáticos porque es la computadora, con ciertos datos de entrada provistos por la persona que programa, la que realiza todos los pasos necesarios y genera los resultados; y conocidos, porque generalmente hay una serie de refactorings que tienen un nombre con el que se los identifican en la comunidad, y de los que hay documentación escrita. De esta forma cuando se habla de *Extract Method*, por ejemplo, se hace referencia a una transformación específica del programa en la que ya se conocen cuáles son sus entradas y salidas, sus precondiciones y los pasos que realiza para llegar al resultado.

Extract Method es el refactoring que consiste en la extracción de una colaboración o conjunto de colaboraciones dentro de un método, a un método separado, dándole un nombre que represente la semántica de dichas colaboraciones, y agregando los colaboradores externos que sean necesarios para realizar la tarea. Es uno de los refactorings más usados ya que contribuye a la expresividad del código como así también a reducir la cantidad total de colaboraciones de un método, abstrayendo detalles de implementación.

Respecto a las colaboraciones que se pueden extraer, estas pueden representar expresiones de las cuales nos interesa el resultado (fig. 2.a); como así también pueden ser varias colaboraciones de un método (fig. 2.b). Existen varias precondiciones para que este refactoring pueda aplicarse, que serán desarrolladas más en la sección 3.

Antes de aplicar *Extract Method*
`FillInTheBlankMorph >> defaultColor`

[5] Denotative Objects: https://github.com/hernanwilkinson/Cuis-Smalltalk-DenotativeObject

```
	"answer the default color/fill style for the receiver"
	^ Theme current menu
```

Después de aplicar *Extract Method*

```
FillInTheBlankMorph >> defaultColor
	"answer the default color/fill style for the receiver"
	^ self currentTheme menu

FillInTheBlankMorph >> currentTheme

	^ Theme current
```

Fig. 2.a - Ejemplo de *Extract Method* sobre una expresión Smalltalk

Antes de aplicar *Extract Method*

```
ResizeMorph >> initialize
	super initialize.
	extent := `400@300`.
	grid := `8@6`.
	gridLineWidth := 2.
	color := `Color white`.
	gridColor := `Color black`.
	selectionColor := `Color red`
```

Después de aplicar *Extract Method*

```
ResizeMorph >> initialize
	super initialize.
	self initializeDimensions.
	color := `Color white`.
	gridColor := `Color black`.
	selectionColor := `Color red`

ResizeMorph >> initializeDimensions
	extent := `400@300`.
	grid := `8@6`.
	gridLineWidth := 2
```

Después de aplicar *Extract Method*

```
ResizeMorph >> initialize
	super initialize.
	self initializeDimensions.
	self initializeColors

ResizeMorph >> initializeDimensions
	extent := `400@300`.
	grid := `8@6`.
	gridLineWidth := 2
```

```
ResizeMorph >> initializeColors
	color := `Color white`.
	gridColor := `Color black`.
	selectionColor := `Color red`
```

Fig. 2.b - Ejemplo de *Extract Method* sobre conjuntos de colaboraciones Smalltalk

Extract Variable es el refactoring que consiste en la extracción de una expresión (objeto o colaboración) a una variable local en el método donde ocurre, declarada en el *scope* más cercano posible. Este refactoring puede tener varios propósitos:

- Dar un nombre a un objeto para mejorar la legibilidad del código (una intención similar a la que puede lograrse con *Extract Method*).
- Eliminar duplicación en caso de colaboraciones u objetos repetidos a lo largo de la definición de un método.
- Parametrizar objetos que luego pueden ser extraídos utilizando *Extract Method*. Este es un caso en el que no se lo usa para un propósito final, sino como paso intermedio para llegar a una solución que implica combinar más de un refactoring.

La figura 2.c muestra un ejemplo de este refactoring.

Antes de aplicar *Extract Variable*

```
Character >> isSeparator
	"Answer whether the receiver is one of the separator characters
	space, tab, lf, cr, or form feed."

	^ #(32 9 10 13 12) statePointsTo: self numericValue
```

Después de aplicar *Extract Variable*

```
Character >> isSeparator
	"Answer whether the receiver is one of the separator characters
	space, tab, lf, cr, or form feed."

	| separatorCharacterValues |
	separatorCharacterValues := #(32 9 10 13 12).
	^ separatorCharacterValues statePointsTo: self numericValue
```

Fig. 2.c - Ejemplo de *Extract Variable* aplicado a una expresión Smalltalk

2.1.3. Parseo y generación de código

Un *parser* (Aho 2013) es el responsable de generar información con semántica a partir de un código fuente, dicha información se utiliza para la compilación, o para los refactorings, entre otros usos. Hay dos pasos importantes en este proceso: el análisis léxico, en donde el código fuente se divide en porciones denominadas *tokens*, y luego el análisis sintáctico, donde se le da semántica a los *tokens* y se construye con ellos una estructura, generalmente de tipo árbol, con los elementos que el programa necesita para su ejecución o análisis.

En Cuis, el proceso de parsing lo realiza la clase `Parser` (análisis léxico y sintáctico), que es subclase de `Scanner` (que realiza sólo la parte de análisis léxico).

Un compilador es el responsable de generar, en base a una representación que construye un parser, otra representación en un lenguaje diferente, y generalmente de más bajo nivel.

En Smalltalk el código que se ejecuta no es Smalltalk (que es lo mismo que decir que Smalltalk no es un lenguaje interpretado), sino que se ejecuta una representación del mismo que se conoce como *bytecode*. El proceso de compilación ocurre cada vez que se genera o se cambia un método y es completamente transparente a quien programa, siendo el objetivo que no sea necesario comprender o visualizar los *bytecodes*.

Árbol de sintaxis abstracto (*Abstract Syntax Tree* en inglés, o su sigla AST), es como se denomina generalmente a la estructura generada por el parser como resultado de su tarea. La característica de un AST es que contiene diferentes tipos de nodos, denominados *parse nodes*; cada uno de ellos representará diferentes elementos del lenguaje: variables, envíos de mensaje, asignaciones entre otros. La figura 2.d muestra un resumen del árbol de sintaxis abstracto de un método sencillo en Cuis.

```
String >> asText
	"Answer a Text whose string is the receiver."

	^Text fromString: self
```

Núm. de nodo	Tipo de nodo	Qué representa	Sub-nodos
1 (raíz)	`MethodNode`	Todo el método	2
2	`BlockNode`	Cuerpo del método `^Text fromString: self`	3
3	`ReturnNode`	Expresión de retorno `^Text fromString: self`	4
4	`MessageNode`	Envío de mensaje `Text fromString: self`	5, 6, 7
5	`LiteralVariableNode`	Objeto receptor del mensaje clase `Text`	no posee
6	`SelectorNode`	Nombre del mensaje (`#fromString:`)	no posee
7	`VariableNode`	Parámetro del mensaje (pseudo variable `self`)	no posee

Fig. 2.d - Ejemplo de un AST de un método de Cuis

2.1.4. Elementos de Cuis

Un *source range* es la forma en la que Cuis denota posiciones de nodos del AST en el código fuente. Están implementados con instancias de la clase `Interval`, donde el inicio

del intervalo representa la primer posición en el código fuente que el nodo representa, y el final del intervalo la última posición en el código fuente que el nodo representa. Un nodo del AST puede tener uno o más *source ranges*. Los *source ranges* se definen al momento de parsear el código fuente. La figura 2.e muestra los *source ranges* que se reportan para un método sencillo.

```
Magnitude >> <= aMagnitude
	"Answer whether the receiver is less than or equal to the argument."

	^(self > aMagnitude) not
```

Elemento del código	Clase del Nodo AST	*Source range(s)*
`self`	`VariableNode`	`(90 to: 93)`
`aMagnitude`	`VariableNode`	`(4 to: 13)` `(97 to: 106)`
`> aMagnitude`	`MessageNode`	`(95 to: 106)`
`not`	`MessageNode`	`(109 to: 111)`
`^(self > aMagnitude) not`	`ReturnNode`	`(88 to: 111)`

Fig. 2.e - Ejemplos de *source ranges* reportados en un método de Cuis

Un *changeset* (conjunto de cambios) es la forma que tradicionalmente poseen las distribuciones de Smalltalk para distribuir propuestas de cambios. En un archivo (con la extensión .cs.st) se incluyen todos los cambios propuestos: definiciones de clases o métodos nuevos, cambios a métodos existentes, agregado de comentarios, recategorización de mensajes, entre otros cambios. Generalmente también este archivo se asocia a un autor, y tiene una descripción que explica el propósito del cambio. El objetivo es que otros usuarios puedan de una manera sencilla incorporar todos estos cambios en sus respectivos ambientes locales. La figura 2.f muestra un ejemplo de un *changeset*.

```
'From Cuis 5.0 [latest update: #4074] on 29 March 2020 at 5:06:59 pm'!
"Change Set:		4075-CuisCore-AuthorName-2020Mar29-16h47m
Date:			29 March 2020
Author:			Nahuel Garbezza

Fix Extract Method error occurring on some optimized selector cases"!

!MessageNode methodsFor: 'testing' stamp: 'RNG 3/29/2020 17:02:11'!
hasEquivalentArgumentsWith: aMessageNode

	self arguments with: aMessageNode arguments do:
		[ :myArgument :otherParseNodeArgument | (myArgument equivalentTo:
otherParseNodeArgument) ifFalse: [ ^ false ] ].
	^ true! !
```

```
!MessageNode methodsFor: 'testing' stamp: 'RNG 3/29/2020 17:06:17'!
equivalentTo: aParseNode

	^ aParseNode isMessageNode
		and: [ self receiver equivalentTo: aParseNode receiver ]
		and: [ self selector = aParseNode selector ]
		and: [ self hasEquivalentArgumentsWith: aParseNode ]! !
```

Fig. 2.f - Ejemplo de un archivo que contiene un *changeset* de Cuis

2.1.5. Metodología

Test-Driven Development (Beck 2002) (de aquí en adelante, TDD) es la técnica de desarrollo de software creada por Kent Beck en la que se propone construir el software en base a pequeños incrementos, cada uno de ellos partiendo primero de una prueba (test) que falla, para luego escribir el código mínimo necesario para que dicha prueba pase, y finalmente mejorar la solución escrita hasta el momento. En general se usa el término *test-first* para referirse al enfoque de comenzar por pruebas y luego por la solución, opuesto al enfoque de programar la solución para luego escribir las pruebas.

2.2. Estado del arte: Refactorings existentes en diferentes distribuciones de Smalltalk

Tanto *Extract Method* como *Extract Variable* se encuentran implementados en la mayoría de las distribuciones de Smalltalk. Se eligieron tres distribuciones de Smalltalk (Squeak, Pharo[6], VisualWorks[7]) para comparar su desempeño respecto a la versión de Cuis implementada en este trabajo.

Las implementaciones de refactorings en Squeak, Pharo y VisualWorks están todas basadas en Refactoring Browser, el primer proyecto de refactorings automáticos para Smalltalk (Roberts., Brant & Johnson 1997). Refactoring Browser fue implementado inicialmente para VisualWorks y luego se realizaron *ports* a otras distribuciones. Es por ello que la funcionalidad en líneas generales es muy similar en estas tres distribuciones de Smalltalk, aunque luego en cada distribución se agregaron algunas funcionalidades o se mejoraron algunos mensajes de error.

Se analizaron los refactorings desde 4 puntos de vista diferentes: casos de éxito (qué funcionalidad es soportada por cada refactoring), casos de error (cómo cada refactoring informa los errores que ocurren), experiencia de usuario (cómo es la interfaz gráfica, como se interactúa para iniciar los refactorings y ver sus resultados), y por último la cantidad de tests, como un indicador de calidad técnica.

[6] Ambiente de programación Pharo Smalltalk: https://pharo.org/
[7] Ambiente de programación Cincom® VisualWorks®: http://www.cincomsmalltalk.com/main/products/visualworks/

2.2.1. Comparativa de *Extract Method*

La figura 2.g muestra la comparativa de casos de éxito para *Extract Method*, en la cual se puede observar que desde el punto de vista funcional, tanto las distribuciones con Refactoring Browser (Squeak, Pharo, VisualWorks) como Cuis soportan los casos más frecuentes de uso, con algunas salvedades en la implementación de Cuis: la capacidad de no agregar caracter de retorno si no es necesario, y poder extraer código que incluya declaración de variables temporales que deben moverse al método de destino y variables que deban quedarse en el método de origen.

Caso de uso de *Extract Method*	Comportamiento en las diferentes distribuciones de Smalltalk			
	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
Extracción de objetos literales	Soportado	Soportado	Soportado	Soportado
Extracción de expresiones que incluyan más de un envío de mensaje	Soportado	Soportado	Soportado	Soportado
Extracción que incluya declaración de variables temporales	Soportado	Soportado	Soportado	Soportado, sólo si se extrae código con todas las variables temporales de la declaración
Extracción de asignación a variable temporal	Soportado	Soportado	Soportado	Soportado si el código incluye todos los usos de la variable temporal
Introducción de paréntesis para no alterar orden de evaluación	Soportado	Soportado	Soportado	Soportado
No incluir caracter de retorno si lo que se extrae es un *statement*	Soportado, con cualquier cantidad de *statements*	Soportado, con cualquier cantidad de *statements*	Soportado, con cualquier cantidad de *statements*	Sólo si se extrae más de un *statement*

Fig. 2.g - Tabla comparativa: casos de éxito de *Extract Method* en diferentes distribuciones de Smalltalk

La figura 2.h muestra la comparativa de casos de error para *Extract Method*. Aquí no se encuentran muchas diferencias: los mensajes de error son descriptivos, y excepto por Squeak se pueden recibir advertencias para continuar con el refactoring, en el caso de redefinir un método existente.

Caso de uso de *Extract Method*	Comportamiento en las diferentes distribuciones de Smalltalk

	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
La expresión contiene retorno	Soportado, con mensaje de error descriptivo. Además si se selecciona una expresión con retorno, automáticamente se asume que se desea retornar el método extraído	Soportado, con mensaje de error descriptivo. Además si se selecciona una expresión con retorno, automáticamente se asume que se desea retornar el método extraído	Soportado, con mensaje de error descriptivo. Además si se selecciona una expresión con retorno, automáticamente se asume que se desea retornar el método extraído	Soportado, con mensaje de error descriptivo
El método a extraer ya está definido en la clase actual	Soportado, pero lanza debugger con el error	Soportado, con mensaje de error descriptivo. Lanza advertencia que permite continuar con el refactoring sobrescribiendo el método	Soportado, con mensaje de error descriptivo. Lanza advertencia que permite continuar con el refactoring sobrescribiendo el método	Soportado, con mensaje de error descriptivo. Lanza advertencia que permite continuar con el refactoring sobrescribiendo el método
El método a extraer ya está definido en una superclase	Soportado, pero lanza debugger con el error	Soportado, con mensaje de error descriptivo. Lanza advertencia que permite continuar con el refactoring redefiniendo el método en la subclase	Soportado, con mensaje de error descriptivo. Lanza advertencia que permite continuar con el refactoring redefiniendo el método en la subclase	Soportado, con mensaje de error descriptivo. Lanza advertencia que permite continuar con el refactoring redefiniendo el método en la subclase
El intervalo a extraer no contiene una expresión Smalltalk completa	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo
El intervalo a extraer es el lado izquierdo de una asignación	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo
El intervalo a extraer forma parte de la cabecera del método	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo

Fig. 2.h - Tabla comparativa: casos de error de *Extract Method* en diferentes distribuciones de Smalltalk

La figura 2.i muestra la comparativa de experiencia de usuario para *Extract Method*. Aquí es donde más diferencias hay. La diferencia más importante radica en la pérdida del formato, que ocurre consistentemente en Squeak, Pharo y VisualWorks. En estas distribuciones, una vez finalizado el refactoring, tanto el método de origen como el de destino son formateados acorde a las reglas existentes de formateo de código, eliminando así cualquier formateo realizado intencionalmente por quien escribió el código. Luego, VisualWorks y Pharo soportan el reordenamiento de parámetros y el reemplazo de

expresiones similares en el código luego de finalizar el refactoring. Squeak y Cuis aún no tienen esta funcionalidad.

Caso de uso de *Extract Method*	Comportamiento en las diferentes distribuciones de Smalltalk			
	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
Instalado por defecto	No (en paquete RefactoringTools)	Sí	Si	Sí
Atajo de teclado para iniciar el refactoring	No	Ctrl+t inicia menú que tiene el refactoring como una opción disponible	Sí, es configurable	Sí, Ctrl+Shift+k
Menú contextual para iniciar el refactor	Sí	Sí	Sí	Sí
Preservar formato de código original y extraído	No, altera formato de método original y nuevo	No, altera formato de método original y nuevo	No, altera formato de método original y nuevo	Soportado
Seleccionar con o sin paréntesis	Soportado, remueve paréntesis adicionales	Soportado, remueve paréntesis adicionales	Soportado, remueve paréntesis adicionales	Soportado
Seleccionar incluyendo espacios en blanco o puntos (.)	Soportado, remueve espacios adicionales debido a la pérdida de formato	Soportado, remueve espacios adicionales debido a la pérdida de formato	Soportado, remueve espacios adicionales debido a la pérdida de formato	Soportado
Sugerir nombre de mensaje con cantidad de parámetros correcta	Soportado, aunque hay que escribir a mano los nombres de los parámetros	Soportado	Soportado	Soportado
Poder renombrar parámetros	No soportado, y el refactoring genera código inválido si se intenta renombrar	Soportado, a través del formulario previo a aplicar el refactoring	No soportado	No soportado, con validación que previene renombres
Poder reordenar parámetros	No soportado	Soportado, a través del formulario previo a aplicar el refactoring	Soportado, a través del formulario previo a aplicar el refactoring	No soportado
Sugerir extracción de fragmentos de código similares	No soportado	Soportado, aplica refactoring y luego sugiere fragmentos de código repetidos en el método actual	Sugiere utilizar un método existente si ese mismo código fue extraído anteriormente	No soportado

Fig. 2.i - Tabla comparativa: experiencia de usuario de *Extract Method* en diferentes distribuciones de Smalltalk

La figura 2.j muestra la comparativa respecto a cantidad de tests en *Extract Method*. Aquí el que se destaca es Cuis, ya que por su implementación realizada completamente con TDD, cuenta con una vasta cantidad de tests.

Extract Method	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
Cantidad de tests	9	11	No fue posible obtener tests	40

Fig. 2.j - Tabla comparativa: cantidad de tests de *Extract Method* en diferentes distribuciones de Smalltalk

2.2.2. Comparativa de *Extract Variable*

La figura 2.k muestra la comparativa de casos de éxito para *Extract Variable*, en la cual se puede observar que los casos más comunes de uso están soportados por todas las distribuciones.

Caso de uso de *Extract Variable*	Comportamiento en las diferentes distribuciones de Smalltalk			
	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
Extracción de objetos literales	Soportado	Soportado	Soportado	Soportado
Extracción de expresiones que incluyan más de un envío de mensaje	Soportado	Soportado	Soportado	Soportado
Extracción dentro de bloques de código	Soportado	Soportado	Soportado	Soportado

Fig. 2.k - Tabla comparativa: casos de éxito de *Extract Variable* en diferentes distribuciones de Smalltalk

La figura 2.l muestra la comparativa de casos de error para *Extract Variable*, en la cual se puede observar que en líneas generales Cuis es el que mejores mensajes de error posee, y se detecta un error muy particular en todas las implementaciones de Refactoring Browser, lo cual podría tratarse de un *bug*.

Caso de uso de *Extract Variable*	Comportamiento en las diferentes distribuciones de Smalltalk			
	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
La variable a definir ya existe en el método actual	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo
El nombre de variable no es válido	Soportado, con mensaje de error descriptivo	Soportado, aunque el mensaje de error dice “variable de	Soportado, aunque el mensaje de error dice “variable de	Soportado, con mensaje de error descriptivo

		instancia” en lugar de “variable temporal”	instancia” en lugar de “variable temporal”	
La variable temporal existe como una variable de instancia	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo	Soportado, con mensaje de error descriptivo
El intervalo a extraer no contiene una expresión Smalltalk completa	Soportado (aunque valida luego de preguntar por el nombre de variable)	Realiza el refactoring seleccionando automáticamente el nodo AST más próximo	Pregunta nombre de variable y luego falla con el mensaje: “Cannot assign to non value nodes”	Soportado, con mensaje de error descriptivo
El intervalo a extraer forma parte de la cabecera del método	Intenta hacer el refactoring y falla abriendo el debugger	Intenta hacer el refactoring y falla abriendo el debugger	Intenta hacer el refactoring y falla abriendo el debugger	Soportado, con mensaje de error descriptivo

Fig. 2.l - Tabla comparativa: casos de error de *Extract Variable* en diferentes distribuciones de Smalltalk

La figura 2.m muestra la comparativa respecto a experiencia de usuario de *Extract Variable*. En este caso Cuis parece ser el que mejor se comporta: al igual que en el *Extract Method*, es el único que preserva formato y maneja mejor el caso de extraer con cambios sin guardar. Pharo tiene además la posibilidad de reemplazar ocurrencias de la misma expresión con la variable extraída, pero lo realiza siempre sin preguntar, lo que podría resultar en un cambio en el comportamiento.

Caso de uso de *Extract Variable*	Comportamiento en las diferentes distribuciones de Smalltalk			
	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
Instalado por defecto	No (en paquete RefactoringTools)	Sí	Sí	Sí
Atajo de teclado para iniciar el refactoring	No	Ctrl+t inicia menú que tiene el refactoring como una opción disponible	Sí, es configurable	Sí, Ctrl+Shift+j
Menú contextual para iniciar el refactor	Sí	Sí	Sí	Sí
Intentar extraer con cambios sin guardar	Falla silenciosamente, editor de texto queda en estado inválido	Falla silenciosamente, editor de texto queda en estado inválido	Pregunta si se desean guardar los cambios, pero si se elige que no, el refactoring continúa y falla más adelante	Avisa que es necesario guardar cambios
Preservar formato de código original y extraído	No, altera formato del método en su totalidad	No, altera formato del método en su totalidad	No, altera formato del método en su totalidad	Soportado

Seleccionar con o sin paréntesis	Soportado, remueve paréntesis adicionales	Soportado, remueve paréntesis adicionales	Soportado, remueve paréntesis adicionales	Soportado, respeta paréntesis existentes en el código
Seleccionar incluyendo espacios en blanco o puntos (.)	Soportado	Soportado	Soportado	Soportado
Sugerir extracción de fragmentos de código similares	No soportado	Reemplaza fragmentos iguales dentro del método sin preguntar	No soportado	No soportado

Fig. 2.m - Tabla comparativa: experiencia de usuario de *Extract Variable* en diferentes distribuciones de Smalltalk

La figura 2.n muestra la comparativa en cantidad de tests de *Extract Variable*. Nuevamente, al igual que en *Extract Method*, Cuis es el que mayor cantidad de tests posee, principalmente debido al desarrollo con TDD.

Extract Variable	Squeak (v5.2 revisión 18229)	Pharo (v8.0.0 build 1136)	VisualWorks (v8.3)	Cuis (v5 revisión 4249)
Cantidad de tests	5	6	No fue posible obtener tests	24

Fig. 2.n - Tabla comparativa: cantidad de tests de *Extract Method* en diferentes distribuciones de Smalltalk

2.2.3. Resumen de la comparativa

En cuanto a nuevas funcionalidades, el que se destaca es Pharo: para el *Extract Method*, sugiere fragmentos de código duplicados para ser reemplazados por el nuevo método que se extrae (se puede aceptar o rechazar la sugerencia), que es algo que originalmente no formaba parte del Refactoring Browser.

Lamentablemente, también se comparten algunos errores, como el caso de intentar extraer parte de la definición de un método a una variable local. Esto debería lanzar un mensaje de error diciendo que no es posible aplicar el refactoring en el texto seleccionado, sin embargo las tres distribuciones fallan de la misma manera, exponiendo el debugger con una excepción que ocurre en el código del refactoring.

En resumen, las desventajas de las implementaciones de Refactoring Browser ante la implementación de Cuis son consistentes en VisualWorks, Squeak y Pharo y se pueden resumir a lo siguiente:

- No preservan el formato del método original ni del nuevo método generado (en el caso del *Extract Method*) y del código de la asignación de variable (en el caso del *Extract Variable*). Esto se debe a que las transformaciones de código se realizan a nivel AST, y luego desde ese AST se genera el código. Los nodos del AST no conservan información de formato, por ende es imposible reconstruir el código tal como estaba originalmente después de haber pasado por un refactoring.
- Operan sobre un modelo de AST utilizado únicamente para refactorings. Esto representa una duplicación (a nivel técnico porque existen conjuntos similares de

clases, y conceptual porque realizan un propósito similar) respecto al AST que es el resultado del parseo de código.
- Poseen pocos tests automatizados en comparación a Cuis, lo que representa un riesgo de introducir regresiones, o de no saber de la existencia de ciertos errores.

2.3. Estado del arte: Refactorings en Cuis Smalltalk

Hasta el año 2017, Cuis no tenía refactorings automatizados. Esto hacía el desarrollo más lento y propenso a errores, lo cual requería de una atención extra a muchos detalles que podrían realizarse de manera automática.

En el año 2017, Hernán Wilkinson inició un proyecto para agregar refactorings a Cuis[8]. Permaneció como un paquete instalable hasta que en 2018 fue agregado a la distribución principal de Cuis. El catálogo de refactorings disponibles fue creciendo con el tiempo con varias contribuciones de estudiantes de diferentes universidades.

La diferencia que tiene este modelo de refactorings, con respecto a otras distribuciones de Smalltalk, es que opera directamente sobre el código fuente, valiéndose de la información de los *source ranges* que cada nodo del AST posee. No necesita de un modelo paralelo del código fuente u otra representación del AST, como sí ocurre con las implementaciones basadas en *Refactoring Browser*. También está pensado para ser desarrollado con TDD, con responsabilidades bien distribuidas en objetos con un bajo nivel de acoplamiento y altamente cohesivos.

En términos de implementación, un refactoring es una subclase de la clase `Refactoring`, cuyo protocolo de instancia incluye únicamente al mensaje abstracto `#apply`, mientras que su protocolo de clase incluye mensajes para lanzar errores o advertencias (*warnings*). Cada refactoring debe dar su implementación de `#apply`. Se espera que los refactorings que fallen (ya sea por una precondición que no se cumple o por algún otro motivo) lancen errores de tipo `RefactoringError` (situación de la cual no se puede recuperar para continuar el refactoring) o `RefactoringWarning` (situación de la cual se podría recuperar y continuar con el refactoring). Como para poder aplicar refactorings hay que proveerlos de diferente *user input*, los detalles de cómo obtener esa información pertenecen a otro tipo de objeto, el `RefactoringApplier`. `RefactoringApplier` es una clase abstracta, con lo cual se debe tener una subclase concreta para cada refactoring. El protocolo de `RefactoringApplier` incluye mensajes para obtener datos a través de la interfaz de usuario, manejar excepciones de refactorings y mostrar mensajes al usuario, y mostrar información que dependiendo del refactoring puede ser útil saber antes de aplicarlo y/o después de aplicarlo. Además, algunas precondiciones de refactorings están implementadas como subclases de `RefactoringPrecondition` (cuyo protocolo de instancia incluye únicamente al mensaje `#value`).

2.4. Objetivos

Los objetivos del trabajo son los siguientes:

[8] Proyecto de refactorings de Hernán Wilkinson:
https://github.com/hernanwilkinson/Cuis-Smalltalk-Refactoring

- Tener la mejor implementación posible de los refactorings *Extract Method* y *Extract Variable* en Cuis, considerando los requerimientos funcionales más importantes de cada refactoring y requerimientos no funcionales como robustez, claridad del código y performance.
- Mejorar el modelo actual de refactorings de Cuis a través de más comentarios en el código, mejores abstracciones de objetos necesarios para los refactorings, detalles de experiencia de usuario y otros cambios que surjan de agregar los nuevos refactorings propuestos en este trabajo. El objetivo es que el modelo de refactorings permanezca claro y fácil de extender para que más personas puedan contribuir con mejoras o nuevos refactorings.
- Mejorar el parser, el AST y otras herramientas auxiliares de Cuis en las áreas que se vean afectadas por los nuevos refactorings, ya sea incorporando funcionalidad que no esté previamente, o mejorando la calidad del código.
- Integrar los nuevos refactorings a la distribución principal de Cuis, y también a Cuis University, haciendo el trabajo disponible a toda la comunidad de profesionales, estudiantes y docentes que utilizan estas herramientas.

La motivación más importante es la de tener un ambiente de desarrollo más eficaz, tanto para personas que se inician en la programación (aprendiendo a trabajar con refactorings lo antes posible), como así también para usuarios avanzados (que utilizan de manera intensiva diferentes refactorings). Al ser Cuis y Cuis University desarrollos de código abierto, toda la comunidad de programadoras/es puede utilizar estos beneficios.

3. Refactoring: *Extract Method*

Como se menciona en la sección 2.1.2, el objetivo de este refactoring es mover código (objetos puntuales, colaboración o conjunto de colaboraciones) a un nuevo método con el fin de dar semántica a esa parte de código, reducir la cantidad de colaboraciones y aumentar la legibilidad del método en donde se origina el refactoring.

3.1. Funcionalidad

El refactoring necesita los siguientes datos de entrada:

- Método origen del cual se desea extraer código
- Intervalo de código del método en el cual deseamos realizar la extracción
- Nombre del mensaje a extraer (con sus nombres de colaboradores externos si es necesario)

Las siguientes validaciones se realizarán antes de aplicar el refactoring:

- Sobre el nuevo nombre de mensaje
  - Es un nombre válido: comienza con letra minúscula (en el caso de mensajes unarios o de palabra clave), no contiene espacios ni caracteres especiales (excepto el guión bajo). Ejemplos: `#value`, `#valueWith:`, `#+`, `#valueWith:and:`.
  - No existe actualmente un método definido con ese mismo nombre en la clase donde se realiza el refactoring o en alguna de sus superclases.
  - Acepta exactamente la misma cantidad de argumentos que objetos parametrizables que se encuentran en el intervalo de extracción de código.
- Sobre el intervalo de extracción de código:
  - Se encuentra dentro de los límites del método del cual hay que extraer.
  - Representa código Smalltalk válido para ser parte de un método separado
    - No hace retorno (denotado por el caracter `^`).
    - Es una expresión / conjunto de sentencias válidas (que pueden ser parseadas por sí mismas).
    - No contiene parte de la cabecera del método (nombre del mensaje, argumentos, definiciones de pragmas).
    - No es el lado izquierdo de una asignación (donde se encuentra la variable).

En caso de poder aplicarse, genera el siguiente resultado:

- Método origen modificado con el texto a extraer reemplazado con el nuevo envío de mensaje.
- Nuevo método relacionado al nombre del mensaje ingresado, definido en la misma clase, categorizado de la misma manera que el método original y con el conjunto de colaboraciones seleccionadas en el método original.

3.2. Interfaz de usuario

El refactoring se inicia desde un Browser de Cuis, ya sea utilizando el menú contextual o un atajo de teclado (Ctrl+Shift+k en Windows o GNU/Linux; Command+Shift+k en Mac). La figura 3.a muestra la opción del menú contextual agregada.

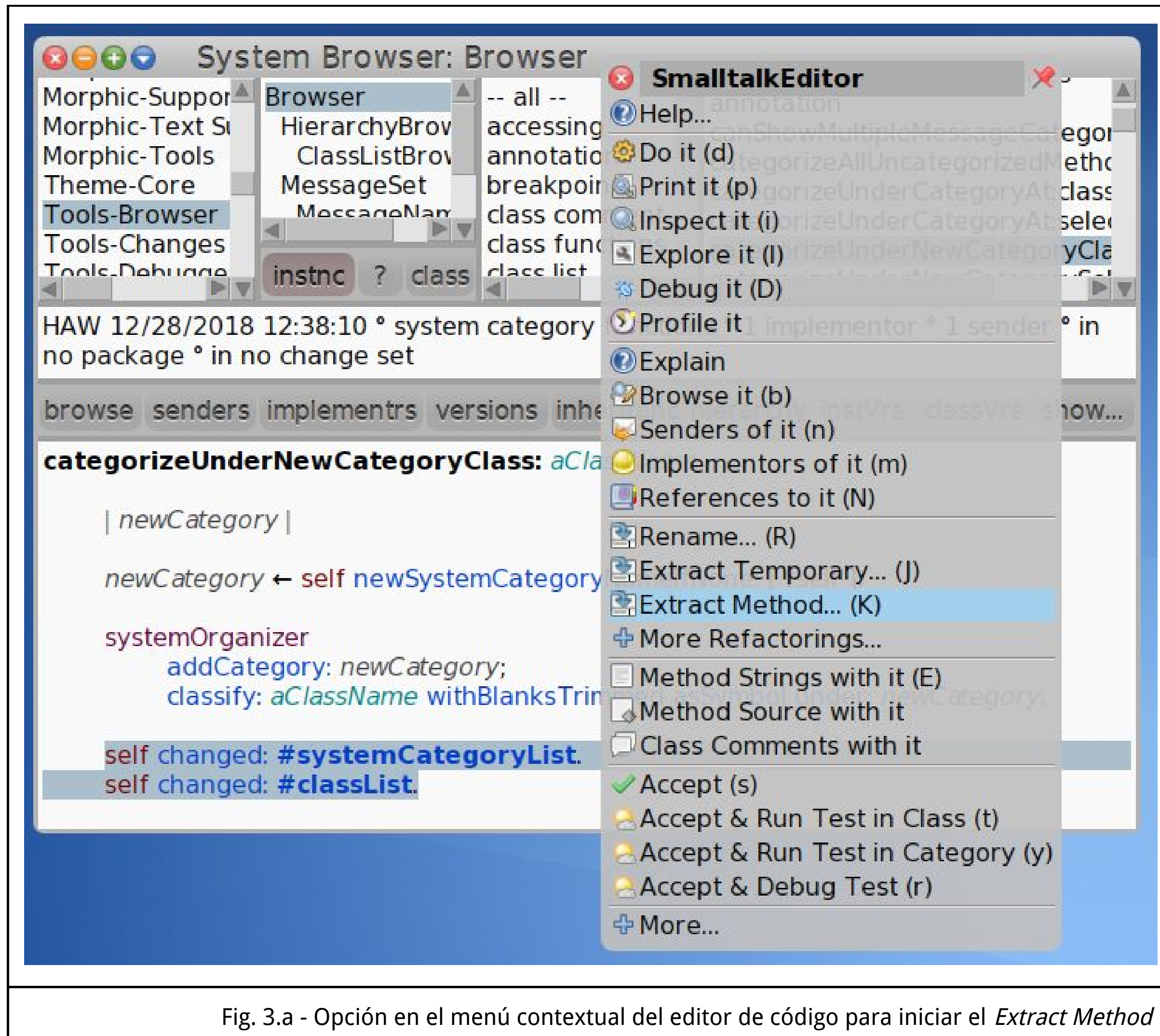


Fig. 3.a - Opción en el menú contextual del editor de código para iniciar el *Extract Method*

Al seleccionar el código deseado para extracción y presionar el atajo de teclado o la opción de menú, el próximo paso será elegir el nombre del mensaje nuevo. Inicialmente aparece una sugerencia de nombre (editable por completo) que tendrá un nombre acorde a la cantidad de objetos parametrizables o no. La figura 3.b muestra el cuadro para ingresar el nombre del mensaje con sugerencias predeterminadas para mensajes unarios, y la figura 3.c la sugerencia para un mensaje de palabra clave, en cuyo caso se nombran los parámetros para que se sepa a qué palabra clave corresponde cada nuevo parámetro.

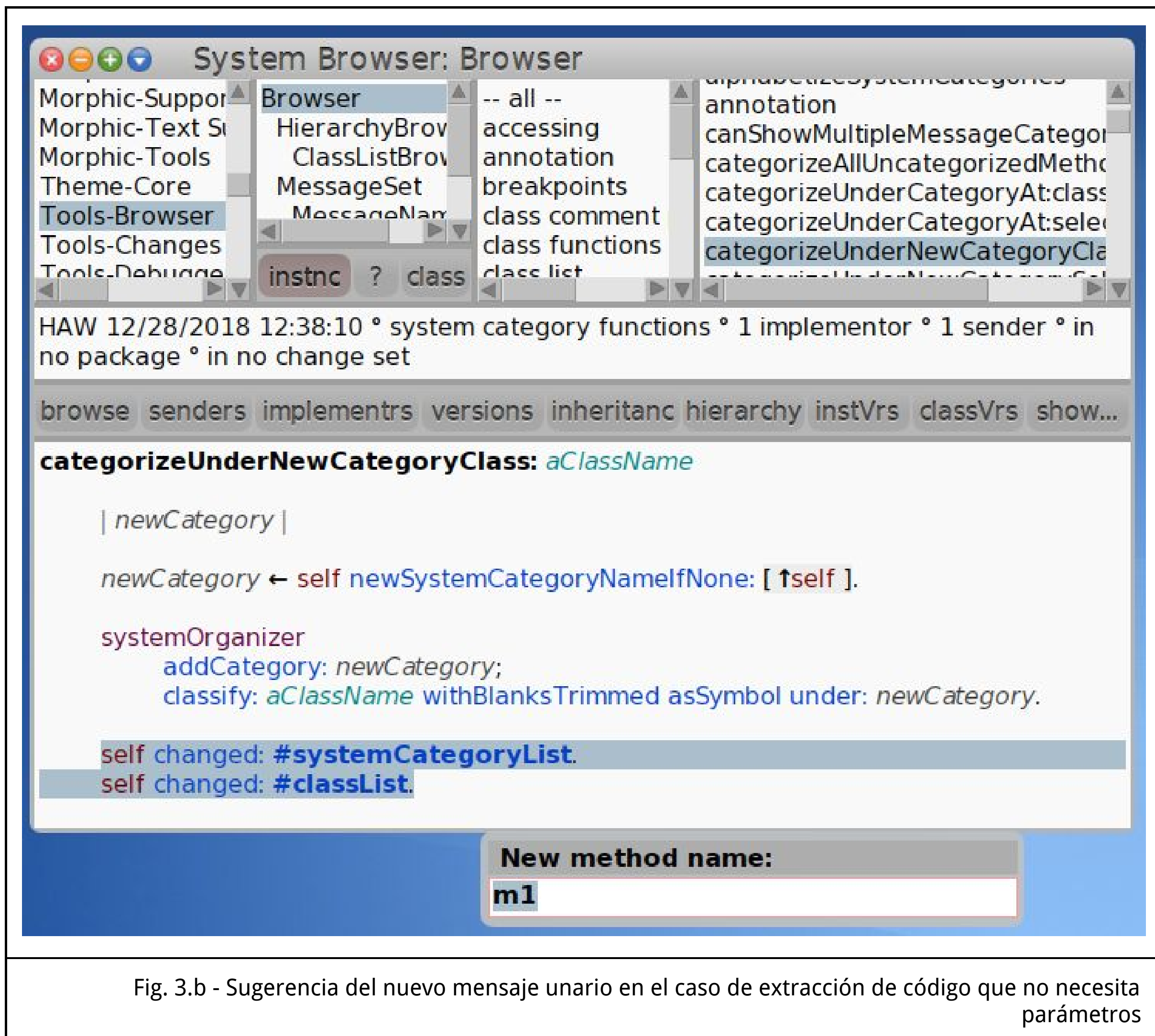


Fig. 3.b - Sugerencia del nuevo mensaje unario en el caso de extracción de código que no necesita parámetros

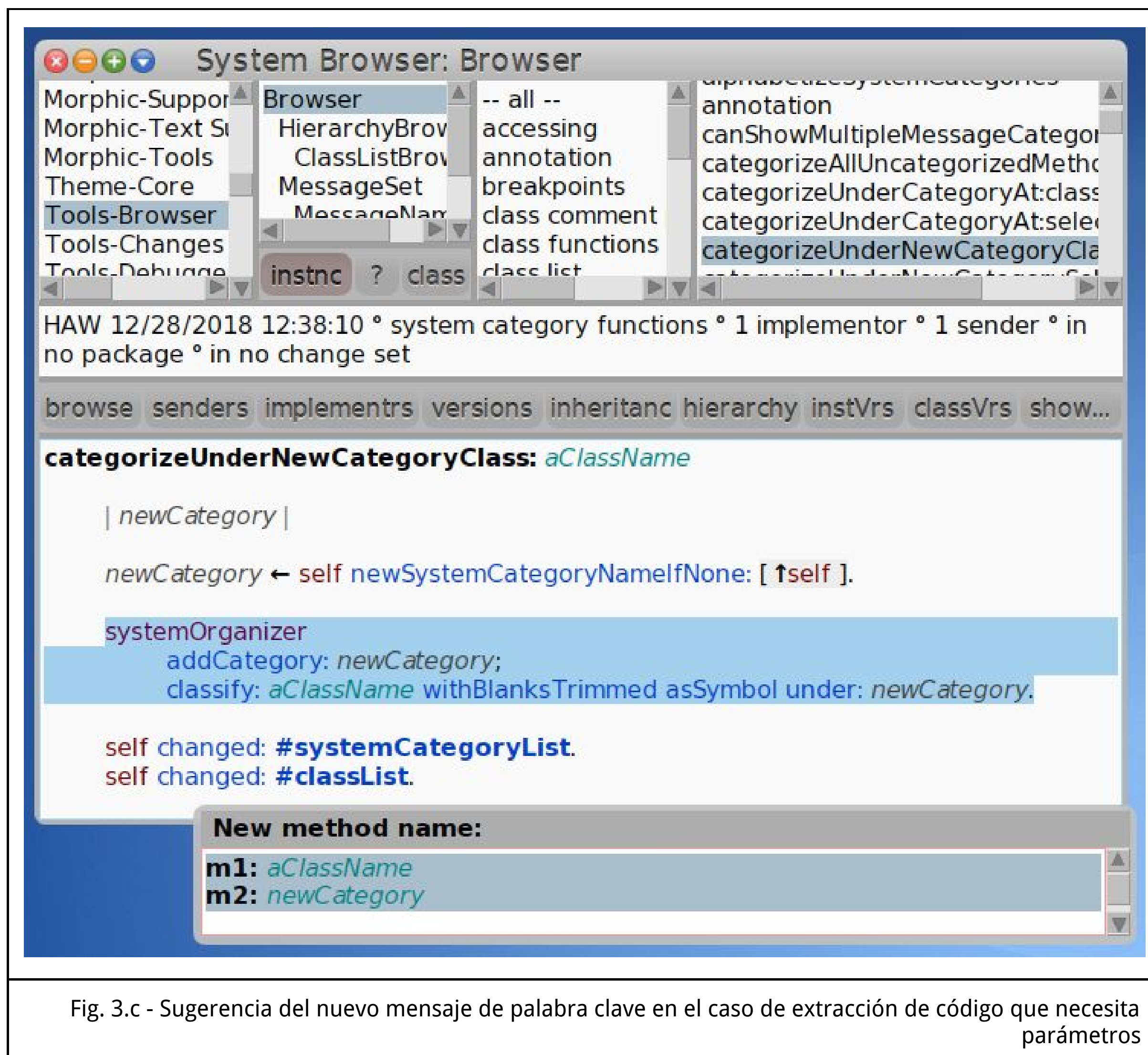


Fig. 3.c - Sugerencia del nuevo mensaje de palabra clave en el caso de extracción de código que necesita parámetros

En caso de ocurrir un error o advertencia, se mostrará una ventana correspondiente a cada caso. La figura 3.d muestra un ejemplo de error (selección de código inválida), en la cual no se puede continuar con el refactoring. La figura 3.e muestra un ejemplo de advertencia (mensaje ya existente) en la que se permite elegir si continuar o cancelar el refactoring.

En caso de realizar correctamente el refactoring, los cambios se verán reflejados inmediatamente en el Browser.

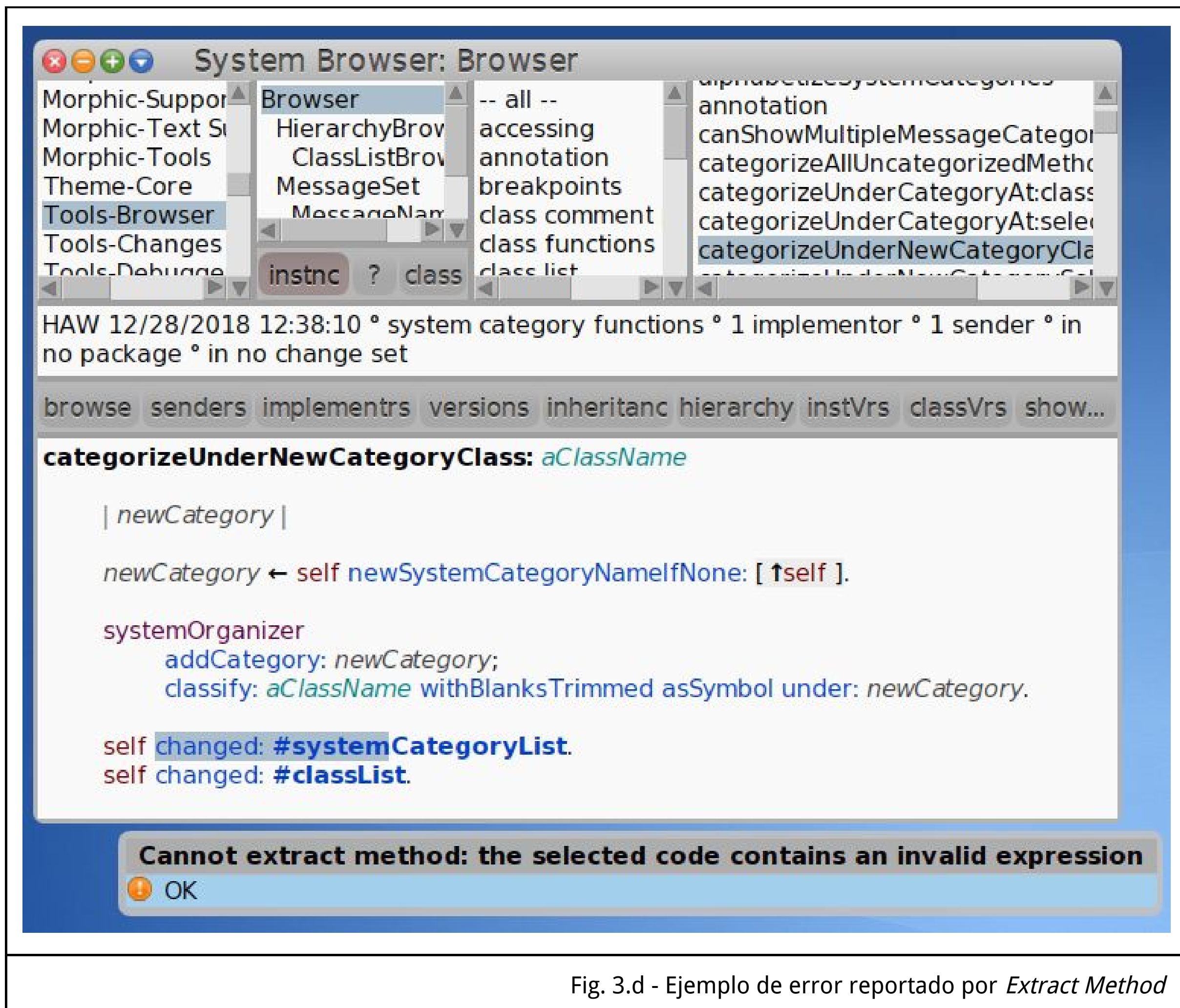


Fig. 3.d - Ejemplo de error reportado por *Extract Method*

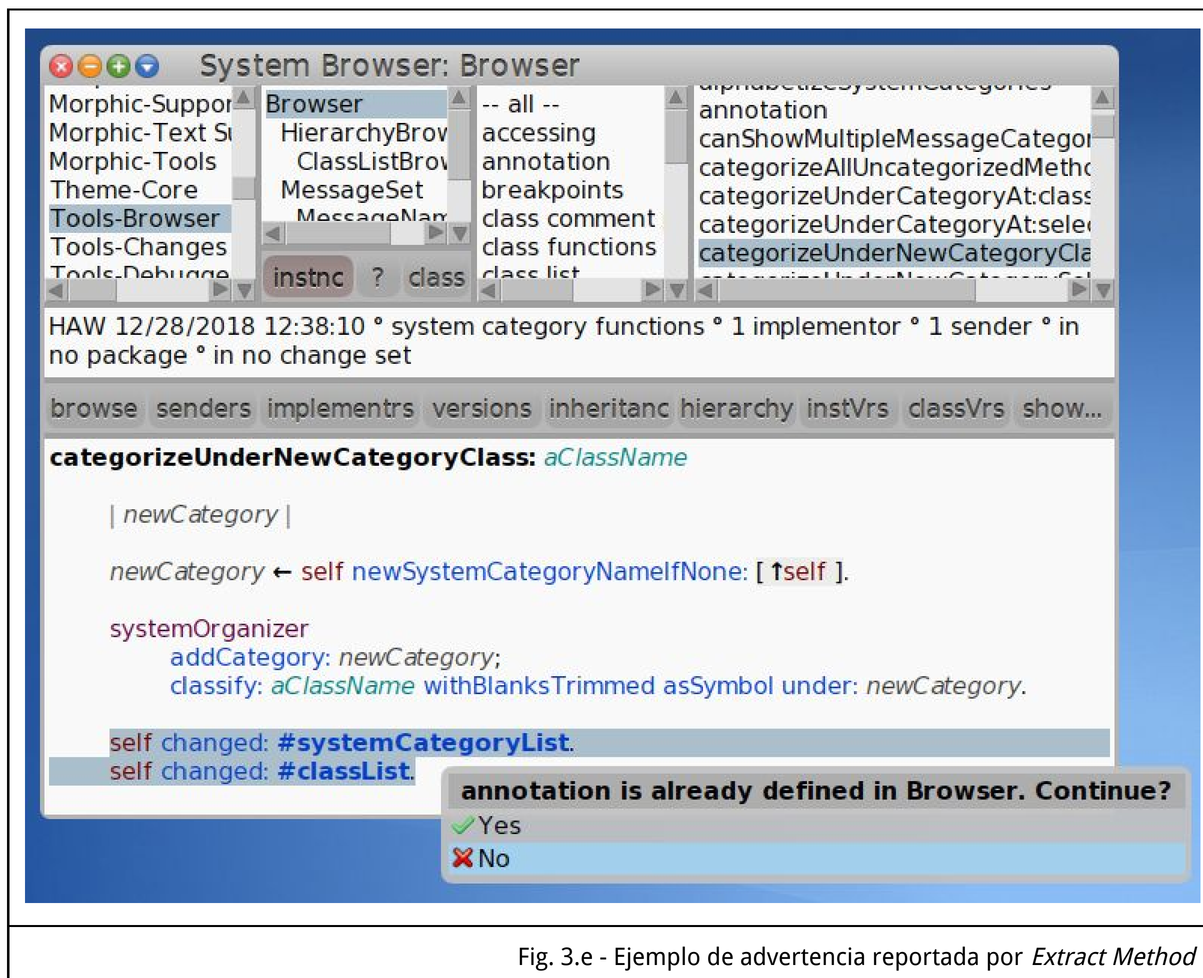


Fig. 3.e - Ejemplo de advertencia reportada por *Extract Method*

3.3. Implementación

La lógica propiamente dicha del refactoring está en la clase **`ExtractMethod`**. La figura 3.f muestra la definición de la clase. La clase posee cinco colaboradores internos:

- `intervalToExtract`: el intervalo donde se encuentra el código a extraer, instancia de **`SourceCodeInterval`**.
- `categoryOfNewSelector`: nombre de la categoría del nuevo método a definir. Por defecto es la misma categoría del método origen, pero puede personalizarse al momento de instanciar el refactoring, instancia de **`String`**.
- `newMessage`: el nuevo mensaje a definir, incluye su selector y nombre de argumentos si corresponde, instancia de **`Message`**.
- `existingMethod`: el método desde donde se inicia el refactoring, instancia de **`CompiledMethod`**.
- `extractedSourceCode`: calculado al momento de inicialización para ser utilizado luego (en base a `intervalToExtract` y `existingMethod`), es el código fuente extraído que formará parte del método nuevo. Instancia de **`String`**.

Definición de clase

```
Refactoring subclass: #ExtractMethod
  instanceVariableNames: 'intervalToExtract categoryOfNewSelector newMessage
```

```
extractedSourceCode existingMethod'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tools-Refactoring'
```

Comentario

```
I am a refactoring that extracts a selected piece of code to a separate method.
The input is the following:

* interval of code to extract (from index - to index)
* the CompiledMethod where this change applies
* the new method selector + argument names (instance of Message)
* the category name for the new method

Many conditions have to be satisfied for this refactoring to be made, I delegate
into ExtractMethodExpressionValidation and ExtractMethodNewSelectorPrecondition
some of these checks. Refer to the class comment of those classes for more
information.
```

Fig. 3.f - Definición de la clase `ExtractMethod` y su comentario

El único mensaje público de instancia requerido, como en la mayoría de los refactorings, es `#apply`, que en este caso debe realizar dos pasos: en primer lugar, definir el nuevo mensaje, luego cambiar el código del método actual para que llame al nuevo mensaje. Para ambos pasos se utiliza el mensaje `#compile:` para compilar los nuevos códigos fuente de cada mensaje. En la figura 3.g se ve la definición de `#apply` y sus dos principales pasos.

```
ExtractMethod >> apply

  self
    defineExtractedMethod;
    changeExistingMethod
```

```
ExtractMethod >> defineExtractedMethod

  self sourceClass
    compile: self newMethodSourceCode
    classified: categoryOfNewSelector
```

```
ExtractMethod >> changeExistingMethod

  self sourceClass
    compile: self updatedSourceCodeOfExistingMethod
    classified: existingMethod category
```

Fig. 3.g - Definición del método `#apply` de `ExtractMethod` y sus dos principales pasos

Previo a los mensajes de instancia, durante la fase de construcción de las instancias de `ExtractMethod` se valida que el refactoring pueda realizarse, de esta manera se asegura que el refactoring una vez instanciado puede aplicarse con seguridad. La figura 3.h muestra el mensaje de clase utilizado para instanciar el refactoring, donde previo a la construcción e inicialización de la instancia se realizan las validaciones necesarias.

```
ExtractMethod class >> fromInterval: anIntervalToExtract of:
aMethodToExtractCodeFrom to: newMessage categorizedAs: aCategory

  | trimmedIntervalToExtract |
  trimmedIntervalToExtract := anIntervalToExtract trimToMatchExpressionOn:
aMethodToExtractCodeFrom sourceCode.
  self
    assert: newMessage selector canBeDefinedIn: aMethodToExtractCodeFrom
methodClass;
    assertCanApplyRefactoringOn: aMethodToExtractCodeFrom at:
trimmedIntervalToExtract;
    assert: newMessage hasValidParametersForExtracting: anIntervalToExtract
from: aMethodToExtractCodeFrom methodNode.

  ^ self new
      initializeFrom: trimmedIntervalToExtract
      of: aMethodToExtractCodeFrom
      to: newMessage
      in: aCategory
```

Fig. 3.h - Definición del método de creación de instancias de `ExtractMethod`

Las validaciones que se realizan son las siguientes:

- El nuevo nombre de mensaje sea válido (para lo cual se delega en `ExtractMethodNewSelectorPrecondition`)
- El mensaje nuevo contiene la cantidad correcta de argumentos (para contar la cantidad de parámetros necesarios se delega en la clase `ExtractMethodParametersDetector`)
- El intervalo de código seleccionado contiene código válido para ser movido hacia otro método (para lo que se delega en la clase `SourceCodeOfMethodToBeExtractedPrecondition`)

En caso que alguna de estas validaciones falle, se lanzará un error de tipo `RefactoringError` o una advertencia de tipo `RefactoringWarning`, dependiendo el caso. La figura 3.i detalla todos los casos posibles de error, con la clase de error y el mensaje correspondiente.

Caso de error	Tipo de error lanzado	Mensaje de error
Nombre de mensaje vacío	`RefactoringError`	`'New selector can not be empty'`

Nombre de mensaje incluye caracteres de separación	`RefactoringError`	`'New selector can not contain separators'`
Nombre de mensaje incluye caracteres inválidos	`RefactoringError`	`'New selector should only contain letters, numbers or _'`
Nombre de mensaje existe ya en la clase	`RefactoringWarning`	`'selector is already defined in Class'`
Nombre de mensaje comienza con un caracter no válido	`RefactoringError`	`'New selector should begin with a lowercase letter or _'`
Intervalo de selección vacío	`RefactoringError`	`'Please select some code for extraction'`
Intervalo de selección está por fuera de las dimensiones del código fuente	`RefactoringError`	`'The requested source code selection interval is out of bounds'`
Intervalo de selección incluye una expresión con un caracter de retorno	`RefactoringError`	`'Cannot extract method: the selected code includes a return statement'`
Intervalo de selección incluye código que no representa una expresión completa	`RefactoringError`	`'Cannot extract method: the selected code contains an invalid expression'`
Intervalo de selección incluye código dentro de la declaración de variables temporales	`RefactoringError`	`'Cannot extract method: it is not possible to extract temporary variable definitions'`
Intervalo de selección incluye código en el lado izquierdo de una asignación	`RefactoringError`	`'Cannot extract method: it is not possible to extract the left side of an assignment'`
Extracción de código que incluye variables temporales sin su correspondiente declaración	`RefactoringError`	`'Cannot extract method: an assignment is being extracted without its declaration'`
Cantidad incorrecta de parámetros especificada como parte del mensaje nuevo	`RefactoringError`	`'The number of arguments in the given selector is not correct'`
Extracción de código que incluye referencias a variables temporales que son utilizadas	`RefactoringError`	`'Cannot extract method: there are temporary`

fuera del intervalo seleccionado		`variables used outside of the code selection'`
		Fig. 3.i - Errores que pueden generarse como parte de la ejecución de `ExtractMethod`

Para poder validar que el código que estamos extrayendo es una expresión o conjunto de expresiones que pueden ser movidas a otro método, se revisan los *complete source ranges* de los nodos involucrados. Es una validación compleja porque implica conectar los *source ranges* con la estructura de tipo grafo del AST (no es un árbol estrictamente hablando porque hay nodos como literales o variables que se reutilizan en diferentes lugares de la estructura).

Lo que se conoce de entrada es en qué posición se inicia el refactoring y en qué posición termina. Delimitado por estas posiciones, se espera encontrar algún *complete source range* para algún nodo del AST. Si esto no ocurriera, estaríamos ante un caso de un elemento sintáctico que no reporta correctamente los *source ranges*, que no posee *source ranges* en su totalidad, o que ni siquiera existe como nodo en el AST. En estos casos el refactoring no se puede realizar porque no existe seguridad que se pueda aplicar en ese contexto faltante de información.

En el caso en que conocemos qué nodo del AST está al inicio de la selección y qué nodo está al final, lo primero que se debe validar es que dichos nodos estén completos, es decir: que el inicio de la selección coincida con el inicio de un nodo del AST y que el final de la selección coincida con el final de un nodo del AST. Si esto no ocurre, estamos seguros que el refactoring no puede aplicarse, ya que esta selección representa una expresión incompleta.

En el caso más trivial el nodo de inicio y el nodo final son iguales, y en este caso el refactoring puede aplicarse sin problemas, siempre y cuando el resto de las validaciones pasen.

Si los nodos de inicio y final no son iguales se trata de una expresión que involucra más de un elemento. Se necesita un siguiente paso para validar alguno de estos dos escenarios:

- El inicio y fin de la selección está dentro de un mismo *statement*, y representan una sub-expresión válida (como por ejemplo un bloque o un envío de mensaje). Por ejemplo, en la expresión `1 + 2 * 3`, no es posible extraer `2 * 3`, pero sí `1 + 2`.
- El inicio y el fin de una selección están en distintos *statements*, entonces se requiere que el nodo de inicio coincida con el inicio de su *statement*, y el nodo final coincida con el final de su *statement*. En otras palabras, si se extraen múltiples *statements*, deben extraerse completos.

Respecto al objeto *applier* (responsable de generar el contexto necesario para aplicar el refactoring), éste está representado por instancias de la clase `ExtractMethodApplier`, subclase de `RefactoringApplier`. Las dos principales responsabilidades de este objeto son: (a) la de solicitar la información necesaria a quien inició el refactoring (en este caso, el nombre del nuevo mensaje) y (b) instanciar el refactoring, aplicarlo y manejar eventuales excepciones. El diseño estilo *template method* en la clase `RefactoringApplier` permite sólo redefinir aquello que es particular en este refactoring. La figura 3.j muestra los tres métodos más importantes de la clase `ExtractMethodApplier`: el que solicita los

parámetros del refactoring; el que se encarga de instanciar el refactoring y el que refleja los cambios aplicados.

```
ExtractMethodApplier >> requestRefactoringParameters

  | parseNodesToParameterize initialAnswer userAnswer |
  parseNodesToParameterize := self parseNodesToParameterize.
  initialAnswer := self buildInitialSelectorAnswer: parseNodesToParameterize.
  userAnswer := self request: 'New method name:' initialAnswer: initialAnswer.

  parseNodesToParameterize
    ifEmpty: [ self saveUnarySelector: userAnswer ]
    ifNotEmpty: [ self saveBinaryOrKeywordSelector: userAnswer withArguments:
parseNodesToParameterize ]
```

```
ExtractMethodApplier >> createRefactoring

  ^ self refactoringClass
      fromInterval: intervalToExtract
      of: methodToExtractCodeFrom
      to: self buildNewMessage
      categorizedAs: methodToExtractCodeFrom category
```

```
ExtractMethodApplier >> showChanges

      codeProvider currentMethodRefactored
```

Fig. 3.j - Métodos más importantes de la clase `ExtractMethodApplier`

La sugerencia del nombre de mensaje depende de la cantidad de parámetros necesarios (un dato que proporciona la clase `ExtractMethodParametersDetector`); si no es necesario parámetros se sugiere un mensaje unario (ver figura 3.b), si se necesita uno, se sugiere un mensaje de palabra clave que puede ser cambiado a mensaje binario si así lo desea el usuario; si son necesarios múltiples parámetros se sugerirá un mensaje de palabra clave del estilo `#m1:m2:...` con tantas palabras claves como objetos a parametrizar (ver figura 3.c).

Para poder mostrar los mensajes de error o advertencias no fue necesaria ninguna implementación adicional, todo ese código ya estaba en la clase `RefactoringApplier`.

3.4. Testeo

Este refactoring posee 41 tests (al día 30/07/2020) que verifican diferentes casos en los que es posible o no es posible realizar el refactoring. Los tests son parte del *package*

`BaseImageTests`[9] que contiene todos los tests del núcleo de Cuis. Estos tests se ejecutan en el servidor de integración continua de Cuis[10].

De estos 41 tests, 18 representan casos exitosos, es decir, en donde se puede aplicar el refactoring:

- Extraer un único objeto literal.
- Extraer más de un *statement* a un método que no incluya retorno.
- Extraer un *string* que contenga el caracter de retorno (^): este test es para asegurar que la validación por expresión de retorno no detecta un falso positivo como sería en este caso, y que se busca por AST en lugar de por texto.
- Al extraer múltiples *statements*, si la selección contiene puntos al inicio o al final (separando otros *statements*), no se toman como parte del refactoring y continúa exitosamente.
- Extraer expresiones encerradas entre múltiples paréntesis.
- Una ocurrencia de una variable local dentro del texto seleccionado para extracción debe ser parametrizada.
- Extraer expresiones complejas en las que algunas partes contienen paréntesis.
- Extraer la última expresión de un método.
- Extraer un bloque completo, en el que se incluya una declaración y asignación a una variable local a ese bloque.
- Extraer una expresión que incluya mensajes optimizados por el compilador, lo cual es relevante porque dicha optimización afecta la generación de *complete source ranges* para dichos envíos de mensaje.
- Al extraer una expresión que contiene un mensaje binario, a un mensaje de palabra clave, se introducen paréntesis para no alterar el orden de evaluación.
- Extraer una expresión encerrada entre *backticks*.
- Extraer expresiones con mensajes optimizados por el compilador en las que el receptor del mensaje no sea un simple literal.
- Extraer una declaración de variable temporal dentro de un bloque siempre y cuando no se esté utilizando fuera del intervalo de extracción elegido.
- Extraer una declaración de variable temporal a nivel método, siempre y cuando no se esté utilizando fuera del intervalo de extracción elegido.
- Extraer una expresión que involucra mensajes en cascada.
- Extraer una declaración de variable temporal junto con un bloque.
- Extraer código que está encerrado entre múltiples paréntesis y potenciales espacios entre ellos.

Los 23 casos restantes, se refieren a casos en donde no se puede aplicar el refactoring, ya que alguna precondición no se cumple:

- Validar que el nuevo nombre de mensaje no puede ser vacío.
- Validar que el nuevo nombre de mensaje no contiene espacios ni ningún caracter separador.

---

[9] Package `BaseImageTests`: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/blob/master/Packages/BaseImageTests.pck.st

[10] Servidor de integración continua de Cuis: https://travis-ci.org/github/Cuis-Smalltalk/Cuis-Smalltalk-Dev

- Extraer utilizando un mensaje que ya está definido en la clase actual lanza un *warning*.
- Validar que el nuevo nombre de mensaje no puede comenzar con un número.
- Validar que el nuevo nombre de mensaje no puede empezar con un símbolo.
- Validar que el intervalo de extracción no comienza en un número negativo.
- Validar que el intervalo de extracción no finaliza más allá de la longitud del código fuente de origen.
- Validar que no se puede extraer una expresión de retorno.
- Validar que la selección de código no incluya la cabecera del método.
- Validar que la selección de código no incluya parte de la declaración de una variable temporal.
- Validar que no se pueden extraer pragmas de un método (esto incluye, por ejemplo, primitivas).
- Validar que la selección de código contiene una expresión sin errores de sintaxis.
- Validar que no es posible extraer el lado izquierdo de una asignación.
- Validar que no se puede extraer parte de un objeto literal.
- Validar que el intervalo seleccionado de código sea válido, con un lugar de inicio anterior al de final.
- Validar que no es posible extraer sólo la asignación de una variable temporal.
- Validar que el nuevo nombre de mensaje no contiene caracteres especiales no permitidos.
- Validar que no es posible extraer una variable local dentro de la sección que la define.
- Validar que no se puede extraer parametrizando menos objetos que los necesarios.
- Validar que no se puede extraer parametrizando más objetos que los necesarios.
- Validar que extraer a un método que ya está definido en una superclase, se lanza un *warning*.
- Validar que no se puede extraer la declaración y asignación de una variable temporal si ésta se utiliza más allá del intervalo de código seleccionado.
- Validar que no se puede extraer un colaborador externo que forma parte de la definición de un método.

Los tests requieren un importante uso de metaprogramación: para ejercitar correctamente el refactoring debe existir una clase de prueba en la que se aplique, y que sólo valga durante la ejecución de un test. De esta manera aseguramos que el test corre en un contexto aislado y no afecta a otros tests y/o partes del sistema. Dicha clase de prueba se puede crear a través de la clase `DynamicallyCodeCreationTest` que es superclase de todas las clases de refactorings, en este caso `ExtractMethodTest`, y que provee mensajes como `#createClassNamed:` o `#createClassNamed:subclassOf:`, que se encargan de crear clases en una categoría de prueba que se crea antes de ejecutar el test en el paso de `#setUp`, y se elimina completamente después de ejecutar el test en el paso de `#tearDown`.

La figura 3.k contiene un ejemplo de test sobre un caso exitoso. Con el fin de evitar complejidad en el test (y repetición de pasos similares en otros test) y que sea comprensible, se agregó una aserción personalizada que verifica las dos poscondiciones del refactoring: la actualización del método existente, y la definición del método nuevo con el nombre y parámetros dados.

```
ExtractMethodTest >>
test38ItIsPossibleToExtractCodeThatContainsMultipleParenthesisWithSpacesBetweenT
hem

  | codeToExtract newMethodCode originalCode updatedCode |
  codeToExtract := '(  (3 + 4))'.
  originalCode := 'm1 ^ ' , codeToExtract.
  newMethodCode := 'm2

  ^ 3 + 4'.
  updatedCode := 'm1 ^ (  (self m2))'.

  self
    assertExtracting: codeToExtract from: originalCode
    named: (Message selector: #m2)
    defines: newMethodCode andUpdates: updatedCode
```

Fig. 3.k - Ejemplo de un test sobre un caso exitoso de *Extract Method*

La figura 3.l contiene un ejemplo de test sobre un caso en el que se verifica que una validación impide que se aplique el refactoring. De la misma manera que en los casos exitosos, fue necesario valerse de aserciones personalizadas para que el test sea lo más conciso y declarativo posible.

```
ExtractMethodTest >>
test34TryingToExtractAMethodWithLessArgumentsThanNeededFails

  self
    tryingToExtract: 'localVar1 + localVar2 + 2'
    from: 'm1 | localVar1 localVar2 | ^ localVar1 + localVar2 + 2'
    using: (Message selector: #m1: arguments: #('localVar1'))
    failsWith: [ ExtractMethod wrongNumberOfArgumentsGivenErrorMessage ]
```

Fig. 3.l - Ejemplo de un test que prueba un error de validación de *Extract Method*

4. Refactoring: *Extract Variable*

Como se menciona en la sección 2.1.2, el objetivo del refactoring *Extract Variable* es mover código que forma parte de una expresión Smalltalk (puede ser un único objeto, colaboración o conjunto de colaboraciones) a una nueva variable temporal. Esto permite descomponer expresiones complejas en partes, favorecer la reutilización y en algunas ocasiones, es el paso inicial de una serie de refactorings (continuado, por ejemplo, por un *Extract Method*).

4.1. Funcionalidad

El refactoring necesita los siguientes datos de entrada:

- Método desde el que se inicia el refactoring.
- Intervalo de código del método en el cual deseamos realizar la extracción.
- Nombre de la variable temporal a introducir.

En caso de poder aplicarse, se genera el siguiente resultado:

- Método original modificado con:
    - La declaración de la variable temporal nueva.
    - La asignación de la variable temporal nueva, cuyo valor es la expresión seleccionada.
    - El uso de la variable temporal nueva, omitido en el caso que se extraiga un *statement* entero.

Se realizan las siguientes validaciones:

- Respecto a la variable:
    - Tiene nombre válido (comienza con minúsculas, no contiene espacios, separadores ni caracteres especiales).
    - No está actualmente definida (ya sea como variable temporal, de instancia, parámetro u objeto globalmente conocido).
- Respecto a la selección de código:
    - No contiene múltiples *statements*.
    - Es una expresión Smalltalk completa (es decir, que puede ser parseada).
    - No es una asignación, ni una declaración de variable ni una expresión de retorno.
    - No forma parte de la cabecera del método.

4.2. Interfaz de usuario

El refactoring se inicia desde un Browser de Cuis, ya sea utilizando el menú contextual (ver figura X) o un atajo de teclado (Ctrl+Shift+j en Windows o GNU/Linux; Command+Shift+j en Mac). La figura 4.a muestra la opción del menú contextual agregada.

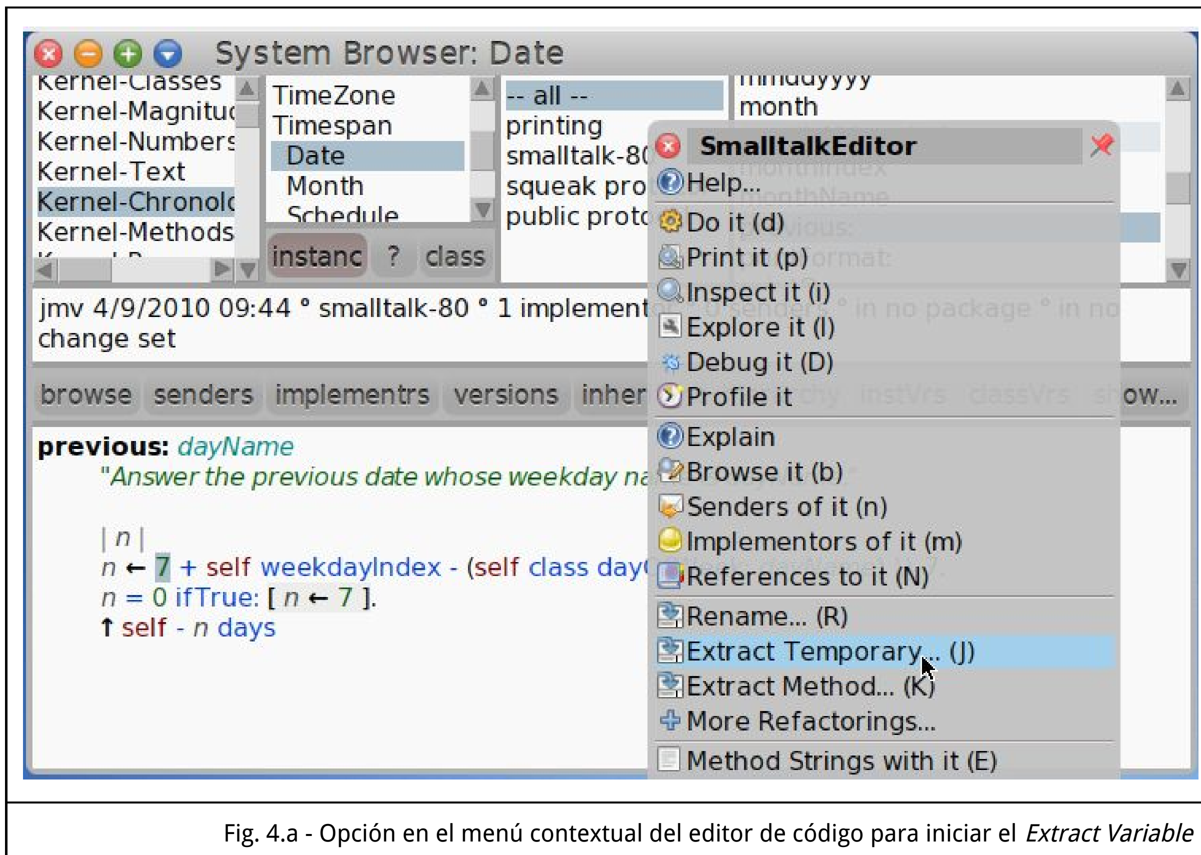


Fig. 4.a - Opción en el menú contextual del editor de código para iniciar el *Extract Variable*

Al seleccionar el código deseado para extracción y presionar el atajo de teclado o la opción de menú, el próximo paso será elegir el nombre de la variable temporal nueva. La figura 4.b muestra el cuadro para ingresar el nombre de la variable nueva.

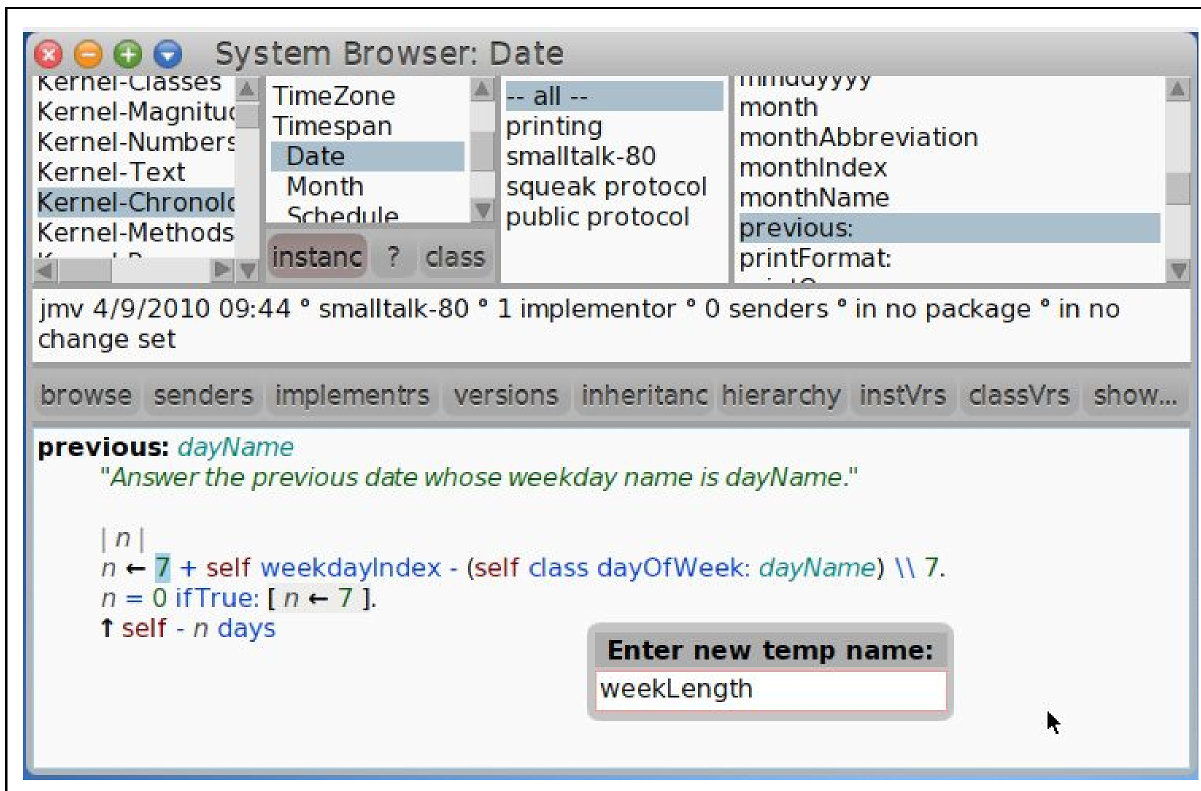


Fig. 4.b - Elección del nombre de variable temporal a utilizar aplicando *Extract Variable*

Si el refactoring puede aplicarse acorde a las validaciones configuradas, se modificará el método agregando la nueva declaración de variable temporal, y su correspondiente asignación y uso como indica la figura 4.c.

Como puede apreciarse, también existen otros lugares donde se debe reemplazar `7` por `weekLength`; el refactoring actualmente no los contempla, así que este cambio debe hacerse de manera manual por el momento. Está considerado en el trabajo futuro agregar esta funcionalidad (ver sección 7.2.2), dada su utilidad.

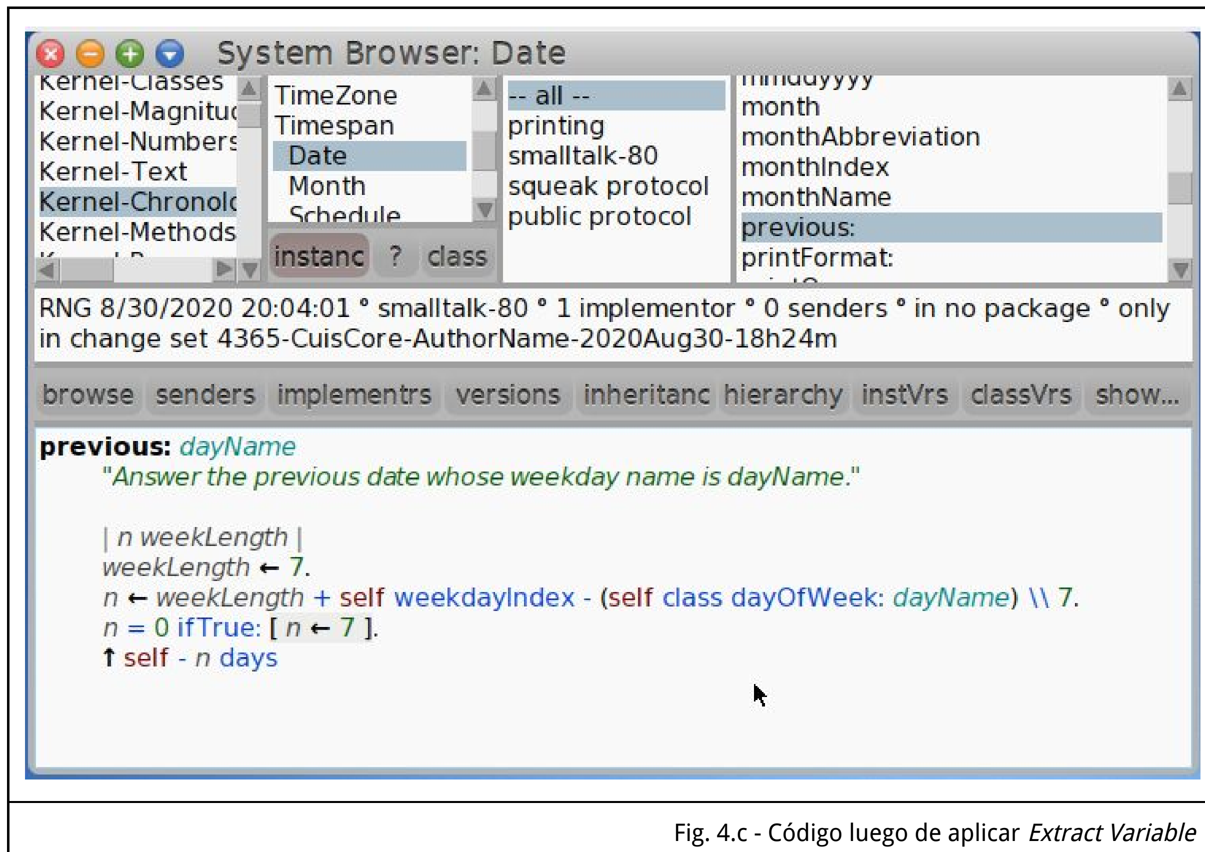


Fig. 4.c - Código luego de aplicar *Extract Variable*

En caso de ocurrir un error se mostrará una ventana correspondiente (ver figura 4.d), no se aplicará el refactoring y se regresará al editor de código.

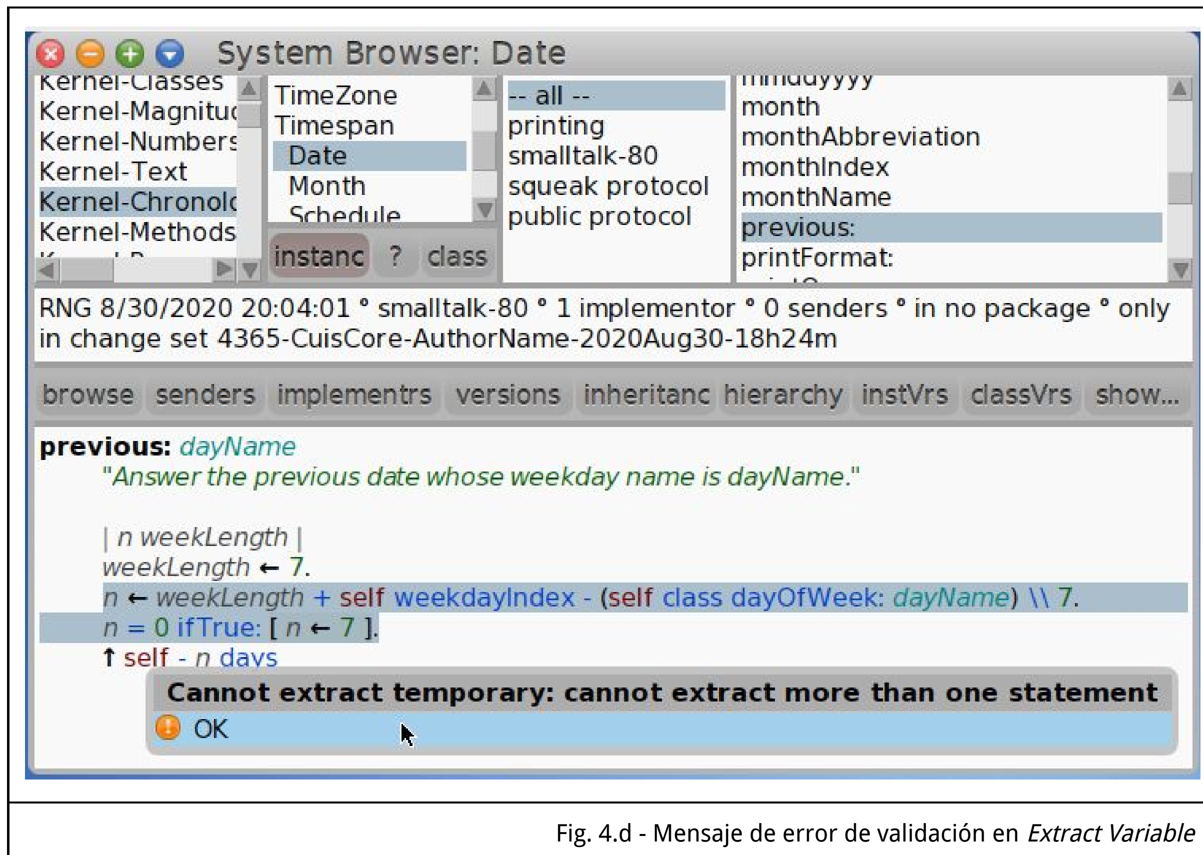


Fig. 4.d - Mensaje de error de validación en *Extract Variable*

4.3. Implementación

La lógica propiamente dicha del refactoring está en la clase `ExtractToTemporary`. La figura 4.e muestra la definición de la clase. La clase posee siete colaboradores internos:

- `newVariableName`: el nombre elegido para la nueva variable temporal, instancia de `String`.
- `methodToRefactor`: el método desde donde se inicia el refactoring, instancia de `CompiledMethod`.
- `methodNodeToRefactor`: el nodo del AST correspondiente al método a refactorizar, se obtiene a partir de `methodToRefactor`, y se precalcula porque se va a utilizar en varias ocasiones. Instancia de `MethodNode`.
- `updatedSourceCode`: el código fuente que inicialmente es el el original, pero que se va modificando a medida que se aplica el refactoring, y finalmente es el código que debe quedar luego de aplicar todos los pasos. Instancia de `String`.
- `intervalToExtract`: intervalo de código elegido para realizar la extracción, instancia de `SourceCodeInterval`.
- `sourceCodeToExtract`: el código fuente representado por `intervalToExtract`, y que va a formar parte de la asignación luego de finalizar el refactoring, instancia de `String`.
- `parseNodeWithNewVariableScope`: el nodo del AST sobre el cual se debe definir la nueva variable; dependiendo el caso, puede ser una instancia de `MethodNode` o de `BlockNode`.

Definición de clase
`Refactoring subclass: #ExtractToTemporary instanceVariableNames: 'newVariableName methodNodeToRefactor methodToRefactor updatedSourceCode intervalToExtract sourceCodeToExtract parseNodeWithNewVariableScope' classVariableNames: '' poolDictionaries: '' category: 'Tools-Refactoring'`
Comentario
*I am a refactoring that extracts a selected piece of code to a temporary variable. The input is the following: * interval of code to extract (from index - to index) * the CompiledMethod where this change applies * the new temporary variable name Many conditions have to be satisfied for this refactoring to be made, I delegate into SourceCodeOfTemporaryToBeExtractedPrecondition and NewTemporaryPrecondition most of these checks. Refer to those classes' comments for more information.*
Fig. 4.e - Definición de la clase `ExtractToTemporary` y su comentario

El único mensaje público de instancia requerido, como en la mayoría de los refactorings, es `#apply`, que en este caso debe realizar cuatro pasos:

1. Escribir la nueva asignación y su uso (reemplazar la expresión a extraer por la nueva variable, y en una línea previa, ubicar la asignación).
2. Escribir la declaración de la nueva variable.
3. Recompilar el método a refactorizar con el nuevo código fuente producto de las modificaciones de los dos pasos anteriores.
4. Retornar el código fuente. Este paso no es estrictamente necesario, aunque varios refactorings lo realizan. Es para facilitar la integración del refactoring con el editor de texto.

En la figura 4.f se ve la definición de `#apply` y sus tres principales pasos.

```
ExtractToTemporary >> apply

  self
    resolveNewVariableAssignment;
    declareNewTemporaryVariable;
    reflectSourceCodeChanges.

  ^ updatedSourceCode
```

```
ExtractToTemporary >> resolveNewVariableAssignment
```

```
  self extractingAnEntireStatement
    ifTrue: [ self addAssignmentToCurrentStatement ]
    ifFalse: [
      self
        replaceExtractedCodeWithNewTemporaryVariable;
        writeAssignmentStatementOfNewTemporaryVariable ]
```

```
ExtractToTemporary >> declareNewTemporaryVariable

  self hasTemporariesDeclarationBlock
    ifTrue: [ self addNewTemporaryVariableToExistingDeclarationStatement ]
    ifFalse: [ self insertNewTemporaryDeclarationWithNewVariable ]
```

```
ExtractToTemporary >> reflectSourceCodeChanges

  methodToRefactor methodClass
    compile: updatedSourceCode
    classified: methodToRefactor category
```

Fig. 4.f - Definición del método `#apply` de `ExtractToTemporary` y sus tres principales pasos

Previo a los mensajes de instancia, durante la fase de construcción de las instancias de `ExtractToTemporary` se valida que el refactoring pueda realizarse, de esta manera se asegura que el refactoring una vez instanciado puede aplicarse con seguridad. La figura 4.g muestra el mensaje de clase utilizado para instanciar el refactoring, donde previo a la construcción e inicialización de la instancia se realizan las validaciones necesarias.

```
ExtractToTemporary class >> named: aNewVariable at: anIntervalToExtract from:
aMethodToRefactor

  | trimmedNewVariable trimmedIntervalToExtract codeNodeForNewVariable
methodNodeToRefactor |

  self assertCanApplyRefactoringOn: aMethodToRefactor at: anIntervalToExtract.
  methodNodeToRefactor := aMethodToRefactor methodNode.
  trimmedNewVariable := aNewVariable withBlanksTrimmed.
  trimmedIntervalToExtract := anIntervalToExtract trimToMatchExpressionOn:
aMethodToRefactor sourceCode.
  codeNodeForNewVariable := self methodOrBlockNodeIncluding: anIntervalToExtract
in: methodNodeToRefactor.
  self newTemporaryPreconditionClass valueFor: trimmedNewVariable in:
codeNodeForNewVariable of: methodNodeToRefactor.

  ^ self new
      initializeNamed: trimmedNewVariable
      extractingCodeAt: trimmedIntervalToExtract
      from: aMethodToRefactor
      declaringTempIn: codeNodeForNewVariable
```

Fig. 4.g - Definición del método de creación de instancias de `ExtractToTemporary`

Las validaciones que se realizan son las siguientes:

- El nuevo nombre de la variable temporal es válido (para lo cual se delega en la clase `NewTemporaryPrecondition`).
- El código fuente a extraer no está vacío, y el intervalo de código no está fuera de los límites del código fuente del método donde se inicia el refactoring.
- El intervalo de código seleccionado contiene código válido para ser asignado en una variable temporal nueva (para lo que se delega en la clase `SourceCodeOfTemporaryToBeExtractedPrecondition`).

En caso que alguna de estas validaciones falle, se lanzará un error de tipo `RefactoringError` impidiendo continuar con el refactoring. La figura 4.h detalla todos los casos posibles de error, con el mensaje correspondiente.

Caso de error	Mensaje de error
La variable que se intenta definir ya existe como variable de instancia en la jerarquía del método a refactorizar	`'myVariable cannot be used as a temporary variable name because it is defined as an instance variable in SomeClass'`
La variable que se intenta definir es vacía	`'New variable can not be empty'`
La variable que se intenta definir contiene caracteres inválidos	`'myVariable is not a valid temporary variable name'`
La variable que se intenta definir ya existe como variable temporal o parámetro dentro del método a refactorizar	`'myVariable is already defined in SomeClass>>methodUnderRefactor'`
La variable que se intenta definir es una nombre reservado en el ambiente	`'myVariable can not be used as temporary variable name because it is a reserved name'`
El código que se intenta extraer contiene errores de sintaxis	`'Can not extract a source code with syntax error: <message>'`
El código que se intenta extraer contiene más de un *statement*	`'Cannot extract temporary: cannot extract more than one statement'`
Intervalo de selección está por fuera de las dimensiones del código fuente	`'The requested source code selection interval is out of bounds'`
Intervalo de selección de código es vacío	`'Source code to extract can not be empty'`
Intervalo de selección incluye una expresión con un caracter de retorno	`'Cannot extract temporary: the selected code includes a return statement'`
Intervalo de selección incluye código que no representa una expresión completa	`'Cannot extract temporary: the selected code contains an invalid expression'`

Intervalo de selección incluye parte de la declaración de variables temporales	`'Cannot extract temporary: it is not possible to extract temporary variable definitions'`

Fig. 4.h - Errores que pueden generarse como parte de la ejecución de `ExtractToTemporary`

Respecto al objeto *applier*, `ExtractToTemporary` se integra de la misma manera que ExtractMethod y el resto de refactorings del sistema, en este caso a través de una subclase de `RefactoringApplier`. denominada `ExtractToTemporaryApplier`. La figura 4.i muestra los tres métodos más importantes de la clase `ExtractToTemporaryApplier`: el que solicita los parámetros del refactoring; el que se encarga de instanciar el refactoring y el que refleja los cambios aplicados.

```
ExtractToTemporaryApplier >> requestRefactoringParameters

  self askNewVariableName
```

```
ExtractToTemporaryApplier >> createRefactoring

  ^ self refactoringClass
      named: newVariable
      at: intervalToExtract
      from: methodToExtractCodeFrom
```

```
ExtractToTemporaryApplier >> showChanges

  codeProvider currentMethodRefactored
```

Fig. 4.i - Métodos más importantes de la clase `ExtractToTemporaryApplier`

4.4. Testeo

Este refactoring posee 24 tests (al día 30/07/2020) que verifican diferentes casos en los que es posible o no es posible realizar el refactoring. Al igual que los tests de *Extract Method*, los tests son parte del *package* `BaseImageTests` y se ejecutan en el servidor de integración continua de Cuis.

De estos 24 tests, 9 representan casos exitosos, es decir, en donde se puede aplicar el refactoring:

- Extraer un único literal de un método sin ninguna otra variable temporal o parámetro.
- Extraer una expresión de un método que ya contiene un bloque de declaración de variables temporales.
- Extraer una expresión de un método que contiene un bloque de declaración de variables temporales vacío.
- Extraer una expresión generando la asignación usando la sintaxis de *ANSI assignment* (denotada por los caracteres `:=`).
- Extraer un único literal que está dentro de un bloque de código.

- Extraer habiendo seleccionado un intervalo que contiene espacios adicionales al inicio y al final.
- Extraer un bloque vacío.
- Extraer un envío de mensajes en cascada.
- Cuando se extrae un *statement* entero, sólo se genera la asignación y no el uso de la variable nueva.

Los 15 casos restantes, se refieren a casos en donde no se puede aplicar el refactoring, ya que alguna precondición no se cumple:

- El nombre de la variable temporal a introducir no puede ser vacío.
- El nombre de la variable temporal a introducir tiene que ser válido (por ejemplo, no contener un espacio en blanco).
- El nombre de la variable temporal a introducir no puede estar definido previamente.
- No puede haber una variable de instancia, en la clase donde se origina el refactoring, con el mismo nombre que la variable que se desea introducir.
- No puede haber una variable de instancia, en una superclase de la clase donde se origina el refactoring, con el mismo nombre que la variable que se desea introducir.
- El código a extraer no puede contener una expresión de retorno.
- El código a extraer no puede ser vacío.
- El código a extraer no puede tener errores de sintaxis, por representar una expresión incompleta.
- El código a extraer no puede abarcar más de un *statement*.
- El intervalo de selección es inválido debido a que comienza previo a los límites del código fuente.
- El intervalo de selección es inválido debido a que finaliza más allá de los límites del código fuente.
- El código a extraer no puede contener parte de la cabecera del método.
- No se puede extraer parte de un envío de mensaje.
- No se puede extraer el lado izquierdo de una asignación.
- No se puede utilizar un nombre reservado como nombre de la variable temporal nueva (por ejemplo, `self`).

Los tests requieren el mismo contexto de creación de clases y métodos que el *Extract Method*. La clase **`ExtractToTemporaryTest`**, que contiene todos los tests mencionados previamente, es subclase de **`RefactoringTest`** que a su vez es subclase de **`DynamicallyCodeCreationTest`**.

La figura 4.j contiene un ejemplo de test sobre un caso exitoso, el cual delega en una aserción personalizada para reducir la cantidad de código repetido en cada test.

```
ExtractToTemporaryTest >> test22ItIsPossibleToExtractACascadeExpression

  | sourceCode sourceCodeAfterRefactoring |
  sourceCode := 'm1

  ^ 3 factorial; yourself'.
```

```
  sourceCodeAfterRefactoring := 'm1

  | new |
  new := 3 factorial; yourself.
  ^ new'.

  self assertExtracting: '3 factorial; yourself' from: sourceCode
toVariableNamed: 'new' updatesTo: sourceCodeAfterRefactoring
```

Fig. 4.j - Ejemplo de un test sobre un caso exitoso de *Extract Variable*

La figura 4.k contiene un ejemplo de test sobre un caso en el que se verifica que una validación impide que se aplique el refactoring. De la misma manera que en los casos exitosos, fue necesario valerse de aserciones personalizadas para que el test sea lo más conciso y declarativo posible.

```
ExtractToTemporary >> test23CannotUseAReservedNameAsTheNewTemporaryVariable

  | intervalToExtract methodToRefactor newVariable sourceCode |

  ClassBuilder reservedNames do: [ :reservedName |
    newVariable := reservedName asString.
    sourceCode := 'm1 ^ 2'.
    classToRefactor compile: sourceCode.
    intervalToExtract := self intervalOf: '2' locatedIn: sourceCode.
    methodToRefactor := classToRefactor >> #m1.

    self
      assertCreation: [ ExtractToTemporary named: newVariable at:
intervalToExtract from: methodToRefactor ]
      failsWith: [ NewTemporaryPrecondition
errorMessageForNewTemporaryVariableCanNotBeAReservedName: newVariable ] ]
```

Fig. 4.k - Ejemplo de un test que prueba un error de validación de *Extract Variable*

5. Cambios y mejoras necesarias para la implementación de refactorings

5.1. Modelo de refactorings

5.1.1. Comentarios de clases

Como parte del desarrollo del trabajo, se agregaron comentarios a algunas clases del modelo de refactorings: clases principales como `Refactoring` y `RefactoringPrecondition`; y clases de refactorings puntuales o auxiliares (entre los que están incluidos los refactorings de este trabajo). El propósito de dichos comentarios es resumir el propósito de dichas clases, un protocolo público, y en el caso de los refactorings puntuales, explicitar sus precondiciones y poscondiciones. La figura 5.a muestra los comentarios que fueron agregados para algunas de las clases del modelo de refactorings.

Comentario de la clase `Refactoring`

```
I am a refactoring, a code transformation preserving behavior, based on some
input (provided from the end user through a RefactoringApplier; or provided
programmatically). Instances of me have usually only public method, #apply,
which does all the work.

In case the refactoring cannot be made, or there is a problem during the
application of it, I can throw errors using the class message
#refactoringError:, or warnings using the class message #refactoringWarning:
```

Comentario de la clase `RefactoringPrecondition`

```
I represent a precondition, a prerequisite for a refactoring to be evaluated
successfully. My public instance protocol includes only one message, #value,
which could raise either a RefactoringError (in case the refactoring cannot be
performed) or a RefactoringWarning (in case something needs the programmer's
attention, but it can be resumed to continue with the refactoring).
```

Fig. 5.a - Ejemplos de comentarios agregados a clases del modelo de refactorings

5.1.2. Nuevas precondiciones

Cuando más de un refactoring comparte una precondición similar, es deseable que ésta esté representada en un objeto `RefactoringPrecondition`, en lugar de código duplicado en cada uno de los refactorings que la utilizan.

Este fue el caso de la precondición para definir una variable temporal nueva. Esta precondición es compartida por el refactoring de *Rename Temporary* (ya existente en Cuis), y el de *Extract Variable* (parte de este trabajo).

Como parte del *changeset* #4048 se creó el objeto `NewTemporaryVariablePrecondition`, y se cambió el refactoring *Rename Temporary*

para que utilice esta precondición, como así también *Extract Variable*. Esto significa que además del reuso de código y poder representar las validaciones sin duplicarlas, los mensajes de error que se terminan mostrando al usuario también resultan consistentes.

5.2. Agregados a Cuis Smalltalk

Durante el desarrollo del trabajo, surgieron posibles mejoras al ambiente de programación, algunas directas y necesarias para el desarrollo de los refactorings, otras auxiliares que simplemente surgieron del trabajo habitual con Cuis. La filosofía de código abierto y de *ownership* del código de Cuis hicieron que los cambios propuestos sean discutidos e integrados en tiempo y forma, y no generaran ningún bloqueo durante la realización del trabajo.

5.2.1. *Source ranges* completos

Los *source ranges* reportados por el `Parser` (denominados *raw* y que se obtienen enviando el mensaje `#rawSourceRanges` a un `MethodNode`) no necesariamente indican con exactitud los límites del código que representan. Por ejemplo, el *raw source range* de un nodo que representa un envío de mensaje, `MessageNode`, comienza donde comienza el mensaje a enviarse y termina con la última posición del último parámetro; es decir, no comienza en la posición del objeto receptor de dicho mensaje.

Para los refactorings esto es una complicación; deseamos coincidir selección de código con *source ranges* válidos, ya que esta validación hace que se pueda comprender qué semántica hay detrás de una selección de código.

Dentro de este trabajo se incluye una definición de *complete source range*, que a diferencia del *raw source range*, siempre abarca expresiones Smalltalk válidas. En la mayoría de los casos el *raw source range* y el *complete source range* coinciden (como en el caso de los objetos literales), pero hay casos en donde necesitan ser diferentes, como el envío de mensaje, la asignación, el bloque de código y el mensaje en cascada. La figura 5.b muestra las similitudes y diferencias entre *raw source range* y *complete source range* para cada nodo del AST; aquellos no documentados no poseen diferencias.

Nodo del AST	Ejemplo de código	Código cubierto por *raw source range*	Código cubierto por *complete source range*
`LiteralNode`	`'hola'`	`'hola'`	`'hola'`
`MessageNode`	`3 raisedTo: 2`	`raisedTo: 2`	`3 raisedTo: 2`
`AssignmentNode`	`started := false`	`:= false`	`started := false`
`BlockNode`	`[ :elem | elem isArray ]`	`elem isArray`	`[ :elem | elem isArray ]`

Fig. 5.b - Diferencias entre *complete* y *raw source ranges*

5.2.2. *Source ranges* para *brace arrays*

La construcción sintáctica de los denominados *brace arrays*, que existe en Squeak y fue adoptada también por Cuis, y que se escribe de la siguiente manera: `{ objeto1 . objeto2 . objeto3 }` (siendo `objeto1`, `objeto2` y `objeto3` objetos cualquiera), no generaba *source ranges*, es decir que no podíamos saber en qué posición del código estaban. A través de un cambio en el `Parser` (*changeset* #3762), se agregó la posibilidad de reportar las posiciones en donde comienzan y terminan las definiciones de dichos objetos, lo cual era necesario para que puedan ser extraídos en métodos, o en variables temporales.

5.2.3. *Source ranges* para mensajes en cascada

Los mensajes en cascada están representados en el AST de Cuis como instancias de `CascadeNode`, las que referencian al nodo receptor del mensaje en cascada (llamado `receiver`), y una colección de instancias de `MessageNode` (`messages`) que representan cada mensaje de la cascada. Si bien se reportaban los *source ranges* de cada `MessageNode`, no se reportaba un *source range* para el `CascadeNode`. Luego de introducir un cambio en el `Parser` (*changesets* #4116 y #4117), se permitió reportar los *source ranges* de cada `CascadeNode` que pueda aparecer en un método. La figura 5.c muestra cómo se reportaban antes del cambio, y después, en su versión *raw* y *complete* (fig 5.d).

```
String >> editLabel: labelString

      TextModel new contents: self; openLabel: labelString
```

Elemento del código	Clase del Nodo AST	*Source range*
`contents: self`	`MessageNode`	`(40 to: 53)`
`openLabel: labelString`	`MessageNode`	`(56 to: 77)`

Fig. 5.c - Ejemplo de *source ranges* reportados antes de agregar soporte para mensajes en cascada

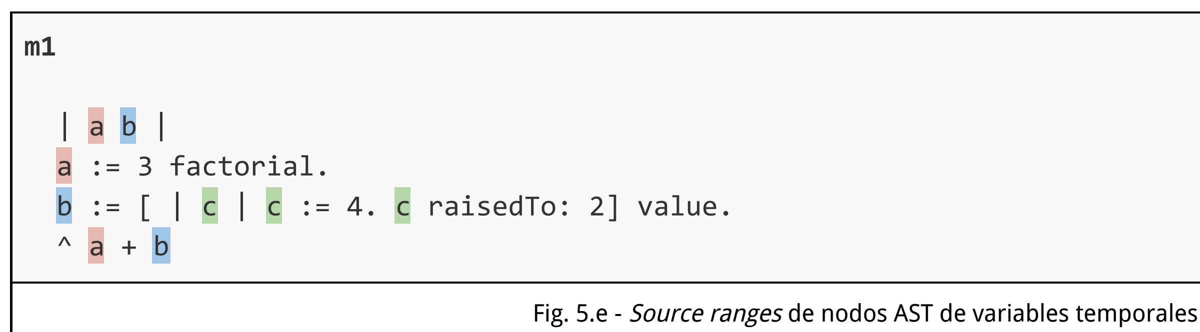
```
String >> editLabel: labelString

      TextModel new contents: self; openLabel: labelString
```

Elemento del código	Clase del Nodo AST	*Source range*
`contents: self`	`MessageNode`	`(40 to: 53)`
`openLabel: labelString`	`MessageNode`	`(56 to: 77)`
`contents: self; openLabel: labelString`	`CascadeNode`	`(40 to: 77) "raw"` `(26 to: 77) "complete"`

Fig. 5d - Ejemplo de *source ranges* reportados después de agregar soporte para mensajes en cascada

5.2.4. Nodos nuevos del AST para declaración de variables temporales, con sus respectivos *source ranges*

Para poder soportar casos de extracción de código que incluyan declaración de variables temporales (necesario para el *Extract Method*), y para crear variables temporales (necesario para el *Extract Variable*), se necesitaba saber en qué posición del código se encontraban las diferentes declaraciones de variables temporales (a nivel método, o en cada bloque que pueda contener el método). Esto no se podía ya que no había nodos del AST que representen “la porción de código que define variables temporales”. Hasta antes de este cambio, sólo sabíamos los *source ranges* de cada ocurrencia de las variables temporales, como en la fig. 5.e.



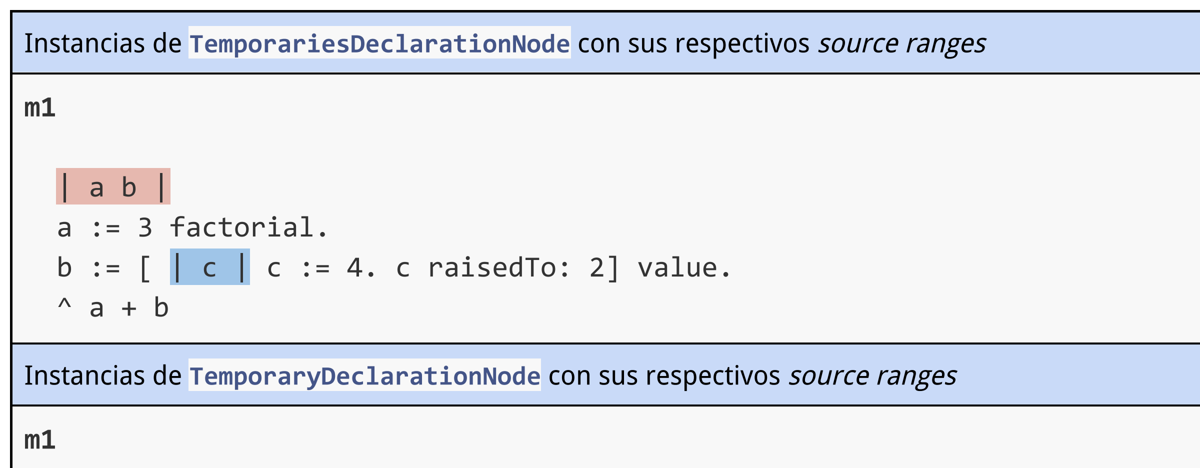
```
m1

  | a b |
  a := 3 factorial.
  b := [ | c | c := 4. c raisedTo: 2] value.
  ^ a + b
```

Fig. 5.e - *Source ranges* de nodos AST de variables temporales

Como parte del cambio se agregaron dos nuevos nodos del AST:

- `TemporaryDeclarationNode`: subclase de `ParseNode`, representa una declaración individual de una variable temporal. Se instancia con un `VariableNode`, al que después se puede acceder a través del mensaje `#variableNode`.
- `TemporariesDeclarationNode`: subclase de `ParseNode`, representa una declaración de variables temporales, es decir, el código que se escribe dentro de barras verticales. Conoce una colección de `#temporaryDeclarationNodes`, instancias de `TemporaryDeclarationNode`.

La figura 5.f muestra los *source ranges* en relación al código representado por estos dos nuevos nodos del AST.



Instancias de `TemporariesDeclarationNode` con sus respectivos *source ranges*

```
m1

  | a b |
  a := 3 factorial.
  b := [ | c | c := 4. c raisedTo: 2] value.
  ^ a + b
```

Instancias de `TemporaryDeclarationNode` con sus respectivos *source ranges*

```
m1
```

```
| a b |
a := 3 factorial.
b := [ | c | c := 4. c raisedTo: 2] value.
^ a + b
```

Fig. 5.f - *Source ranges* de nuevo nodos AST de declaración de variables temporales

Introducir este cambio no fue trivial, ya que hubo que introducir una modificación al parser de Cuis para que cree los nuevos nodos a medida que está parseando un método. Cualquier modificación al parser puede verse afectada por un problema de metacircularidad: como la clase `Parser` se encarga de parsear cualquier mensaje nuevo que se guarde en una imagen de Cuis, esto incluye también los mensajes de la propia clase `Parser`. Entonces, si necesitamos modificar el comportamiento de algunos mensajes, vamos a necesitar que el `Parser` actual pueda parsear el cambio que deseamos introducir. Hay dos opciones de evitar este inconveniente metacircular:

1. Definiendo un nuevo parser temporal, que no es más que una copia del `Parser` original (es común crear una subclase para lograr este propósito) y que sirva como opción de *backup* para realizar las modificaciones al parser original sin que la lógica actual de parseo de la imagen se vea afectada. Una vez introducido el cambio, se puede descartar esta clase de *backup*.
2. Haciendo modificaciones que soporten la lógica de parseo actual y la que se quiere introducir. Esto supone agregar código temporal que luego se puede remover una vez que todos los cambios al `Parser` están aplicados y funcionando.

La opción 2 fue la que se realizó para introducir el cambio de las variables temporales en el `Parser`. Para ello se realizaron 3 *changesets*, los cuales debían ser instalados en secuencia:

1. Agregar las clases `TemporaryDeclarationNode` y `TemporariesDeclarationNode` con su respectivo protocolo (*changeset* #4085).
2. Modificar el Parser para que comience a crear los nodos `TemporaryDeclarationNode` y `TemporariesDeclarationNode`, pero que también soporte la forma anterior de parseo de variables temporales (*changeset* #4086).
3. Eliminar la lógica temporal añadida en el *changeset* anterior, permitiendo así que de ahora en más todos los métodos parseados creen nodos de `TemporaryDeclarationNode` y `TemporariesDeclarationNode` (*changeset* #4087).

Luego, fue necesario extender el actual `ParseNodeVisitor` para que sea capaz de recorrer los nuevos nodos del AST; en este paso también hubo un problema metacircular que no permitía agregar todos los cambios de una vez, así que se crearon 2 *changesets* separados para:

1. Agregar protocolo de *visitor pattern* para los nodos de `TemporaryDeclarationNode` y `TemporariesDeclarationNode`, sin aún ser utilizados (*changeset* #4088).

2. Utilizar los mensajes previamente definidos en el momento de visitar instancias de `MethodNode` y `BlockNode` (los dos tipos de nodos que referencian a la declaración de variables temporales) (*changeset* #4089).

5.2.5. Reificación de intervalos de código

Como menciona la sección 2.1.4, los *source ranges* son instancias de la clase `Interval`, que tiene protocolo de colecciones (por ser subclase de `Collection`). A medida que los refactorings fueron avanzando en la implementación, fueron necesarias dos responsabilidades relacionadas al código fuente:

- Dado un *source range* que representa una selección dentro de un código fuente, poder reducir ese rango lo máximo posible hasta coincidir con una expresión Smalltalk válida. Esto implica, por ejemplo, ignorar espacios en blanco, paréntesis y puntos.
- Dado un *source range* que representa una selección dentro de un código fuente, poder expandir ese rango lo máximo posible sin que cambie la condición de expresión Smalltalk válida. Esto implica, por ejemplo, que se puede expandir paréntesis que encierran la expresión, como así también espacios en blanco.

Dichas responsabilidades no tuvieron un lugar natural, con lo cual el diseño fue evolucionando. Inicialmente, se ubicaron como mensajes auxiliares de instancia en el `ParseNode`, luego como mensajes auxiliares de clase en `Refactoring`. Pero tanto en `ParseNode` como en `Refactoring` las responsabilidades no formaban parte del protocolo esencial de dichas clases, sino que mantenían su condición de auxiliares. Allí fue cuando se diseñó el `SourceCodeInterval` (*changeset* #4156), que es subclase de `Interval`, y además agrega comportamiento para responder a las necesidades de expandirse o contraerse de acuerdo a un código fuente.

5.2.6. Cambios y mejoras auxiliares

Además de los cambios y mejoras necesarios para la realización del trabajo, también se realizaron algunas contribuciones adicionales que apuntan a mejorar la experiencia en general en Cuis, y la calidad del código.

- Atajos de teclado para navegar más fácilmente y realizar acciones sobre clases, métodos y categorías (*changesets* #3658 y #3660).
- Tareas de limpieza de código y refactorización en general (*changesets* #3657, #3692, #3725 y #4230).
- Soportar navegación entre los diferentes paneles del Browser utilizando las flechas izquierda y derecha (*changeset* #3673), esto fue realizado como parte de un evento al estilo hackatón de varios desarrolladores, llamado “Cuis Sprint”.
- Agregado de una sección al “Terse Guide”, la herramienta de ayuda práctica con ejemplos dentro del ambiente de Cuis[11] (*pull request* #161).
- Mejoras de navegación y experiencia de usuario en los inspectores (*changesets* #4030 y #4042).

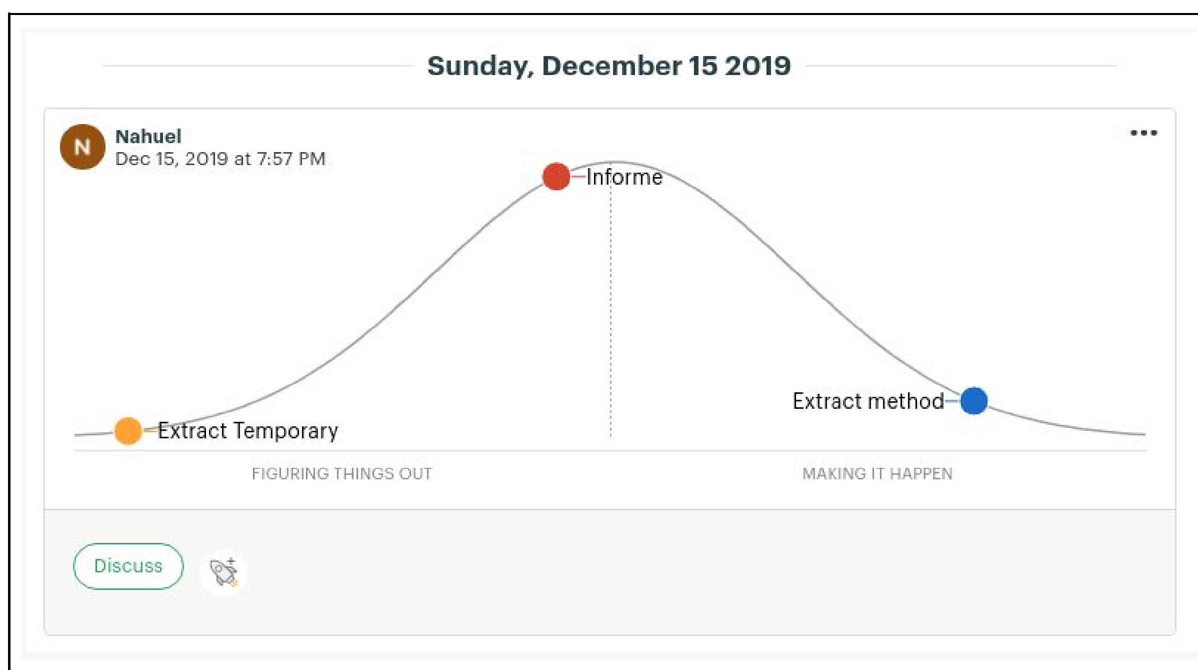
[11] *Terse Guide*: guía práctica con ejemplos: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/blob/master/Packages/TerseGuide.pck.st

6. Metodología y conceptos aplicados

6.1. Forma de trabajo

El trabajo fue realizado siguiendo métodos ágiles (Beck et.al. 2001). Aunque la implementación fue variando a lo largo del desarrollo, siempre se trató de seguir el espíritu de eXtreme Programming (Beck & Andres 2001). En los primeros dos meses de trabajo, se definieron iteraciones semanales con unos compromisos asociados, como lo que propone Scrum (Schwaber 1997); mientras que desde la mitad del trabajo hacia el final se construyó un único *backlog* constantemente priorizado, sin compromisos por iteraciones, más parecido a lo que propone Kanban (Anderson 2010).

A nivel herramientas, también hubo cambios a lo largo del trabajo. Inicialmente la herramienta de seguimiento utilizada fue Trello[12], luego el proyecto se migró a la herramienta Basecamp[13], que proveía un mejor seguimiento del trabajo por hacer, particularmente con el gráfico de *Hill Chart* para visualizar el progreso como se ve en la figura 6.a.



[12] https://trello.com/
[13] https://basecamp.com/

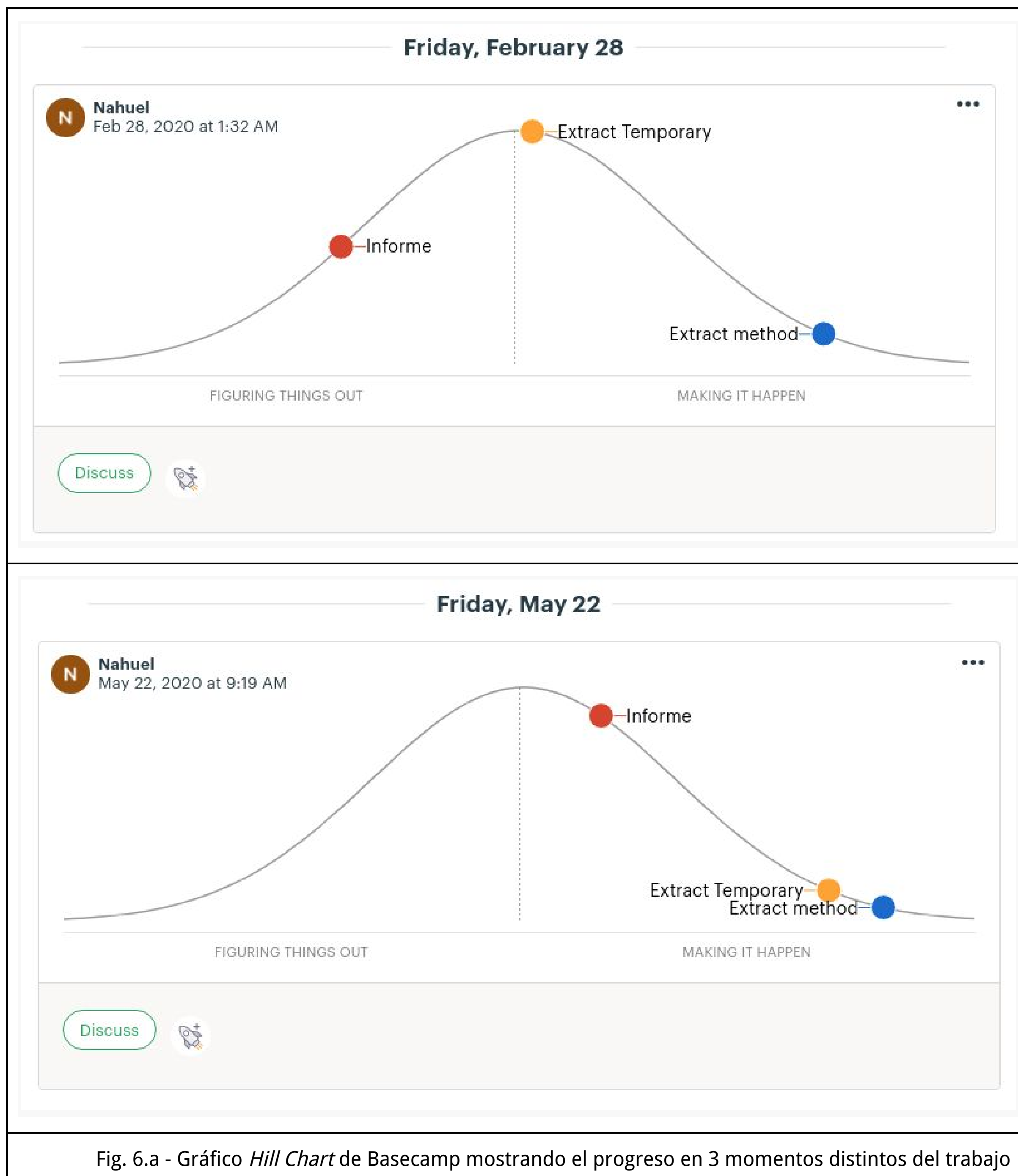


Fig. 6.a - Gráfico *Hill Chart* de Basecamp mostrando el progreso en 3 momentos distintos del trabajo

Lo que se mantuvo durante todo el trabajo fue la sincronización con los directores: salvo algunas excepciones, se organizaron reuniones semanales, las cuales servían para revisar el progreso hasta el momento, plantear dificultades, consultas o bloqueos; y finalmente definir una estrategia para continuar la próxima semana. El tipo de tareas que surgía de estas reuniones podía ser de investigación (como por ejemplo, utilizar otras distribuciones, o leer algún libro o publicación relacionada, o implementar una prueba de concepto para validar alguna hipótesis), de desarrollo (implementar una nueva funcionalidad o corregir errores) o de documentación (principalmente, la confección de este informe).

Esta reunión semanal incluía a varios estudiantes realizando sus respectivos trabajos finales de carrera, todos ellos relacionados de alguna manera a Cuis, lo cual propició la colaboración y la visión conjunta para definir cómo debían ser las contribuciones a Cuis.

6.2. Publicación

6.2.1. Versión beta o no oficial

Inicialmente, se divulgó un Package de Cuis (.pck.st instalable) en versión beta con el fin de recolectar feedback de parte de ciertos usuarios de Cuis, muchos de ellos alumnos de las siguientes universidades: Universidad de Buenos Aires (UBA), Universidad Nacional de Quilmes (UNQ), y Universidad Católica Argentina (UCA); como así también docentes y otros usuarios interesados.

Se realizó el mismo procedimiento para el *Extract Method* y para el *Extract Variable*. Esto sirvió para llegar con una versión más robusta y luego integrarla de manera definitiva a la distribución de Cuis.

6.2.2. Versión final para usuarios

Luego de recolectar feedback y realizar pruebas con casos reales de diferentes piezas de software, se procedió a integrar cada refactoring a la distribución principal de Cuis Smalltalk (lo que se suele denominar el *core*). Esto se realiza proponiendo uno o más *changesets*, dependiendo de la complejidad de los cambios.

Al momento de esta publicación los refactorings se encuentran disponibles en la distribución principal de Cuis para que cualquiera pueda utilizarlos.

6.2.3. Licencia

El trabajo está licenciado bajo MIT[14], que es la misma licencia que posee Cuis, y que garantiza las 4 libertades del software libre: usar el programa para cualquier propósito, estudiar cómo funciona el programa y poder modificarlo, distribuir copias del programa a cualquier persona u organización y mejorar el programa y compartir las mejoras en beneficio de todos.

6.3. Conceptos aplicados

6.3.1. Patrones y buenas prácticas de diseño

Uno de los objetivos del trabajo es lograr un diseño robusto, reutilizable y mantenible; no sólo que funcionalmente cumpla con lo esperado. Para ello, fueron claves las referencias a buenas prácticas conocidas de diseño, que aplicadas en los contextos adecuados contribuyen al buen estado respecto a requerimientos no funcionales.

Al trabajar en Smalltalk, resultaron útiles los patrones mencionados por Kent Beck en su libro *Smalltalk: Best Practice Patterns* (Beck 1996), siendo los más relevantes que aparecen a lo largo de este trabajo los siguientes:

[14] Licencia MIT. Disponible en https://opensource.org/licenses/MIT

- *Composed Method*: en ambas clases de refactoring (`ExtractMethod` y `ExtractToTemporary`) debido a la complejidad de la tarea, sus respectivos métodos `#apply` se compusieron de varios pasos, cada uno de ellos con un nombre descriptivo.
- *Constructor Method / Constructor Parameter Method*: todas las clases que necesitaban de objetos en su momento de creación, tienen sus mensajes de creación de instancias, que se utilizarán en alternativa al mensaje por defecto `#new`. No existen mensajes del tipo *setters* (mensajes para cambiar un colaborador interno por otro) a menos que respondan a una necesidad del dominio.
- *Shortcut Constructor Method*: la clase `SourceCodeInterval` es una clase que debe ser instanciada en varias ocasiones, y lo que se realizó para evitar nombrar a la clase y a un mensaje de creación de instancias, se definió el mensaje `#asSourceCodeInterval` que convierte una instancia de `Interval` en una instancia de `SourceCodeInterval`, lo cual resultó muy conveniente.
- *Method Object*: las precondiciones (como `NewTemporaryPrecondition`) y el objeto detector de parámetros `ExtractMethodParametersDetector` son buenos ejemplos de *Method Object*. Su complejidad y su necesidad de conocer un contexto con varios objetos ameritó la creación de estos objetos.
- *Enumeration Method*: dado que era algo frecuente iterar por los diferentes *complete source ranges* de un `ParseNode`, se agregó un método de enumeración `#completeSourceRangesDo:` con el fin de controlar el acceso a la colección de *source ranges*.
- *Role Suggesting Instance Variable Name*: en general, las variables de instancia que se definieron refieren al rol que cumple ese objeto en el contexto donde está, aunque quizás el ejemplo más evidente de esto es el caso del método a refactorizar en `ExtractMethod` y `ExtractToTemporary`. En ambos casos, la variable de instancia que representa a ese método se llama `methodToRefactor`.

En menor medida, los patrones de diseño según el libro de *Design Patterns* (Gamma et. al. 1995) también sirvieron de referencia: *Template Method* en `RefactoringApplier` y su subclase `ExtractMethodApplier`; *Visitor* para experimentar con refactoring de búsqueda de nodos para validación del *Extract Method* y para modificar el `ParseNodeVisitor` (*changesets* #4088 y #4089).

6.3.2. Test-Driven Development

Test-Driven Development fue la metodología utilizada desde el inicio hasta el final del trabajo, tanto en la implementación inicial como en el posterior mantenimiento. Cada vez que aparecía un error, el arreglo debía incluir el test que compruebe que el problema no aparece más.

Cabe hacer una distinción entre los objetos del modelo de dominio (como `ExtractMethod` o `ExtractToTemporary`) y los objetos de la interfaz gráfica (`ExtractMethodApplier`, `ExtractToTemporaryApplier`); estos últimos son testeados manualmente y no fueron desarrollados utilizando TDD, en el sentido que no existen al momento tests automatizados que verifiquen su funcionamiento.

Respecto al ciclo de TDD, en las etapas iniciales fue bastante largo (en algunos casos, entre una o dos horas), ya que los tests no son triviales de configurar, tanto en el *set up* como en las aserciones. A medida que se fueron agregando más tests, este ciclo se acortó ya que definir un nuevo test no difería demasiado de tests existentes, y se podían reutilizar aserciones definidas previamente. Esto permitió poder resolver errores o agregar casos de funcionalidad nueva, en ciclos completos de TDD en no más de 10 minutos.

6.3.3. Testing unitario vs. testing funcional

Uno de los primeros pasos en la implementación del *Extract Method* fue la separación de la lógica que representaba la precondición, del refactoring propiamente dicho. Erróneamente, al principio ambos objetos (`ExtractMethod` y `ExtractMethodPrecondition`) estaban cada uno testeados por separado. Esto hacía que no todos los casos de tests posibles del *Extract Method* no estén en su suite de tests, si no en la suite de tests de su clase de precondición. Luego de una mejora del diseño, se decidió mover todos los tests a la clase de test `ExtractMethodTest`. De esta manera, estamos testeando el refactoring desde un punto de vista puramente funcional, sin importar con cuántos objetos colabora y de qué manera lo hace.

6.3.4. Producto Mínimo Viable

Tanto para el *Extract Method* como para el *Extract Variable*, el foco del desarrollo estuvo en poder lograr una implementación lo suficientemente robusta para no causar problemas al utilizarla, pero lo suficientemente pronto en el tiempo, para ser integrada y utilizada sin tener la totalidad de los casos contemplados, en pos de obtener feedback para futuras mejoras; y que se mantenga dentro de los lineamientos de tiempo establecidos por la Universidad para la realización del Seminario Final. Una vez que los refactorings fueron integrados se fueron agregando casos no contemplados previamente, que no son esenciales, sí deseables. Esta distinción entre casos esenciales y accesorios terminó definiendo el Producto Mínimo Viable (MVP, por sus siglas en inglés) de cada refactoring.

6.3.5. eXtreme Programming (XP)

En la sección Forma de Trabajo (6.1) se menciona eXtreme Programming como guía durante la realización de este trabajo. XP describe valores (comunicación, simplicidad, feedback, coraje y respeto), prácticas primarias y prácticas corolarias.

Dentro de las prácticas primarias, en este trabajo se destacan tres:

- *Weekly Cycle*: las reuniones semanales de seguimiento marcaban el fin de un ciclo y el inicio de uno nuevo, tomando feedback para ser trabajado durante los próximos ciclos.
- *Test-First Programming*: el hecho de aplicar TDD nos asegura cumplir con la consigna de *test-first*.
- *Incremental Design*: en la mayoría de los componentes desarrollados el diseño fue emergente, y en cada iteración se fue mejorando ya que el aprendizaje respecto al dominio luego de cada etapa se iba acumulando y reflejando en el código.

Dentro de las prácticas corolarias, hay dos que toman relevancia para este trabajo:

- *Incremental Deployment*: el trabajo no fue publicado una sola vez, sino que constó (y consta) de sucesivos despliegues con pequeños incrementos de valor.
- *Root Cause Analysis*: la idea de escribir tests con el fin de detectar la causa raíz de errores (*bugs*) en la implementación.

6.3.6. Metaprogramación

Dado que los refactorings pertenecen al dominio de programas que manipulan programas, este trabajo es en su mayoría un problema de metaprogramación. Los usos más importantes de metaprogramación son:

- Para poder hacer consultas sobre la semántica de elementos dentro de un método, como por ejemplo: saber si en unos determinados nodos de AST existe una expresión de retorno.
- Para poder definir un nuevo método y reemplazar uno existente en el *Extract Method*.
- Para poder agregar una nueva variable temporal y reescribir parte de un método en el *Extract Variable*.
- Para poder crear clases y métodos de prueba en los tests de los refactorings.

6.3.7. Parseo y generación de código Smalltalk

Algunos cambios del trabajo, como el agregado de nodos en el AST o nuevos *source ranges* para nodos existentes, requerían conocer el funcionamiento del parser de Cuis: cómo se escanean los caracteres de un cierto input, el reconocimiento de elementos sintácticos y luego semánticos para crear los nodos del AST. Los conocimientos de la materia de “Parseo y Generación de Código” fueron claves para comprender la idea general del parser, como así también los detalles de implementación.

7. Conclusiones y Trabajo Futuro

7.1. Conclusiones

Modelo de refactorings: las abstracciones existentes dentro del modelo de refactorings (Refactoring, Applier, Precondition) fueron perfectamente aplicadas en implementaciones concretas tanto para el *Extract Method* como para el *Extract Variable*. En todas las clases implementadas para los refactorings nuevos, sólo apareció una abstracción (el objeto `ExtractMethodParametersDetector`) que no es ni un refactoring, ni un applier, ni una precondición. Es un objeto auxiliar que colabora tanto al momento de aplicar el refactoring, como al momento de validar que el refactoring sea posible. Luego, al aparecer más de un uso de una precondición, es natural que éstas pasen a ser objetos en lugar de métodos dentro de un refactoring: este fue el caso de `NewTemporaryPrecondition`.

Costo de implementar un refactoring: cada refactoring tiene sus dificultades particulares, como en el caso de *Extract Method* la validación del intervalo de código seleccionado, no tanto la aplicación del refactoring en sí. Hay un aprendizaje por haber realizado el *Extract Method* antes del *Extract Variable*: el segundo tuvo menos momentos de incertidumbre y muchas decisiones de diseño (relacionadas a la validación del intervalo de código) fueron resueltas con facilidad. La parte más incierta sigue estando por fuera de los refactorings, y tiene que ver con la información sobre el código que necesitamos para aplicar refactorings: que existan nodos de AST para cada elemento del código, y que podamos saber para cada nodo de AST, en qué posición del código está (el *source range*). Los momentos en los que no fue posible continuar con el refactoring por una dependencia a un cambio del Parser fueron los más difíciles del trabajo.

Metodología: las reuniones de seguimiento con los directores fueron claves para mantener enfocado el trabajo y descubrir las partes de la solución que más valor aportan. Cuis es un ambiente en el que, una vez que comprendemos cómo funcionan sus partes, es muy fácil mejorarlas, con lo cual hubo varias mejoras que, si bien son siempre bienvenidas, pueden ser una distracción del objetivo de este trabajo. Fueron frecuentes las revisiones de trabajo sin terminar, lo que fue útil para evitar retrabajo y obtener feedback lo más rápido posible. Respecto a la manera de integrar cambios a medida que estaban listos, si bien fue bastante rápida gracias al trabajo de Juan Vuletich y Hernán Wilkinson, tiene sus aspectos a mejorar: para el caso de *changesets*, éstos se deben enviar por correo electrónico a la lista de Cuis, y luego la revisión de código puede darse verbalmente o por correo electrónico. Es deseable poder llevar este flujo de trabajo a un sistema de revisión como el que posee Github con los *pull requests*.

Uso de TDD: en las etapas iniciales costó escribir los primeros tests de manera completa y los ciclos de feedback fueron extensos (del orden de horas por cada test), aunque luego la velocidad de desarrollo aumentó rápidamente sin perder confianza en la calidad del resultado obtenido. Comenzar testeando las validaciones primero fue útil para liberar ciertos bloqueos debido a la complejidad de un “caso feliz”. Una vez superada esa etapa de

construcción y diseño de la solución, más de la mitad de los tests subsiguientes implicaron cambios de unos pocos minutos. Fueron también muy visibles los momentos donde ameritaba dejar de escribir nuevos tests para enfocarse en mejoras. Los nombres de tests quedaron numerados para reflejar la historia de cómo fueron construidos. En resumen fue una experiencia positiva trabajar con TDD y definitivamente ayudó a poder aprender sobre un dominio parcialmente conocido, como así también a evitar errores de regresión.

Recolección de feedback: si bien gran parte de la funcionalidad de los refactorings estaba constantemente validada por los tests automatizados, aún era necesario utilizarlos en búsqueda de casos no contemplados, y sobre todo de detalles en la experiencia de usuario. Allí fue cuando el feedback de los directores y algunos alumnos fue útil para comprender cuáles eran los casos de uso más frecuentes y los problemas comunes que causaba el uso de esos refactorings. Todo ese feedback era analizado en cada una de las reuniones de seguimiento semanales.

Utilidad de los refactorings: tal como se preveía en el momento de armar la propuesta para este trabajo, los refactorings resultaron útiles incluso en sus versiones iniciales donde no estaba toda la funcionalidad terminada. Las primeras personas en adoptarlo fueron los docentes mostrando su uso en las clases.

Calidad del código general de Cuis: al comenzar este trabajo no se conocía con certeza qué partes de la distribución había que modificar. Lo que respecta al proyecto de refactoring, al ser un código reciente y construido desde cero con Test-Driven Development resultó más fácil de cambiar que otras partes, como por ejemplo el Parser. Aún el Parser actual conserva muchos detalles de implementación heredados de Squeak, con métodos largos o difíciles de comprender y acoplamiento con objetos como el Encoder. Este trabajo apuntó a solucionar deuda técnica en el parser, dejando el código al menos mejor de lo que estaba antes de cada cambio. Situación similar con toda la lógica de manejo de *source ranges*, aunque allí la mejora introducida fue mayor. Los refactorings realizados sobre esa base de código fueron muy desafiantes y debieron realizarse con sumo cuidado para evitar causar errores en usuarios de Cuis.

Contribuciones de código abierto: definitivamente fue gratificante poder construir un producto que tiene una utilidad práctica y que beneficia a todos los usuarios de la distribución de Cuis, y cuyo código está disponible para que cualquier persona pueda mejorarlo o extenderlo. Cabe reiterar que los cambios al parser y los nuevos *source ranges* tienen relevancia y utilidad más allá de hacer posible los refactorings presentados en este trabajo.

7.2. Trabajo Futuro

7.2.1. Mejoras al *Extract Method*

Seleccionar “inteligentemente” de acuerdo a la posición en el código de los diferentes nodos del AST: podría no ser necesario seleccionar explícitamente todo el fragmento de código a extraer, sino que se infiera en base a la posición actual del cursor y el nodo de

AST más cercano a esa posición. Por ejemplo, si el cursor está posicionado dentro de un string, se podría iniciar el refactoring y que eso se interprete como el deseo de extraer el string, y por ende el intervalo de selección se ajuste al inicio y fin del string.

Sugerir fragmentos repetidos de código: en algunas ocasiones, el *Extract Method* se realiza para evitar duplicación de código. Como parte del proceso de aplicar el refactoring, se podría sugerir si hay fragmentos de código similares al que se desea extraer, dentro del método actual o en otros métodos dentro de la clase donde se realiza el refactoring. Estas sugerencias podrían ser aceptadas o rechazadas por quien programa, y concluir el refactoring con el reemplazo en todos los lugares escogidos además del lugar original donde se inició el refactoring.

Extraer a un método existente: otra estrategia para eliminar duplicación de código, alternativa o complementaria a la sugerencia de fragmentos duplicados, es la siguiente: al momento de realizar una extracción, verificar si el código que se desea extraer ya está definido en un método existente dentro de la clase donde se realiza el refactoring. Si esto ocurre, se puede sugerir utilizar dicho mensaje en lugar de definir uno nuevo.

Mantener indentación consistente: la extracción de código y reescritura dentro de un método nuevo por parte del *Extract Method* es completamente textual, es como si se copiara y pegara el código dentro del método nuevo. Esto podría tener una desventaja de estilo, ya que el nivel de indentación donde se realiza el refactoring no es necesariamente el mismo que el nivel de indentación del método de destino. Una posible mejora sería ajustar los niveles de indentación para mantener consistencia.

7.2.2. Mejoras al *Extract Variable*

Mantener indentación consistente: la extracción de código y reescritura dentro de un método nuevo por parte del *Extract Variable* es completamente textual, es como si se copiara y pegara el código dentro de la nueva asignación de variable. Esto podría tener una desventaja de estilo, ya que el nivel de indentación donde se realiza el refactoring no es necesariamente el mismo que el nivel de indentación de la asignación de variable de destino. Una posible mejora sería ajustar los niveles de indentación para mantener consistencia.

Sugerir expresiones duplicadas: Análogo al *Extract Method*, es probable que haya expresiones duplicadas de código que también puedan ser extraídas. En el caso de *Extract Variable*, esta sugerencia puede presentarse en la interfaz gráfica, pero con la salvedad que es un refactoring que podría cambiar el comportamiento del código; por ejemplo, si la expresión que extraemos genera efectos secundarios (como cambiar el valor de una variable de instancia).

Mover declaración entre diferentes *scopes*: Por defecto, el refactoring ubicará la declaración de la variable nueva en el scope más cercano posible a la asignación y posterior uso. Muchas veces es deseable mover esa declaración a bloques de código

superiores, o incluso a la declaración principal del método. Actualmente, hacer este cambio es manual, no hay refactoring automático que lo haga.

7.2.3. Mejoras al proyecto de refactorings

Implementar la posibilidad de deshacer un refactoring: esto no es una tarea trivial, ya que hay casos en los que la posibilidad de deshacer un refactoring se invalida, ya que pudo haber otros refactorings o cambios manuales que hagan imposible regresar el programa al estado original anterior a aplicar el refactoring.

Escribir tests automatizados para los objetos *applier*: estos objetos poseen lógica que tiene que ver con cómo trabajar con los datos que se le solicitan para iniciar un refactoring, como así también con los resultados que se presentan. Dicha lógica se encuentra acoplada a la interfaz gráfica en Morphic, lo que dificulta su test automatizado.

Implementar el refactoring *Inline Method*, que es el refactoring contrario al *Extract Method*; es decir, dado un mensaje, reemplazar los lugares donde se envía ese mensaje por el código del método asociado a ese mensaje.

Implementar el refactoring *Inline Variable*, que es el refactoring contrario al *Extract Variable*. Este refactoring es bastante útil cuando se extrajo duplicación usando *Extract Variable*, y luego *Extract Method*, suele utilizarse *Inline Variable* para revertir el primer efecto del *Extract Variable*.

7.2.4. Mejoras a Cuis Smalltalk

Con el agregado de nuevos nodos de AST para declaración de variables temporales (sección 5.2.4), queda una mejora pendiente en las clases **`MethodNode`** y **`BlockNode`** con respecto a cómo se almacenan y cómo se utilizan las variables temporales. Hoy en día esa información está duplicada, ya que se puede acceder a las variables temporales de dos maneras: como estaba antes del *changeset* #4085 (variable de instancia `temporaries` que referencia a una colección de instancias de **`TempVariableNode`**); y luego del *changeset* #4090 (variable de instancia `temporariesDeclaration`, instancia de **`TemporariesDeclarationNode`**). El impacto de este cambio es alto (implica cambiar lógica de compilación y decompilación), por lo que se decidió no incluirlo como parte de este trabajo.

Nodos de AST para representar aquellas partes de código que aún no están representadas: existen partes del código fuente que no tienen su correspondiente nodo en el AST, lo que dificulta su análisis en herramientas como refactorings. Estos elementos de la sintaxis del lenguaje son los siguientes:

- Cabecera o firma de métodos: el nombre y sus nombres de colaboradores externos si corresponde. Los nombres de colaboradores externos aparecen en los *source ranges*, pero sin distinguir que forman parte de la cabecera del método.
- Comentarios: los comentarios llegan al AST como un atributo `comment` que se le puede añadir a cualquier nodo, pero no están reificados como nodos.

- Pragmas: los pragmas no poseen ningún nodo de AST ni ninguna referencia indirecta desde un nodo existente del AST, con lo cual esta información se pierde en caso de querer analizar un `MethodNode`.

Interfaz de usuario tipo formularios: Los refactorings, cuando necesitan solicitar información del usuario, como por ejemplo nombres de variables o mensajes, lo hacen utilizando un componente básico de Morphic llamado `StringRequestMorph`, que consiste en una entrada de texto, y la opción de aceptar presionando Enter, o cancelar presionando Esc. Esta interfaz puede ser ampliamente mejorada, agregando soporte para múltiples campos de texto por ventana, o validando el texto escrito por el usuario a medida que se va escribiendo (y, en consecuencia, activar o desactivar la opción de Aceptar).

8. Referencias

Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. 2013.

Anderson, D. J. *Kanban: successful evolutionary change in your technology business*. 2010.

Beck, K & Andres, C. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2001.

Beck, K. *Smalltalk: best practice patterns*. Prentice-Hall, Inc., USA, 1996.

Beck, K. *Test Driven Development. By Example*. Addison-Wesley Longman, Amsterdam, 2002.

Beck, K., et al. *The Agile Manifesto*. Agile Alliance, 2001. Disponible en http://agilemanifesto.org/

Fowler, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

Goldberg, A & Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co. Inc, USA, 1983.

Kay, A. C. *The early history of Smalltalk*. In The second ACM SIGPLAN conference on History of programming languages (HOPL-II). Association for Computing Machinery, New York, NY, USA, 69–95. 1993. Disponible en http://worrydream.com/EarlyHistoryOfSmalltalk/

Nierstrasz, O. Ducasse, S. & Pollet, D. *Squeak by Example*. Square Bracket Associates, 2009

Odpyke, W.F. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992. Disponible en: http://www.laputan.org/pub/papers/opdyke-thesis.pdf

Roberts, D., Brant, J. & Johnson, R. *A refactoring tool for smalltalk*. Theory and Practice of Object Systems 3, 253–263, 1997.

Schwaber, K. *SCRUM development process*. Springer London, 1997.

Wilkinson, H. *VM Support for Live Typing: Automatic Type Annotation for Dynamically Typed Languages*. MoreVMs Workshop, 2019.

Wilkinson, H; Prieto, M & Garbezza, N. *CuisUniversity: una herramienta para el aprendizaje de programación orientada a objetos con un enfoque iterativo e incremental*. Póster publicado en las 1ras Jornadas Argentinas de Didáctica de la Programación (JADiPro), 2018.

Wilkinson, H., Prieto, M & Romeo, L. *Arithmetic with Measurements on Dynamically-Typed Object-Oriented Languages*. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA, Congreso), 2005.

Wilkinson, H., Prieto, M & Romeo, L. *A Point Based Model of the Gregorian Calendar*.

European Smalltalk User Group (ESUG, Congreso), 2005.

9. Anexo: contribuciones realizadas a Cuis

- *Changeset* #3657:
  - Descripción: Categorizar métodos que estaban sin categorizar.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/5a3b857b54256936392e31a29061691a6b025857
- *Changeset* #3658:
  - Descripción: Nuevo atajo de teclado para renombrar clases, métodos o categorías.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/58235915bd7acddcd5381b6ae3bbb1d46e0bf901
- *Changeset* #3660:
  - Descripción: Nuevos atajos de teclado para la herramienta *File List*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/d4f4bd1110fcb5e4bf3dab7b3160c66855d9258b
- *Changeset* #3673:
  - Descripción: Posibilidad de navegar en el *browser* utilizando las teclas Izquierda y Derecha para moverse entre categorías, clases y mensajes.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/04349aab443885ffbca198819632b41390d71113
- *Changeset* #3674:
  - Descripción: Agregar implementación faltante de `#isLiteralVariableNode`.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/b6c85872584cac001943640fc5faf1ea2b3e6f6e
- *Changeset* #3692:
  - Descripción: Refactoring en la clase `Browser`.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/60a0578ce2bf6c8a04c38ea1b43df8ce6e415be4
- ChangeSet #3725:
  - Descripción: Refactoring en la clase `KeyboardEvent`.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/0998255e333a1fbbb44b305d17be16dc69ca3b12
- *Changeset* #3753:
  - Descripción: Agregar posibilidad de borrar palabras hacia atrás usando Ctrl/Alt+Backspace.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/5474d381d7a56f975ec77dab8f7ac3dc086c2a5a
- ChangeSet #3759:
  - Descripción: Agregar posibilidad de borrar palabras hacia adelante usando Ctrl/Alt+Delete.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/05678eba379687a0da4edc903237f1e4b58939c0

- *Changeset* #3762:
  - Descripción: Reportar *source ranges* para los nodos de tipo `BraceNode`.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/37f3ee0a993a72f2b4b97fcdb248261e4d2dc375
- *Pull request* #161
  - Descripción: Agregar una sección sobre Browsers a la *Terse Guide* de Cuis.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/pull/161
- *Changeset* #3991:
  - Descripción: Primera versión del refactoring *Extract Method*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/5a3c8df83306a0494b90451bdb231b9e519d7687
- *Changeset* #4005:
  - Descripción: Arreglo de error con sugerencia de parámetros en *Extract Method* y algunas tareas menores de refactorización.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/9b0343c7c5add524271b688b3db7db1b96ce9af9
- *Changeset* #4025:
  - Descripción: Soporte para expresiones con *backticks* en *Extract Method.*
  - Links: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/10606f22cd6c16723ca4da1002ee8798600aac9b y https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/c8778c10a9f649f0a8ee5f162ce01744dd1feef7
- *Changeset* #4030:
  - Descripción: Mejoras al menú contextual del Inspector.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/ad63993b3ea30f92c74d9f85f930c9ff020bfc8c
- *Changeset* #4042:
  - Descripción: Agregar la posibilidad de inspeccionar las claves de un diccionario en un Inspector.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/6ee387a28fb6dd60ae98eec6d1a1580d6226b807
- *Changeset* #4048:
  - Descripción: Agregar comentarios a clases del proyecto de Refactoring; extraer precondiciones de creación de variables temporales a una clase aparte; refactorizar clase de test del *Extract Method*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/dcda7d57bbb5b32470d598700f7e7b65e21f1d88
- *Changeset* #4049:
  - Descripción: Borra implementación anterior (incompleta) de *Extract Variable*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/3dbbea956361566f400500a00922281707961167
- *Changeset* #4059:
  - Descripción: Arregla test errático de *Extract Method*.
  - Link:

https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/304c65445dbeb0cfe7c4bc5996be431d6f55912b

- *Changeset* #4085, #4086, #4087, #4088, #4089, #4090, #4093, #4094 y #4096:
  - Descripción: Agregar nuevos nodos de declaración de variables temporales y protocolo de *visitor* para dichos nodos; corregir error de *Extract Method* al extraer mensajes optimizados por el compilador; corregir validación relacionada a variables temporales en el *Extract Method*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/e7e3959a85c1d54ed3cf4906959696ab5064e514
- *Changeset* #4107:
  - Descripción: Permitir extraer declaraciones de variables temporales como parte de un *Extract Method*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/a44e207ab2971a771e5dbfa4f70077ac962358ce
- *Changeset* #4108:
  - Descripción: Corregir implementación de `#isValidSelector` en la clase Symbol.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/98437aee44c1b24a129b6573839d8cf515c9fefa
- *Changeset* #4115:
  - Descripción: Primera versión del refactoring *Extract Variable*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/d39dd4219e4d84b1a66b0b4cdb31777eb0d13fa8
- *Changeset* #4116 y #4117:
  - Descripción: Soporte para extracción de mensajes en cascada en *Extract Method*; refactoring en la clase `Parser` respecto al parseo de mensajes en cascada. https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/ab27cddfd5e78695390749e42c07b7aa636edf15
- *Changeset* #4128:
  - Descripción: Mejoras a *Extract Variable* y *Extract Method*; corrección de error en *Extract Method*; agregar validación a *Rename Temporary*.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/4c297cf732fcfdb90fc24c2a95ad0abf4a7d465c
- *Changeset* #4156:
  - Descripción: Nueva abstracción `SourceCodeInterval` para representar intervalos de código.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/322e43694e7a56248209478b15572241858e66b3
- *Changeset* #4187:
  - Descripción: Arreglar errores en *Extract Method* y *Extract Variable*; agregar comentario a la clase `RefactoringPrecondition`.
  - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/af9be98929da657c9b587e6c9152b4024fbd64e7
- *Changeset* #4197:
  - Descripción: Corregir mensaje de error confuso de *Extract Variable*. Mejora

de performance en `SourceCodeInterval`.
    - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/e5e41841da1bdb4d3518ea5176463703b38247bb
- *Changeset* #4230:
    - Descripción: Refactorizar usos de la clase `MethodReference`.
    - Link: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/commit/f2e195361e559a04dfaa891f17abbd3ea702a53c