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

Implementación de refactorings move y extract usando LiveTyping

Tesis de Licenciatura en Ciencias de la Computación

Maximiliano Joel Cabezon Alvarez

Director: Hernan Wilkinson

Buenos Aires, 2024

IMPLEMENTACIÓN DE REFACTORINGS MOVE Y EXTRACT USANDO LIVETYPING

El presente trabajo dispone a mostrar la implementación de tres refatorings automatizados: **Move instance variable**, **Move method** y **Extract class** para los entornos de programación **Cuis-Smalltalk** y **Cuis-University**. Este último entorno dispone de una herramienta llamada **LiveTyping** que se utilizará para introducir mejoras a los refatorings mencionados. Finalmente, se concluirá que el objetivo de los refactorings no es alcanzar la perfección, sino ser confiables.

Palabras claves: Refactoring, Automático, Cuis, Smalltalk, Move, Instance, Variable, Method, Extract, Class, Livetyping.

AGRADECIMIENTOS

A mi familia, mis hijas Sofi y Delfi, mi compañera de vida Ceci. Mis padres, Miguel y Patricia. A mis abuelos, algunos presentes físicamente. A mis tíos y primos. La familia es lo mas importante en la vida de toda persona y es lo que siempre permanece en todo camino recorrido.

A mis amigos y compañeros. Se dice que los amigos son la familia que uno elige y que siempre permanece al lado de uno.

A la institución. Departamento de computación (DC), Facultad de Ciencias Exactas y Naturales (FCEN), Universidad de Buenos Aires (UBA). Tanto al cuerpo docente como al no-docente por llevar adelante esta institución y que me ha influido de manera positiva como profesional y como ser humano.

A Hernan y a toda la comunidad Smalltalk. Creo que sus conocimientos y consejos han sido de gran ayuda para la carrera de cada alumno y, por ende, para la industria en general, incluyendo aquellos consejos que van más allá de la ingeniería en software.

Índice general

1.. Introducción 1

2.. ¿Por qué Cuis-Smalltalk? 2
- 2.1. El Browser 4
- 2.2. Live Typing 5
- 2.3. TDD 6

3.. Refactorings 7

4.. Move instance variable 9
- 4.1. Motivación 9
- 4.2. Definición 9
- 4.3. Precondiciones 10
- 4.4. Diagrama 10
- 4.5. Implementación 11
- 4.6. ¿Como se usa? 12
- 4.7. Discusión 15
- 4.8. Trabajo futuro 16

5.. Move method 17
- 5.1. Motivación 17
- 5.2. Definición 17
- 5.3. Precondiciones 18
- 5.4. Diagrama 19
- 5.5. Implementación 20
- 5.6. ¿Como se usa? 21
- 5.7. Discusión 22
- 5.8. Trabajo futuro 24

6.. Extract class 25
- 6.1. Motivación 25
- 6.2. Definición 25
- 6.3. Precondiciones 26
- 6.4. Diagrama 26
- 6.5. Implementación 28
- 6.6. ¿Como se usa? 28
- 6.7. Discusión 30
- 6.8. Trabajo futuro 31

7.. Conclusiones 32

1. INTRODUCCIÓN

En este trabajo veremos la implementación de tres refactorings automáticos para el ambiente de programación **Cuis-Smalltalk**. Estos refactoring son llamados: **Move instance variable**, **Move method** y **Extract class**.

Primero se mencionarán las razones por la que se ha elegido **Cuis-Smalltalk**, por qué lo hicimos con **TDD** y qué beneficio nos provee **LiveTyping**.

Después se dedicará una sección a la definición general de refactoring, cómo son los dos grupos de refactorings, cuáles son las deferencias de implementarlo en un lenguaje estático o dinámico y como nos ayuda **LiveTyping**.

Luego se dedicará un capítulo para cada uno de los tres refactorings mencionados. Cada uno de estos capítulos dispondrán de ocho secciones que nos ayudarán a mostrar todo el trabajo realizado. A continuación se explica cada una de las secciones:

1. **Motivación**: Aquí vamos a mencionar cuáles son las razones por la que debería existir este refactoring y cuáles son los contextos en los que podría ser usado.

2. **Definición**: Aquí explicaremos en qué consiste el refactoring y cuál debería ser el resultado final luego de ejecutarlo.

3. **Precondiciones**: No siempre se pueden ejecutar los refactorings. En esta sección vamos a mencionar los requisitos para su ejecución.

4. **Diagrama**: Los diagramas son muy útiles para comunicarse, es por eso que hemos agregado esta sección. Se mostrará un diagrama antes de ejecutar el refactoring y un diagrama después de ejecutarlo.

5. **Implementación**: Aquí se mostrarán cuales son los pasos que realizan los refactorings implementados.

6. **¿Cómo se usa?**: Las interfaces gráficas no siempre son intuitivas. Aquí mostraremos cómo acceder y cómo ejecutar, paso a paso, el refactoring en **Cuis-Smalltalk**.

7. **Discusión**: Hay decisiones que hemos tomado para poder implementar el refactoring. Aquí mencionaremos las razones por las cuales las tomamos.

8. **Trabajo futuro**: Todo refactoring puede continuar mejorándose. Se mencionarán qué mejoras se pueden agregar o cuáles son los posibles caminos a seguir.

Finalizaremos este trabajo con una conclusión en donde expondremos algunas ideas en base al trabajo realizado.

2. ¿POR QUÉ CUIS-SMALLTALK?

El presente trabajo se ha realizado principalmente en **Cuis-University**[18], pero debemos mencionar que todos los refactorings de los que vamos a hablar van a poder ser ejecutados tanto en **Cuis-University** como en **Cuis-Smalltalk**[15].

Cuis-Smalltalk es un ambiente oriendato a objetos derivado de la plataforma Squeak. **Cuis-Smalltalk** se destaca por ser un ambiente minimalista y fácil de entender. Esto puede verse en la cantidad de código que posee, que es, mucho menor respecto de otras distribuciones de Smalltalk. También decimos que es fácil de entender debido a que podemos inspeccionar cualquier parte de la imagen de la máquina virtual y realizar experimentos con ella.

Cuis-University es un ambiente derivado de **Cuis-Smalltalk**. Este ambiente tiene como objetivo facilitar la enseñanza en la programación orientada a objetos. Para lograr este objetivo **Cuis-University** viene con los siguientes paquetes preinstalados:

- **Denotative Objects**[16]: una herramienta para trabajar con objetos concretos en lugar de clases e instancias. Muy parecido a lo que es un lenguaje de prototipación.
- **LiveTyping**[17]: anotación automática de tipos para lenguajes dinámicamente tipados. Mas adelante detallaremos como funciona esta herramienta.
- **Aconcagua**[19]: Herramienta de medidas y unidades.
- **Chaltén**[20]: Implementación de un modelo de un calendario gregoriano enriquecido.
- **TDD-Gurú**[4]: Herramienta que analiza cambios en la imagen de **Cuis-Smalltalk** para detectar si se realizó o no TDD.

Dado este contexto, ¿Por qué elegimos Cuis?. Primero debemos mencionar que los refactorings elegidos no existen en las versiones de Cuis mencionadas. También, hay una gran ventaja en realizar los refactorings en Smalltalk. La capacidad de trabajar bajo un ambiente reflexivo nos permite realizar experimentaciones de una manera más rápida que el común de los entornos de programación. Esto es por que generalmente en los entornos de desarrollo se tiene una máquina virtual para cada ambiente. Entonces, cada vez que generamos una versión nueva de nuestro software debemos instalarla en cada uno de estos ambientes. Si miramos el ejemplo de la fig. 2.1, vemos que cada vez que queremos correr los test debemos instalar nuestro software en el ambiente donde corren los test, cada vez que queremos realizar un debug debemos instalar nuestro software en el ambiente donde existen nuestras herramientas de debug y cada vez que queremos usar nuestro software desde el IDE debemos instalarlo en la VM que corre el IDE.

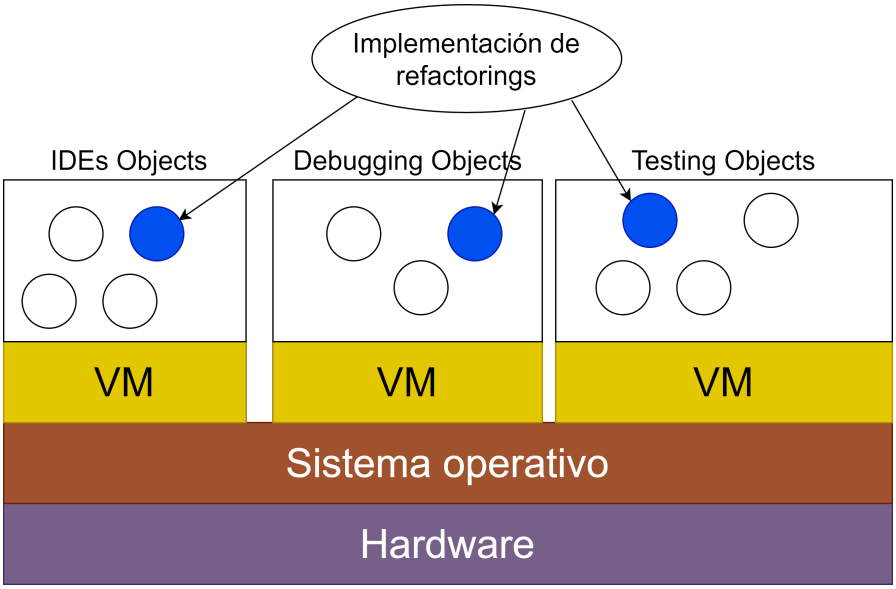


Fig. 2.1: Ejemplo de entorno de desarrollo. Aquí hay una VM para ejecutar el IDE, una VM para realizar debug y una VM para correr los test.

En Smalltalk tenemos la ventaja de tener una única implementación de nuestro software y esta puede ser usada por cualquier herramienta que se encuentra en nuestro ambiente. En la fig. 2.2 podemos ver como es en general un ambiente de programación Smalltalk, en donde nuestro software puede ser usado por el IDE, el debugger o los objetos de testing. Gracias a esto podemos realizar distintos experimentos sin la necesidad de estar reinstalando nuestro software en distintas máquinas virtuales, también se pueden combinar estas herramientas. Por ejemplo, podemos realizar un debug de nuestra implementación llamandola desde el IDE.

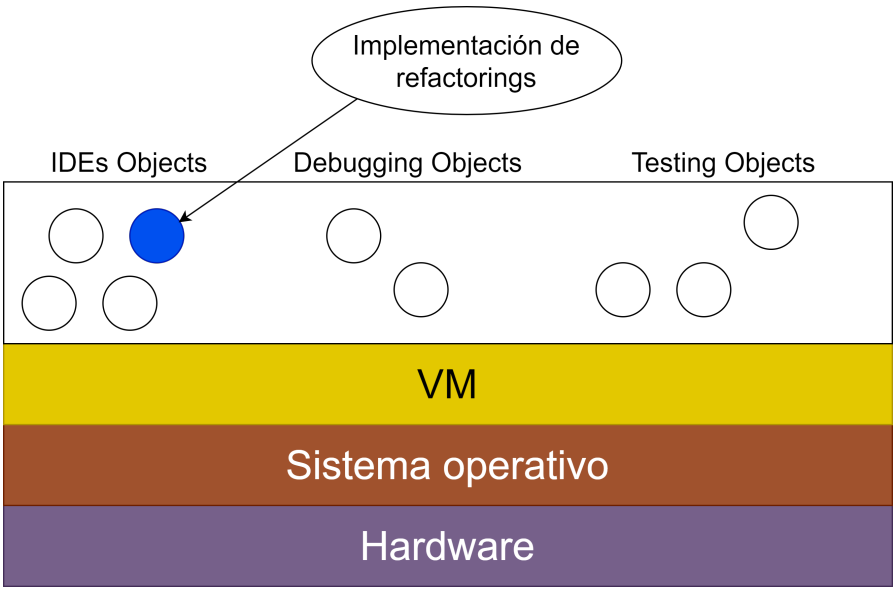


Fig. 2.2: Ejemplo de entorno de desarrollo Smalltalk. Aquí hay una única VM para ejecutar el IDE, realizar debug y correr los test.

2.1. El Browser

Una de las herramientas mas usadas en **Cuis-Smalltalk** es el Browser[21]. Gracias al Browser podemos navegar cualquier categoría o clase que se encuentre en el entorno de **Cuis-Smalltalk**.

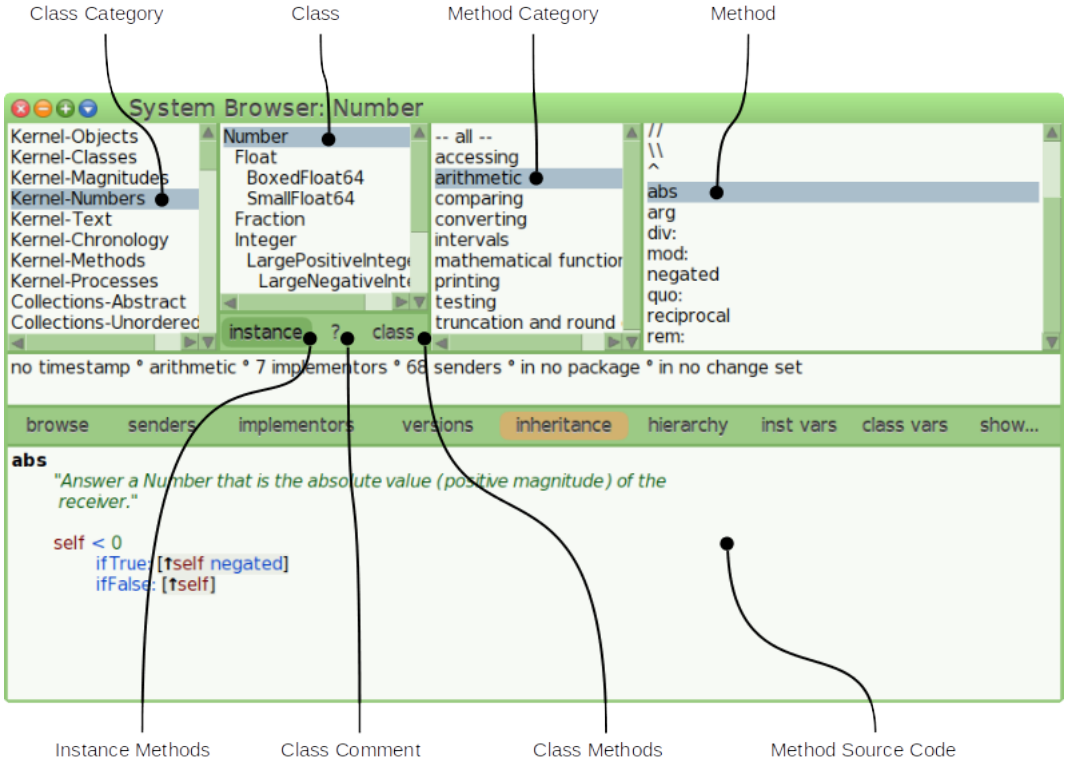


Fig. 2.3: Componentes del Browser

A continuación describiremos los elementos señalados en fig. 2.3 para tener una mejor comprensión de la forma en la que podemos usar esta herramienta:

- **Class Category:** Aquí se encuentran las categorías de clases. Estas categorías están para poder agrupar un conjunto de clases.
- **Class:** A la derecha del **Class Category** se encuentra la lista de clases perteneciente a la categoría seleccionada en **Class Category**.
- **Method Category:** Aquí se encuentran las categorías de métodos. Estas categorías están para agrupar un conjunto de métodos de la clase seleccionada.
- **Method:** En este listado encontramos los selectores de cada método. Al seleccionar uno se nos mostrará el código fuente del método correspondiente a ese selector en el componente **Method Source Code**.
- **Instance Methods:** Al seleccionar esta opción sólo se mostrará los métodos de instancia de la clase que hemos seleccionado.
- **Class Comments:** Con este botón podemos ver los comentarios que se han dejado para la clase que hemos seleccionado. También pueden dejarse códigos de ejemplo de cómo usar la clase seleccionada.

- **Class Methods:** Al seleccionar esta opción sólo se mostrará los métodos de clase de la clase que hemos seleccionado.
- **Method Source Code:** En este elemento podemos ver el código fuente correspondiente a la clase y selector que hemos seleccionado anteriormente.

Se ha decidido integrar los refactorings de este trabajo al browser debido a que es la herramienta principal de programación de todos los ambientes de programación de Smalltalk. Don Roberts propone en su artículo[13], que los refactorings deben estar integrados a las herramientas de desarrollo para que sean usados fácilmente por los desarrolladores. Podemos ver que estos tres elementos del browser están muy relacionados con nuestros refactorings:

- Uno de los elementos del browser es la lista de clases. Los refactorings **Move instance variable** y **Extract class** hacen uso de este elemento debido a que tienen que realizar acciones a partir de una clase dada. Describiremos como es la interacción con esta parte del browser en los capítulos de cada uno de estos refactorings.
- El segundo de los elementos del browser es la lista de métodos. Todo programador sabe que en Smalltalk los métodos de una clase pueden ser visualizados desde aquí. Entonces este elemento puede ser muy útil para el refactoring **Move method**.
- El último elemento del browser útil para nuestros refactorings es la sección de código. En esta sección podemos ver el selector de un método seguido del código del mismo. Este elemento es útil para los refactorings **Move instance variable** y **Move method**.

2.2. Live Typing

En los lenguajes dinámicamente tipados, como Smalltalk, es muy difícil determinar el tipo de una variable o el tipo de retorno de un método debido al late binding. Por otro lado, en los lenguajes estáticamente tipados se exige que el programador escriba de manera explicita el tipo de las variables o el tipo de retorno de la función o bien puede inferirse en algunos casos. Si bien esto último es útil para herramientas como el autocompletar código, type checker o los refactorings, esto hace que la tarea de escribir código sea más complicada. Entonces, ¿Se podría tener una herramienta que proporcione información de tipos en un lenguaje dinámicamente tipado en tiempo de ejecución?. **LiveTyping** es una herramienta que nos brinda mucha ayuda con respecto a esto.

LiveTyping es una herramienta que recolecta información de tipos observando las asignaciones de variables, resultado de retorno de métodos y colaboraciones. Es decir, mientras que el sistema está corriendo, se guardan todos los tipos que se han asignado a una variable y todos los tipos que un método ha devuelto en una estructura aparte para poder hacer uso de ellos. Si bien la idea de inferir tipos no es algo nuevo en Smalltalk[14], debemos destacar que esta herramienta es totalmente automática en lo que respecta a la recolección de tipos.

En las siguientes secciones se mostrará cómo es que **LiveTyping** puede usarse para mejorar la experiencia de uso en los refactorings. Para el **Move instance variable** y **Move method** se ha utilizado **LiveTyping** para hacer recomendaciones a la hora de seleccionar la clase de destino.

2.3. TDD

Test Driven Development (conocido como **TDD**) es una técnica publicada por Kent Beck en el 2002[1] y se basa en el desarrollo iterativo e incremental guiado por tests. **TDD** tiene 3 simples pasos:

1. Escribir un test que falle.
2. Escribir el código necesario para que el test pase y que pasen los test previamente escritos.
3. Refactorizar el código si es necesario.

Si pensamos en los refactorings de este trabajo podemos ver que hay bastantes casos a considerar. Por ejemplo, a la hora de realizar un **Move method**, puede ser que el código del método esté referenciando variables de instancia, variables de clase o incluso la pseudo-variable *super*. Es por esto que **TDD** es una herramienta excelente para el desarrollo de manera iterativa e incremental de los refactorings, en donde cada test es un nuevo caso a tener en cuenta por el refactoring.

3. REFACTORINGS

Hay veces que los programas son muy difíciles de modificar. Existen muchas razones por las cuales estos programas han sido diseñados de una manera en la que las personas tienen muchas dificultades para comprender y modificar las distintas secciones del código. Pero a su vez, también existen múltiples herramientas para cambiar el diseño del mismo y poder obtener programas modelados para una mejor comprensión y una mejor adaptación a un posible cambio.

Una de estas herramientas son los refactorings. Según Opdyke, definimos los refactorings como la alteración del código sin alterar el comportamiento del mismo[12]. En nuestra industria es muy común creer que el diseño del software debe realizarse antes de la construcción del mismo. Sin embargo, gracias a los refactorings, podemos cambiar el diseño de manera segura para lograr un diseño que alcance nuestras expectativas.

Si además automatizamos los pasos de los refactorings, podemos estar tranquilos de que no hemos introducido errores humanos en los nuevos diseños que vamos obteniendo. Es por eso que tener refactorings automatizados nos da una tranquilidad muy alta de que no estamos introduciendo bugs.

Actualmente existen mas de 100 tipos de refactorings y cada día se van definiendo más[6]. Pero todos estos refactorings se pueden descomponer en dos grupos[11]:

1. Refactorings de bajo nivel: Estos refactorings son una operacion basica, por ejemplo: **Create class**, **Add method**, **Add field**, etc.

2. Refactorings de alto nivel: Estos refactorings son aquellos que utilizan al menos 2 refactoring de bajo nivel, por ejemplo: **Move instance variable**, **Move method**, **Extract class**, etc.

Uno estaría tentado a decir que cuanto más alto nivel tenga el refactoring mejores diseños se lograrán, debido a que hay menos intervención humana. Sin embargo, los refactorings de más alto nivel son los menos utilizados [10].

Otra cosa que debemos mencionar de los refactorings es que no se implementan igual en todos los lenguajes. Es muy distinto implementar un refactoring en un lenguaje dinámicamente tipado respecto de implementarlo en un lenguaje estáticamente tipado. En los lenguajes estáticamente tipados uno dispone de la información de tipos en todas las colaboraciones y variables del programa. Esto puede ser muy útil a la hora de implementar un refactoring, ya que esta información puede utilizarse a la hora de tomar decisiones para automatizar pasos del refactoring. Por ejemplo: en el refactoring **Rename method**, uno de los pasos finales es encontrar todas las referencias y cambiarles el nombre, es muy fácil saber cuales son todas las referencias ya que los tipos tienen que coincidir. Por otra parte, en los lenguajes dinámicamente tipados no se dispone de esta información de tipos, por lo tanto es más complicado automatizar los pasos de los refactoring por la falta de la misma, entonces, para suplantar esta falta de información necesitamos que el usuario la provea. En el ejemplo que acabamos de mencionar con el **Rename method**, a la hora

de encontrar las referencias del método a renombrar no podemos determinar cuales son las que deberíamos renombrar de las que no deberíamos renombrar. Sin embargo, en el ambiente de desarrollo de **Cuis-Smalltalk** podemos disponer de **LiveTyping** que nos puede dar la información de tipos de manera parcial y así nuestras implementaciones de refactorings se ven beneficiadas por estas.

Los refactorings implementados en este trabajo no son nuevos. Hay varios IDEs que también implementan estos refactorings en lenguajes estáticamente tipados[7][5]. En las siguientes secciones se mencionarán algunas ventajas y desventajas de cada implementación.

A pesar de que los refactorings deben modificar el código sin alterar su comportamiento, es muy tentador pensar que estas modificaciones son siempre seguras. Los refactorings son un programa como cualquier otro, la única diferencia es que su input es el programa a modificar. Pero como cualquier programa, este puede contener bugs[9]. Es por eso que si bien los refactorings nos dan cierta seguridad de que no se va a alterar el comportamiento, es recomendable tener un buen conjunto de tests para que siempre se valide el comportamiento esperado. Aquí volvemos a mencionar que el utilizar una técnica como **TDD** hace que esto sea mas sencillo.

Por último hay que decir que para que un refactoring pueda seguir manteniendo el comportamiento, hay veces que es necesario romper el encapsulamiento. En las implementaciones que vamos a ver, esto se realiza mediante getters y setters. En la comunidad de Smalltalk un getter es un método que devuelve el valor de una variable de instancia y como selector utiliza el mismo nombre de la variable, por ejemplo para una variable de instancia llamada *firstName*, un getter sería de la siguiente manera:

```
firstName
    ^firstName.
```

Por otra parte, para la comunidad de Smalltalk, un setter es un método que lo único que hace es escribir un valor en una variable de instancia, con un selector que lleva el mismo nombre y un parámetro que es el valor a setear. En el caso de la variable de instancia *firstName* el setter sería de la siguiente manera:

```
firstName: anObject
    ^firstName := anObject.
```

4. MOVE INSTANCE VARIABLE

a guardar, a guardar
cada cosa en su lugar
–Canción de niños

En esta sección vamos a presentar nuestra implementación del refactoring **Move instance variable** para **Cuis-Smalltalk**.

4.1. Motivación

En el proceso de cambiar constantemente nuestro modelo, hay veces que agregamos una variable de instancia en un lugar donde no debería estar. Mucho después nos damos cuenta que esa variable debería estar en otro lugar.

Mover una variable de instancia puede llegar a ser un trabajo muy difícil de hacer si la variable es referenciada en muchas partes del código. Es por eso que automatizar este refactoring es una buena idea ya que nos ahorramos el trabajo tedioso de cambiar todos los lugares en donde se referencia y además podemos evitar errores.

El refactoring **Move instance variable** es usado por el refactoring **Extract class** al momento de extraer una variable de instancia a una clase nueva. Esta es otra razón por la cual lo hemos implementado. De este refactoring vamos a hablar en detalle en las próximas secciones.

4.2. Definición

Este refactoring consiste en mover una variable de instancia de una clase a otra. Debemos recordar, que la variable de instancia a mover puede estar referenciada en cualquier parte de la jerarquía de clases. Además la variable de instancia a mover puede estar referenciada de 2 maneras:

- La variable de instancia a mover es leída.
- La variable de instancia a mover es escrita.

Si la variable de instancia es movida, todas estas referencias deben ser cambiadas para poder mantener el comportamiento. En esta implementación hemos decidido cambiar estas referencias con getters y setters.

4.3. Precondiciones

Procederemos a listar las precondiciones necesarias para poder realizar el refactoring **Move instance variable**:

- La variable de instancia debe existir en la clase de la cual la vamos a quitar y no debe existir en toda la jerarquía de la clase a la que se va a mover.
- La variable de instancia que referencia la clase de destino debe existir en la clase de origen.
- La clase de destino no puede estar en la misma jerarquía que la clase de origen.
- Los selectores que se van a usar como setter y getter, no deben estar definidos en la clase de destino ni en sus superclases.

Si al menos una de estas precondiciones no es satisfecha, entonces se lanzará una excepción.

Todas las precondiciones fueron desarrolladas con **TDD** y los test pueden ser encontrados en la categoría de método **validations** de la clase **MoveInstanceVariableTest**.

4.4. Diagrama

A continuación se muestra un diagrama de clases antes de realizar un **Move instance variable** de la variable de instancia **country** y un diagrama de clases después de realizar un **Move instance variable** de la variable de instancia **country** (este ejemplo lo vamos a seguir usando en las secciones posteriores).

Se dispone de una clase llamada **BankCustomer** la cual tiene las siguientes variables de instancia antes de realizar el **Move instance variable**:

- La variable **firstName** que referencia el nombre de nuestro cliente.
- La variable **lastName** que referencia el apellido de nuestro cliente.
- La variable **phone** que referencia el numero de teléfono de nuestro cliente.
- La variable **email** que referencia el mail de nuestro cliente.
- La variable **livesAt** que referencia el lugar de residencia de nuestro cliente.
- La variable **country** que referencia el país de residencia de nuestro cliente.

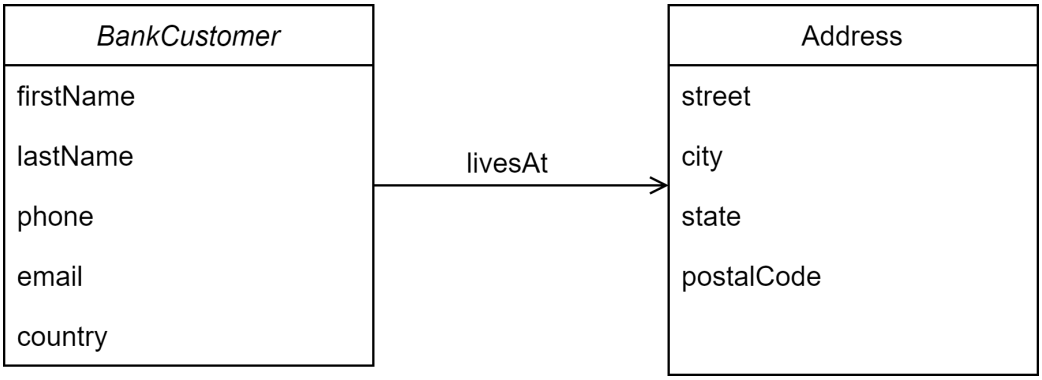


Fig. 4.1: Antes de mover **country**

Luego de realizar el **Move instance variable**, la clase **BankCustomer** dispone de las siguientes variables de instancia:

- La variable **firstName** que referencia el nombre de nuestro cliente.
- La variable **lastName** que referencia el apellido de nuestro cliente.
- La variable **phone** que referencia el numero de telefono de nuestro cliente.
- La variable **email** que referencia el mail de nuestro cliente.
- La variable **livesAt** que referencia el lugar de residencia de nuestro cliente.
- Por último la variable **country**, que referencia el país de residencia de nuestro cliente, ahora se encuentra en la clase **Address**.

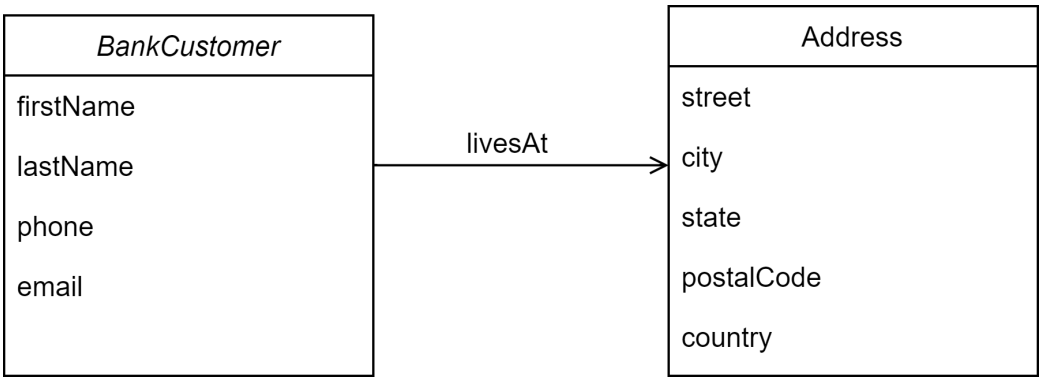


Fig. 4.2: Después de mover **country**

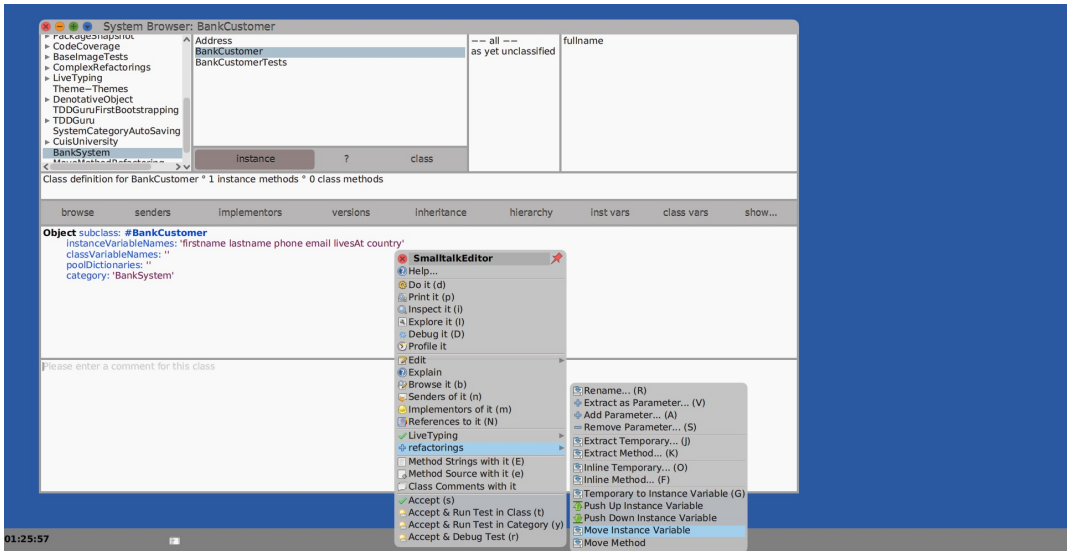
4.5. Implementación

Detallaremos la secuencia de pasos que realiza el **Move instance variable**. Estos pasos son, a grandes rasgos, los que realiza el programa para mover una variable de instancia desde una clase a otra:

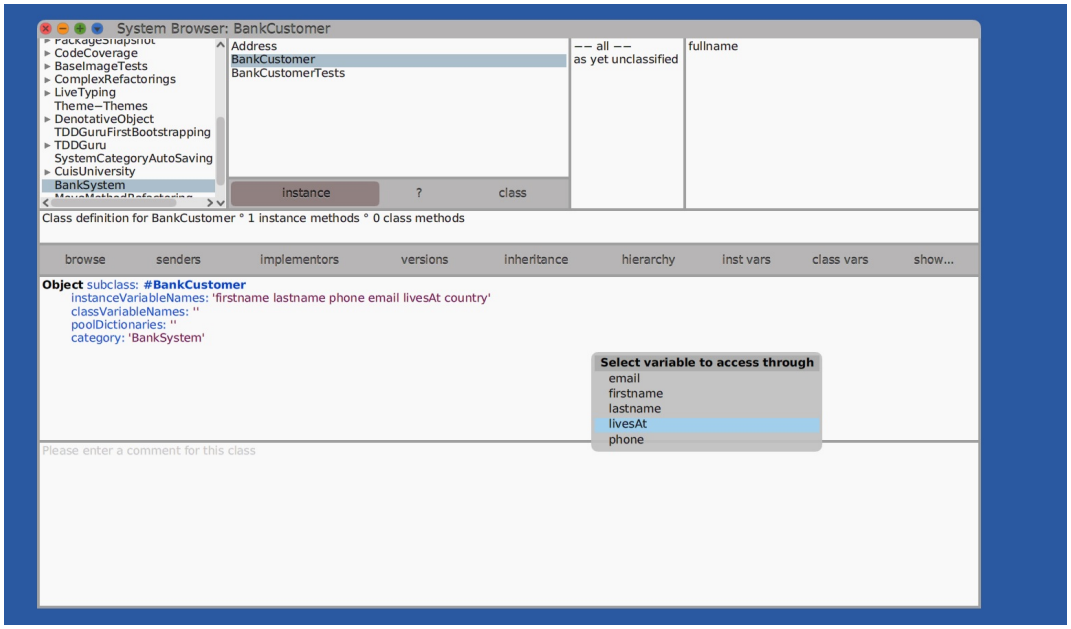
1. Se instancia el refactoring con los siguientes objetos. Debemos recordar que todas las precondiciones mencionadas anteriormente se deben cumplir. Si no llegan a cumplirse, se lanzará una excepción al momento de hacer esta instanciación:
   - La variable de instancia que se desea mover.
   - La clase a la que esa variable de instancia pertenece.
   - La clase de destino a la que se desea mover la variable de instancia.
   - La variable de instancia que referencia la clase de destino.
2. Se buscan todos los accesos a la variable de instancia que se desea mover. Es decir, todos los accesos que hay de la variable a mover en la jerarquía en donde vivía originalmente.
3. Se reemplazan cada acceso de lectura y escritura por un getter y setter respectivamente.
4. Se agrega la variable de instancia a la nueva clase que se ha elegido.
5. Se elimina la variable de instancia de la clase en donde vivía previamente.
6. Si se ha reemplazado al menos un acceso de lectura, se crea un getter en la clase de destino.
7. Si se ha reemplazado al menos un acceso de escritura se crea un setter en la clase de destino.

4.6. ¿Como se usa?

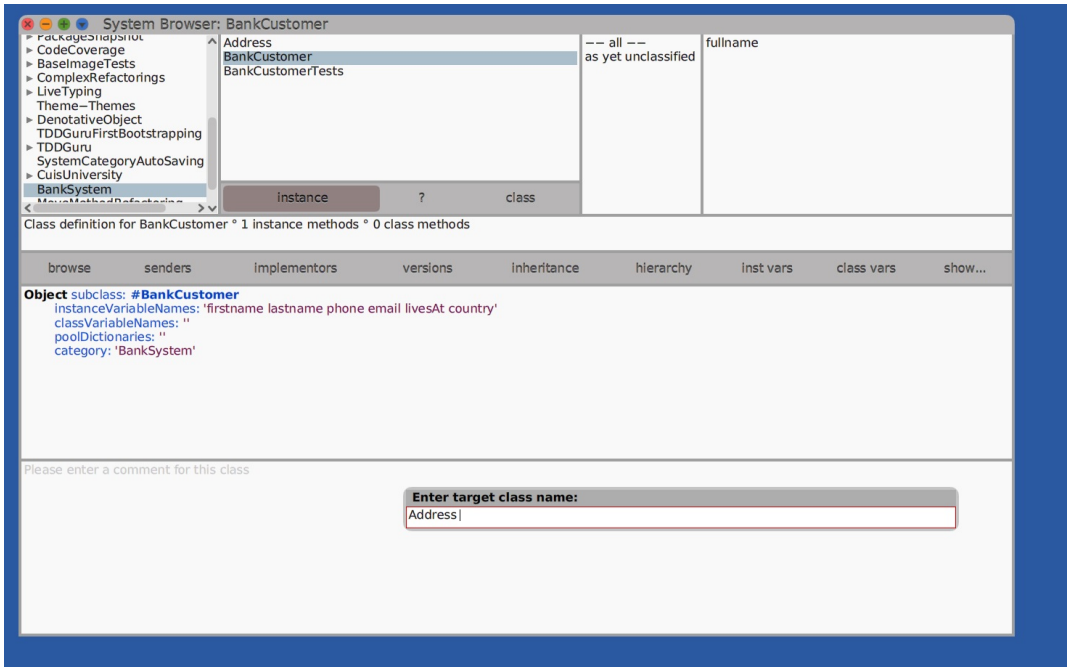
- Se puede acceder al refactoring **Move instance variable** de 3 maneras:
  1. Haciendo click derecho en la variable de instancia que se encuentra en la definición de la clase.
  2. Haciendo click derecho en la variable de instancia que se encuentra en el código fuente de un método.
  3. Haciendo click derecho en el nombre de la clase que se encuentra en el browser y luego seleccionando la variable de instancia que se desea mover.



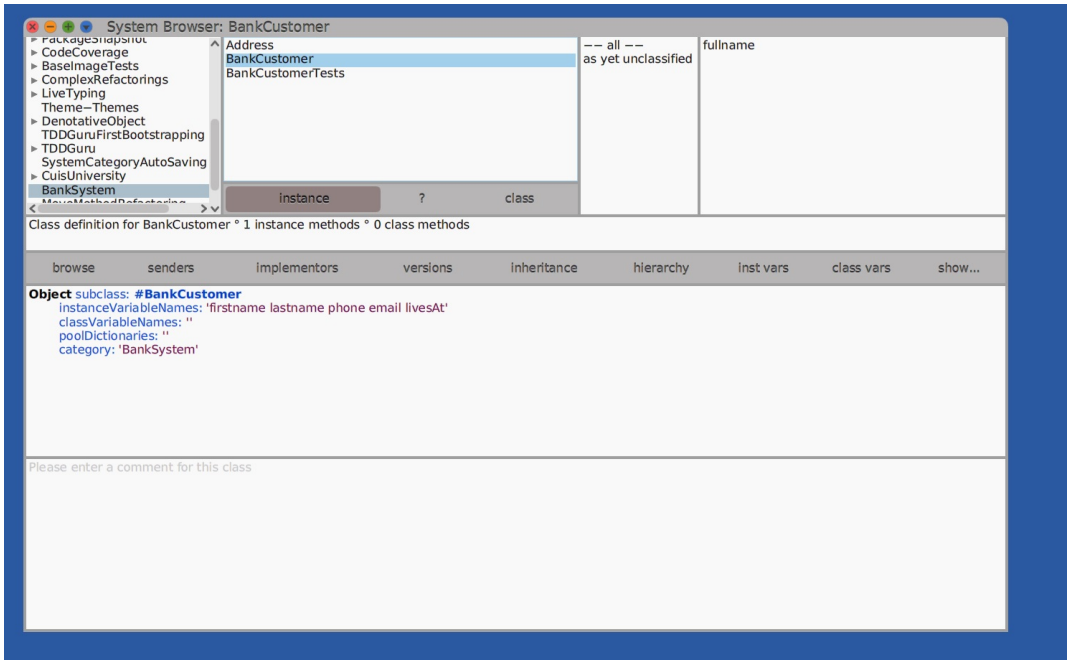
- Ya hemos seleccionado la variable de instancia que deseamos mover. Ahora se debe seleccionar la variable de instancia que referencia la nueva clase a donde se va a mover la variable.



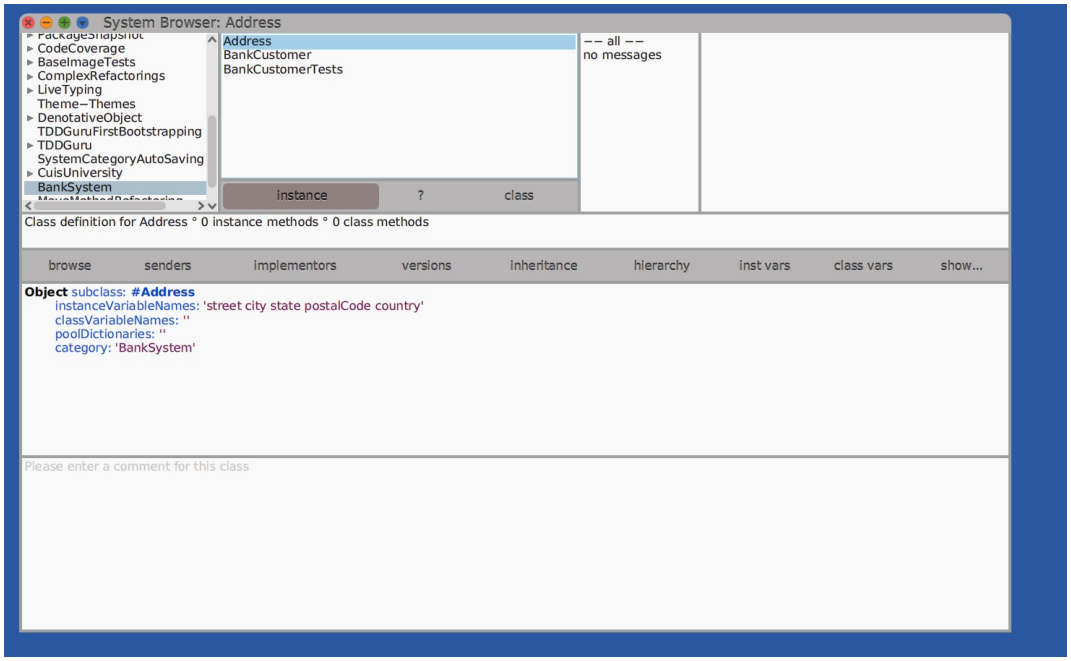
- Después se debe seleccionar la clase de destino a donde se va a mover la variable de instancia. Si disponemos de **LiveTyping** y la variable que seleccionamos en el paso previo dispone de información de tipos, la UI nos va a recomendar un nombre de clase. Este campo se puede auto-completar.



- Finalmente la variable de instancia no pertenece mas a la clase donde pertenecía originalmente.



- Aquí puede verse que la variable de instancia vive en la clase que le hemos indicado.



4.7. Discusión

Al momento de crear un getter o setter en la clase de destino puede ser que el mismo ya exista. Para solucionar este problema, lo que se hace es validar si existe el selector. Si existe, no se permite hacer el refactoring. Si se desea ejecutar el **Move instance variable** aunque exista este selector, se debe usar un segundo constructor que recibe un selector para usar como getter y un selector para usar como setter. Este segundo constructor puede encontrarse en la categoría de métodos **instance creation** de la clase **MoveInstanceVariableRefactoring**. También este constructor es usado por el refactoring **Extract class**.

Otra manera de haber solucionado este problema de getters y setters que existen en la clase de destino, es haber usado metaprogramación con el mensaje **instVarNamed:**. Este mensaje se encuentra implementado en al clase **Object**, por lo tanto, el mensaje **instVarNamed:** lo sabe responder cualquier objeto. Lo que hace este mensaje es devolver el valor de una variable de instancia pasándole como argumento el nombre de la variable. Si bien parece ser muy práctico, debemos recordar que no es una buena práctica usar metaprogramación, ya que si buscáramos las referencias a la variable y una de las lecturas es mediante **instVarNamed:**, esta última referencia no la podríamos encontrar.

LiveTyping se ha usado para determinar cual es el tipo de la variable que referencia la clase a donde se va a mover la variable. Si no se dispone de **LiveTyping** o no hay información de tipos en la variable que referencia la clase de destino, no se hará ninguna recomendación.

A continuación vamos a mostrar un ejemplo en el que esta implementación no funciona debido a que el análisis que se debería hacer para que no falle el refactoring es muy costoso.

Supongamos que tenemos un objeto que representa un paciente de hospital. Este objeto tiene dos variables de instancia, una que referencia el tipo de sangre del paciente (llamada *bloodType*) y otra que referencia datos personales del paciente (llamada *patientData*). Además, tenemos como inicialización del objeto el siguiente código:

```
initialize
    bloodType := 'A positive'.
    patientData := PatientData new.
```

Entonces decidimos que hay que mover la variable de instancia *bloodType* al objeto que referencia la variable *patientData*. Luego de ejecutar el **Move instance variable** el código del método #initialize nos queda de la siguiente manera:

```
initialize
    patientData bloodType: 'A positive'.
    patientData := PatientData new.
```

El problema que puede existir aquí es que la variable *patientData* puede no estar inicializada, lo que provoca que el refactoring introduzca un error. Si bien parece que un simple análisis semántico puede validar esto, estas colaboraciones pueden estar en cualquier parte del código, lo que haría que este análisis lleve mucho tiempo si tenemos mucho código.

Si bien no es un refactoring muy usado, debemos destacar que hay varios IDEs que no tienen este refactoring como el Eclipse IDE[5]. Otros IDEs, como el Intellij[7], poseen este refactoring pero solo esta disponible para variables static.

4.8. Trabajo futuro

Actualmente el refactoring **Self Encapsulate Field** no existe en **Cuis-Smalltalk**. Este último refactoring mencionado consiste en reemplazar todas las referencias de escritura a una variable de instancia por un setter y todas las referencias de lectura a esa misma variable por un getter. Puede ser muy útil aplicar este refactoring antes de ejecutar el **Move instance variable**, ya que cambiaría la cantidad de referencias de la variable que deseamos mover a solo dos referencias.

En la sección anterior vimos que se podría robustecer la implementación con un análisis semántico (ejemplo al mover la variable *bloodType*). Dependiendo de la cantidad de código a analizar esto puede llevar mucho tiempo. Dejamos como trabajo futuro la implementación de distintas estrategias para que el refactoring tenga esta validación (o sea validar que la variable que referencia el destino esté correctamente inicializada) y pueda ser ejecutado en un tiempo razonable.

En la UI se puede ver que no se hace uso del segundo constructor. Se podría pedir los parámetros extra que necesita este segundo constructor, pero por cuestiones de simplicidad hemos dejado que la UI haga uso únicamente del primer constructor.

5. MOVE METHOD

One of the most fundamental, if not the fundamental, decision in object design is deciding where to put responsibilities
–Martin Fowler

En esta sección vamos a ver nuestra implementación del refactoring **Move method** para **Cuis-Smalltalk**.

5.1. Motivación

Es muy común escribir un método en una clase para que funcione un comportamiento que deseamos desarrollar. Luego nos damos cuenta que ese método no debería estar allí.

Cuando los objetos comienzan a tener demasiadas responsabilidades o cuando el método de un objeto referencia otro objeto demasiadas veces puede ser un síntoma de que el comportamiento que realiza no debería encontrarse en esa clase.

Dependiendo de la cantidad de dependencias del método, moverlo de una clase a otra puede ser una tarea muy difícil. Las dependencias podrían ser cualquier elemento de la clase en donde está definido el método antes de ser movido: una variable de instancia, un método de la misma clase, una variable de clase, referencias a *self* y *super*.

Otra razón por la que deberíamos automatizar el refactoring **Move method** es para experimentar si es razonable que el método que estamos moviendo deba ser movido. Realizar el **Move method** de un método y luego darnos cuenta de que tenemos que revertir el cambio, puede ser algo muy frustrante si es algo que toma mucho trabajo por no estar automatizado.

El refactoring **Move method** es reusado por el refactoring **Extract class**. Se utiliza para mover un método que hayamos seleccionado y que deseamos que se encuentre en la nueva clase que se va a crear. De este refactoring vamos a hablar en detalle en las próximas secciones.

5.2. Definición

Este refactoring consiste en mover un método de una clase a otra. Para que se pueda seguir manteniendo el comportamiento se deben satisfacer todas las dependencias que el método necesita para ejecutarse. Estas pueden ser:

- El método referencia una variable de instancia para leerla.
- El método referencia una variable de instancia para escribirla.
- El método referencia una variable de clase para leerla.

- El método referencia una variable de clase para escribirla.
- El método referencia a la pseudo-variable *self*.
- El método referencia a la pseudo-variable *super*.

Una vez realizado el movimiento del método se deben satisfacer todas estas necesidades de lo contrario no va a poder ser ejecutado.

Luego, para simplificar las cosas, hemos decidido que en el lugar donde se encontraba el método originalmente se realice una delegación para que todas las llamadas al método no se vean afectadas.

Haremos una diferenciación entre mover un método de instancia y mover un método de clase: Al mover un método de instancia la clase de destino es referenciada por una variable de instancia. Sin embargo, al mover un método de clase, esta referencia no es necesaria ya que las clases en **Cuis-Smalltalk** pueden ser referenciadas de manera global.

5.3. Precondiciones

Procederemos a listar las precondiciones necesarias para poder realizar el refactoring **Move method**:

- El selector del método que vamos a mover debe existir en la clase de origen.
- El selector del método que vamos a mover no debe existir en la clase de destino. Si es necesario modificar el selector para satisfacer las dependencias del método, entonces, el selector resultante tampoco debe existir.
- La variable de instancia que referencia la clase de destino debe existir en la clase de origen.
- Si es necesario modificar el selector para satisfacer las dependencias del método, entonces, se debe proveer un nombre de parámetro para utilizarlo como referencia a la clase original.
- Si no es necesario modificar el selector para satisfacer las dependencias del método, entonces, no se debe proveer un nombre de parámetro.
- La variable de instancia que referencia la clase de destino debe tener al menos una asignación.

Si al menos una de estas precondiciones no es satisfecha, entonces se lanzará una excepción.

Todas las precondiciones fueron desarrolladas con **TDD** y los test pueden ser encontrados en la categoría de método **validations** de la clase **MoveMethodRefactoringTest**.

5.4. Diagrama

A continuación se muestra un diagrama de clases antes de realizar un **Move method** del método **#homeAddress** fig. 5.1 y un diagrama de clases después de realizar un **Move method** del método **#homeAddress**.

Se dispone de una clase llamada **BankCustomer** la cual tiene un método **#homeAddress** que devuelve una dirección en función de los datos del objeto referenciado por la variable de instancia **livesAt**.

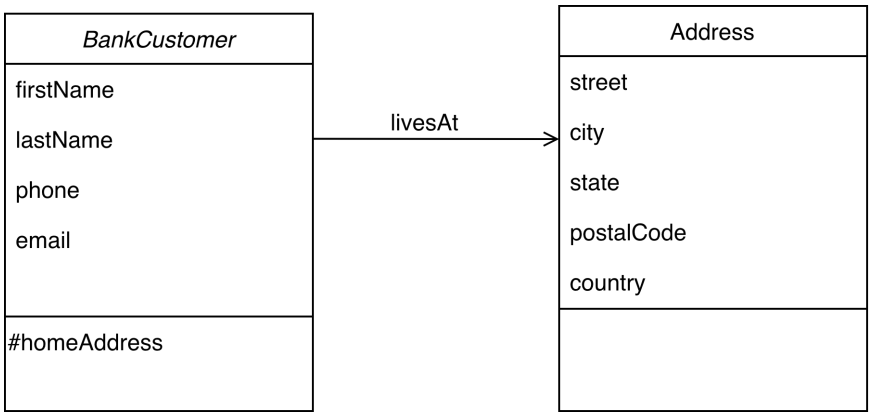


Fig. 5.1: Antes de mover **#homeAddress**

Después de realizar el **Move method** vemos que el selector se encuentra en los dos objetos fig. 5.2. Esto es por que todo el código de método ha sido movido al objeto referenciado por la variable de instancia **livesAt**.

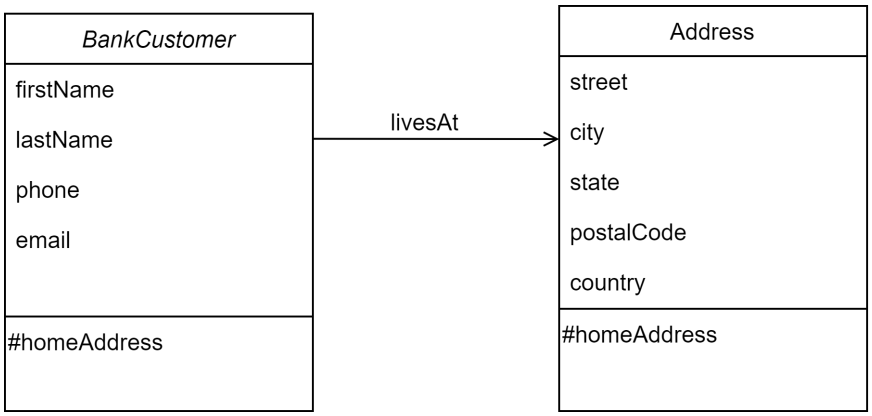


Fig. 5.2: Después de mover **#homeAddress**

Pero para mantener el comportamiento, todas las referencias al método **#homeAddress** tienen que seguir funcionando. Es por eso que el código del método **#homeAddress** del objeto **BankCustomer** se cambia a una simple delegación.

5.5. Implementación

Detallaremos los pasos que se realizan durante la ejecución del **Move method**. Estos pasos son, a grandes rasgos, los que realiza el programa para mover método de una clase a otra y mantener todo el comportamiento del programa:

1. Se instancia el refactoring con los siguientes objetos. Debemos recordar que todas las precondiciones mencionadas anteriormente se deben cumplir. Si no llegan a cumplirse, se lanzará una excepción al momento de hacer esta instanciación:

   a) El selector del método que se desea mover.

   b) La clase a la que el método pertenece.

   c) La clase de destino a la que se desea mover el método.

   d) La variable de instancia que referencia la clase de destino.

   e) Un objeto que representa el nombre que va a tener el colaborador que referencia el objeto en donde vivía originalmente el método (necesario para resolver las dependencias).

2. Se busca el método que se desea mover utilizando el selector con el que se ha instanciado el **Move method**.

3. Se reemplazan todas las lecturas de una variable de instancia con un getter usando como receiver el nombre que se ha pasado por parámetro en el punto 1.e.

4. Se reemplazan todas las escrituras de una variable de instancia con un setter usando como receiver el nombre que se ha pasado por parámetro en el punto 1.e.

5. Se reemplazan todas las lecturas de una variable de clase con un getter usando como receiver el nombre que se ha pasado por parámetro en el punto 1.e.

6. Se reemplazan todas las escrituras de una variable de clase con un setter usando como receiver el nombre que se ha pasado por parámetro en el punto 1.e.

7. Se reemplazan todas las referencias a la variable *self* con el nombre que se ha pasado por parámetro en el punto 1.e.

8. Para cada colaboración a la pseudo-variable *super* se reemplaza con una colaboración con *self* que es una encapsulación de la colaboración original (este paso es detallado mas adelante).

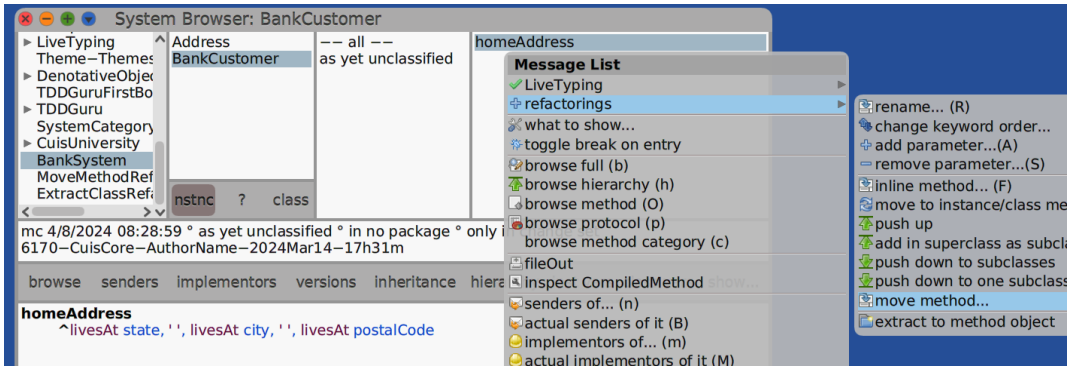
9. Se crean los getters y setter para que el método, después de moverlo, pueda seguir funcionando.

10. Se compila el método con todas sus dependencias cambiadas en la clase de destino que se ha pasado al instanciar el **Move method**.

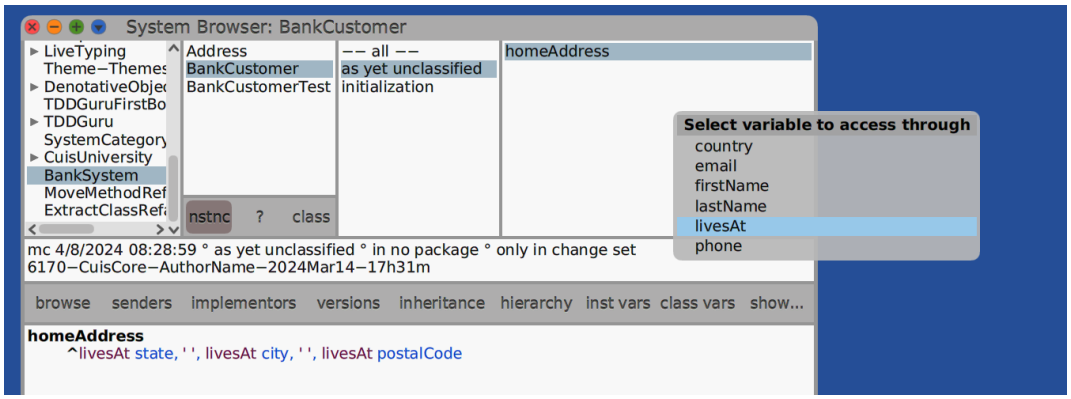
11. Se reemplaza el método original con una delegación al método movido.

5.6. ¿Como se usa?

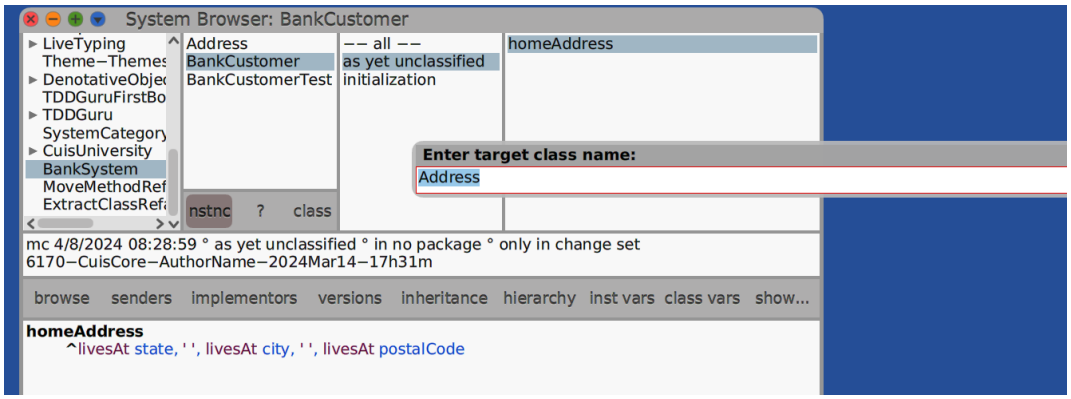
- Se puede acceder al refactoring **Move method** de 2 maneras:
  1. Haciendo click derecho en la lista de métodos del browser.
  2. Haciendo click derecho en selector del método que se encuentra en el código fuente del método del browser.



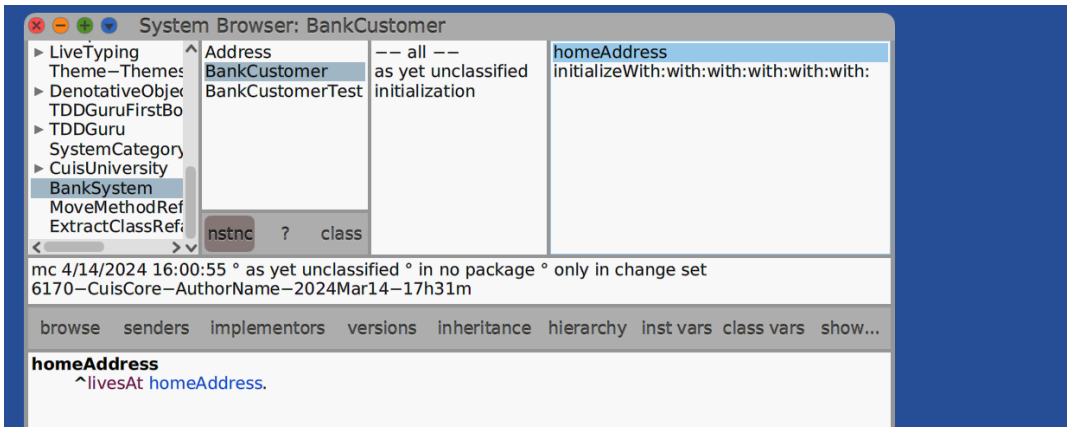
- Luego seleccionamos la variable que referencia al objeto donde vamos a mover el método.



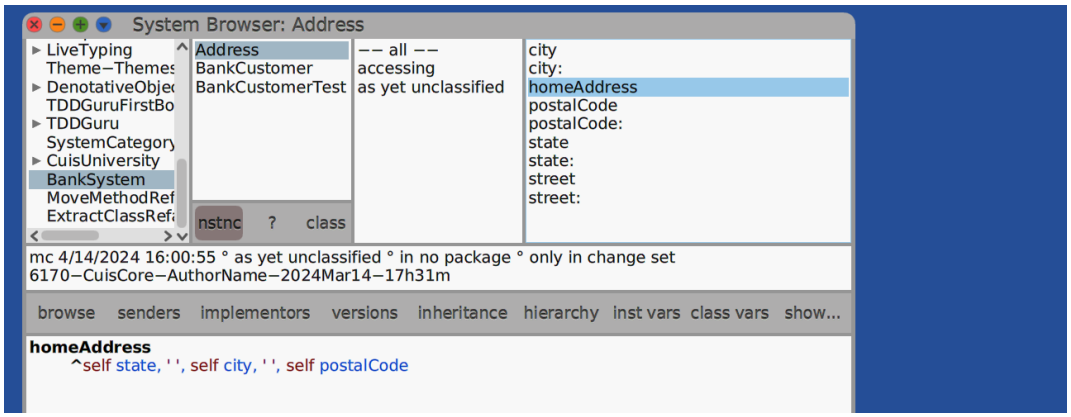
- Escribimos el nombre de la clase la cual pertenece el objeto referenciado por la variable que seleccionamos en el paso anterior. Nuevamente si tenemos **LiveTyping** activado se nos recomendará un nombre de clase.



- Finalmente vemos como el método es cambiado por una delegación debido a que ya ha sido movido a otra clase.



- En la clase de destino podemos ver como el método que hemos movido ha sido cambiado para poder seguir manteniendo el comportamiento.



5.7. Discusión

Hemos mencionado que al realizar un **Move method** no eliminamos el método original, y además, que el método original es cambiado por una delegación para poder satisfacer las llamadas al mismo. Una alternativa a esto es que el método original sea removido y sus llamadas sean cambiadas, sin embargo, no se ha decidido hacerlo de esta manera debido a que esto es posible de hacer a través del refactoring **Inline method**. Pero debemos recordar que este último refactoring posee validaciones y tal vez no sea posible aplicarlo para eliminar todas las llamadas.

Luego de realizar un **Move method** puede existir la necesidad de crear getters y setter en la clase de origen para satisfacer las dependencias del método. Si bien esto rompe el encapsulamiento, es necesario realizarlo para que el método pueda ser ejecutado en la clase de destino.

Al igual que el refactoring **Move instance variable** podríamos resolver algunas dependencias mediante metaprogramación, ya sea usando el mensaje **instVarNamed:** para acceder a las variables de instancia que necesita leer el método o usando el mensaje **perform:withArguments:inSuperclass:** para ejecutar un método de una clase específica. No se ha hecho así debido a que nos dificultaría encontrar las lecturas a una variable de instancia o las llamadas a un método.

Hay veces que el método que queremos mover tiene una referencia a la pseudo-variable *super*. Una forma de resolver esta dependencia es validar que si el método a mover referencia a la pseudo-variable *super* no se debe ejecutar el **Move method**. Si bien esta validación es útil para evitar errores y no afectar el comportamiento (hay IDEs como el Intellij que realizan el refactoring pero la pseudo-variable *super* no es cambiada), creemos que esta decisión recorta el alcance del refactoring, es por esto, que hemos decidido aplicar una solución mencionada en el paper de Jongwook Kim[8]. Si bien es una solución para el lenguaje JAVA, podemos usar la misma idea para poder evitar este recorte de alcance. En pocas palabras lo que hacemos es un encapsulamiento de la colaboración en donde *super* es el receiver a un método con un nombre en particular. Veamos un ejemplo. Supongamos que queremos mover el siguiente método llamado *defaultPrintString*:

```
defaultPrintString
    | tmp |
    tmp := super printString.
    ^tmp
```

Usamos la variable de instancia *destination* como variable que referencia la nueva clase. Entonces el método ha cambiado a la siguiente delegación:

```
defaultPrintString
        ^destination defaultPrintString: self.
```

Pero en la nueva clase el método que acabamos de mover aparece de la siguiente manera:

```
defaultPrintString: sourceObject
    | tmp |
    tmp := sourceObject CHANGE_ME_super_printString.
    ^tmp
```

Vemos que la colaboración «super printString.» ha cambiado a «sourceObject CHANGE_ME_super_printString.». Esto es por que ha sido encapsulada a un nuevo método en el que el selector es de la forma «CHANGE_ME_super_{nombre de colaborador}». Entonces, para poder seguir manteniendo el comportamiento debemos crear un nuevo método en la clase de origen:

```
CHANGE_ME_super_printString
        ^super printString.
```

Otra decisión con respecto a los getters y setters es que sucede si se necesita crearlo pero existe el selector y este no es un getter o setter en la clase de origen. A diferencia del **Move instance variable**, lo que hemos decidido hacer es agregar un guión bajo al final del nombre del selector. Si bien puede ser un poco inconsistente con respecto a la solución que se ha discutido en el **Move instance variable**, creemos que sería demasiado complicado hacer que el usuario provea un nombre para cada getter y setter que necesite el método que vamos a mover previo a la ejecución del refactoring.

5.8. Trabajo futuro

En lo que respecta a la UI, podemos agregar que se le consulte al usuario si desea realizar un **Inline method** luego de la ejecución del **Move method**.

Otra cosa que podemos mejorar es el reemplazo de las referencias a *self* (mencionado en el paso 7 de la sección implementación). Si pensamos en métodos recursivos es posible que no tenga sentido reemplazar las referencias *self*, ya que el método se llama a si mismo.

Finalmente, en el caso de que se desea mover un método que referencia muchas variables de instancia, sería bueno no romper tanto el encapsulamiento al crear un getter y un setter por cada una de esas variables. Queda como trabajo futuro buscar una alternativa a este problema.

6. EXTRACT CLASS

Absolutely one of my favorite moments in software design
–when a complex computation reveals a new abstraction.
–Kent Beck

En esta sección vamos a ver nuestra implementación del refactoring **Extract class** para **Cuis-Smalltalk**.

6.1. Motivación

Muchas veces, en nuestro proceso iterativo de aumentar las funcionalidades de nuestro software, agregamos un método, o una variable de instancia, o ambos a una clase en particular. Si únicamente nos preocupamos por agregar estos elementos y no nos preocupamos por modelar mejor los conceptos que vamos adquiriendo, terminaremos con una clase muy grande y difícil de mantener[3].

Supongamos que tenemos una clase muy grande, difícil de entender y con muchas responsabilidades. Entonces, queremos usar los refactoring **Move instance variable** y **Move method** para dividir sus responsabilidades en clases mas chicas. El problema que tenemos con estos refactorings es que solo pueden mover una sola variable de instancia o un solo método a la vez respectivamente. Esto hace que mover cada una de las variables de instancia y cada uno de los métodos a una clase nueva sea una tarea muy tediosa.

6.2. Definición

Este refactoring se trata de generar una clase nueva a partir de una clase existente. Para esto debemos proporcionarle los siguientes elementos:

- Un nombre de clase para la nueva clase que se va a crear.
- Un nuevo nombre de variable de instancia que va a referenciar la nueva clase.
- Un conjunto de variables de instancia a mover.
- Un conjunto de métodos a mover.

Con estos elementos se generará una nueva clase que tendrá las variables de instancia que hemos seleccionado y los métodos que hemos seleccionado. La nueva variable de instancia que hemos creado en la clase de origen debe ser inicializada con la nueva clase que hemos creado.

Luego todas las dependencias para mover cada una de las variables de instancia y cada uno de los métodos deben ser satisfechas para que todos los comportamientos no sean alterados. Estas dependencias las hemos nombrado en la sección de definición en los capítulos anteriores.

En nuestra implementación el refactoring **Extract class** es un refactoring de alto nivel que utiliza los refactoring **Move instance variable** para mover un conjunto de variables de instancia y **Move method** para mover un conjunto de métodos.

6.3. Precondiciones

Este refactoring crea una clase nueva que no existe en la imagen. Esto es muy beneficioso a la hora de enumerar las precondiciones, ya que, al no existir la clase nueva, no hay muchas precondiciones que pedir al momento de mover los elementos. Entonces, las precondiciones necesarias para poder ejecutar satisfactoriamente el refactoring **Extract class** son:

- No debe existir la clase nueva que vamos a crear.
- La clase de origen debe contener todas las variables de instancia a mover.
- La clase de origen debe contener todos los métodos a mover.
- La variable de instancia que referencia la nueva clase que vamos a crear, no debe existir en la clase de origen.

Si al menos una de estas precondiciones no es satisfecha, entonces se lanzará una excepción.

Todas las precondiciones fueron desarrolladas con **TDD** y los test pueden ser encontrados en la categoría de método **validations** de la clase **ExtractClassRefactoringTest**.

6.4. Diagrama

A continuación se muestra un diagrama de clases antes de realizar un **Extract class** de la información de contacto de nuestro **BankCustomer** y un diagrama después de la ejecución del refactoring **Extract class**.

Vemos que la clase llamada **BankCustomer** la hemos hecho crecer demasiado, analizando la misma, nos damos cuenta que la información de contacto del cliente podría ser abstraída a un nuevo objeto (ver fig. 6.1). Este nuevo objeto podría generarse con la ejecución del refactoring **Extract class**. Como este nuevo objeto va a contener la información de contacto decidimos llamarlo **ContactInformation**.

Decidimos ejecutar el refactoring **Extract class** para poder extraer los siguientes elementos y llevarlos a la nueva clase **ContactInformation**:

- La variable de instancia **phone** que referencia el teléfono de nuestro cliente.
- La variable de instancia **areaCode** que referencia el código de área de nuestro cliente.
- La variable de instancia **email** que referencia la dirección de correo electrónico de nuestro cliente.

- El método **#contactNumber** que sirve para mostrar el teléfono con su código de área.
- El método **#fullContactInfo** que sirve para mostrar toda la información de contacto del cliente.

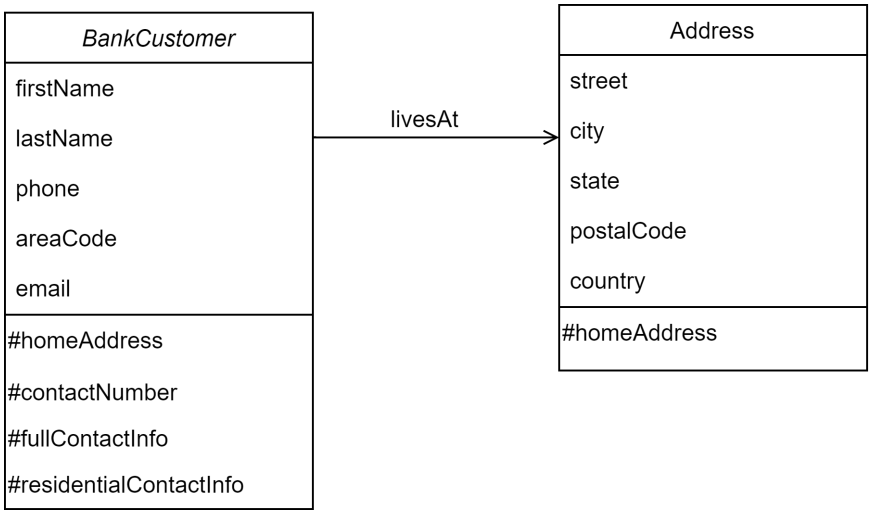


Fig. 6.1: Antes realizar Extract class de la información de contacto

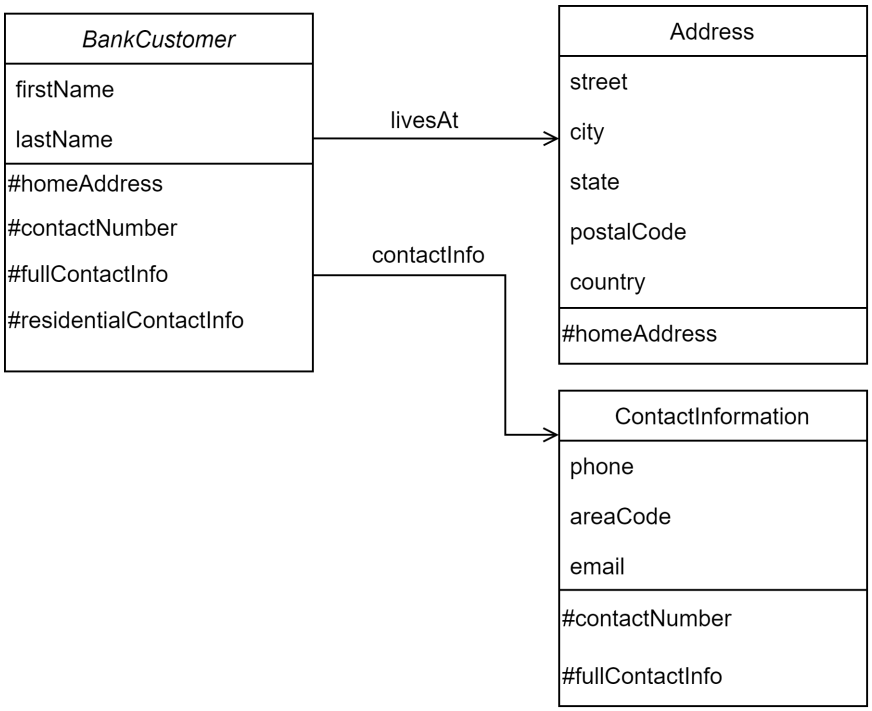


Fig. 6.2: Después realizar Extract class de la información de contacto

Una vez seleccionados todos los elementos que queremos extraer de la clase, ejecutamos el refactoring. Entonces terminamos obteniendo como resultado la clase **ContactInformation** (ver fig. 6.2).

En la clase **BankCustomer** vemos que las variables de instancia que hemos seleccionado han sido movidas a la clase **ContactInformation**. Mientras que los métodos que hemos seleccionado han sido delegados a la clase **ContactInformation**. Además tenemos una nueva variable de instancia llamada **contactInfo** que referencia una instancia de la nueva clase **ContactInformation**.

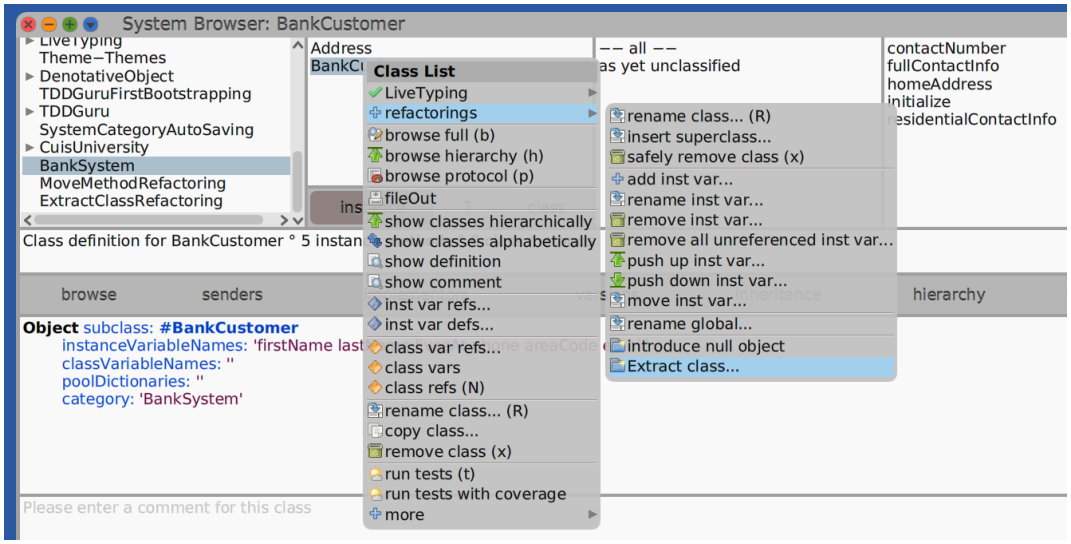
6.5. Implementación

Detallaremos la secuencia de pasos al ejecutar el **Extract class**. Estos pasos son, a grandes rasgos, los que realiza el programa para crear una nueva clase, mover todas las variables de instancia seleccionadas y todos los métodos seleccionados a esta nueva clase.

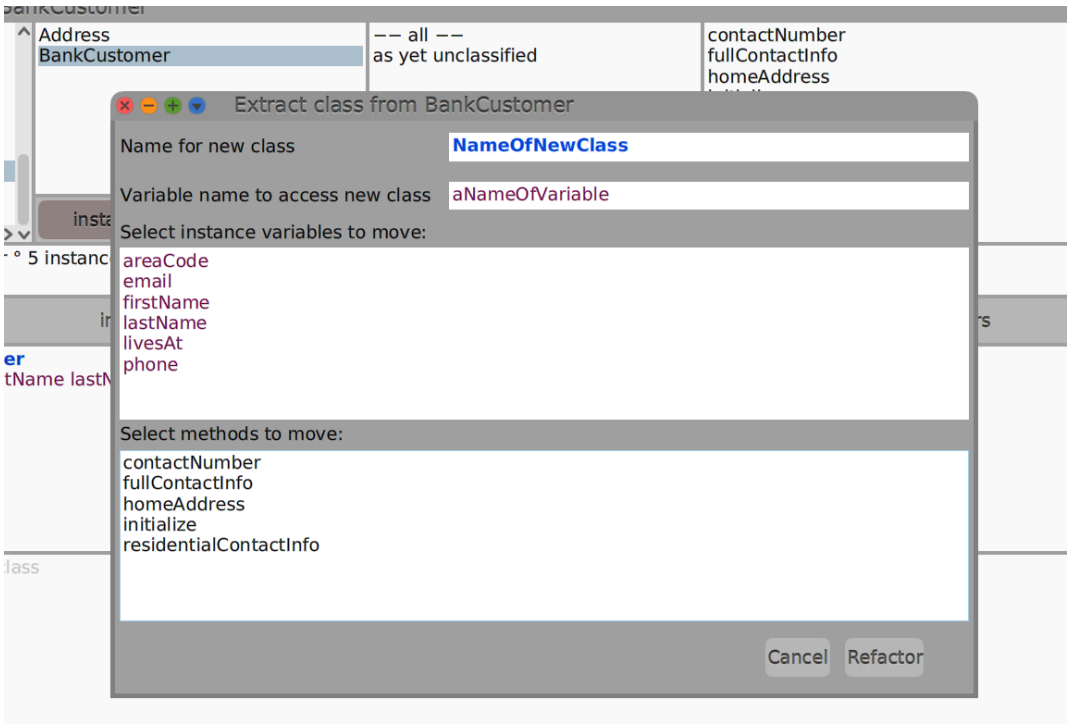
1. Se instancia el refactoring con los siguientes objetos. Debemos recordar que todas las precondiciones mencionadas anteriormente se deben cumplir. Si no llegan a cumplirse, se lanzará una excepción al momento de hacer esta instanciación:
   - La clase de donde se van a extraer las variables de instancia y los métodos.
   - El nombre de la nueva clase que se va a crear y a donde se llevarán las variables de instancia y métodos extraídos.
   - Un nombre para la nueva variable de instancia que se va a crear. Esta variable va a ser inicializada con una instancia de la nueva clase.
   - Una lista de las variables de instancia que se van a extraer a la nueva clase.
   - Una lista de los métodos que se van a extraer a la nueva clase.
2. Se crea la nueva clase a la que se van a mover las variables de instancia y los métodos con el nombre que se ha provisto en el paso anterior.
3. En la clase de origen, se modifica o crea el método **#initialize** para la inicialización de la nueva variable de instancia. Se ha decidido hacerlo aquí debido a que siempre se envía el mensaje **#initialize** inmediatamente después de que se crea un objeto.
4. Se realiza el refactoring **Move instance variable** a la nueva clase por cada una de las variables de instancia que se encuentran en la lista de variables de instancia.
5. Se realiza el refactoring **Move method** a la nueva clase por cada método que se encuentra en la lista de métodos.

6.6. ¿Como se usa?

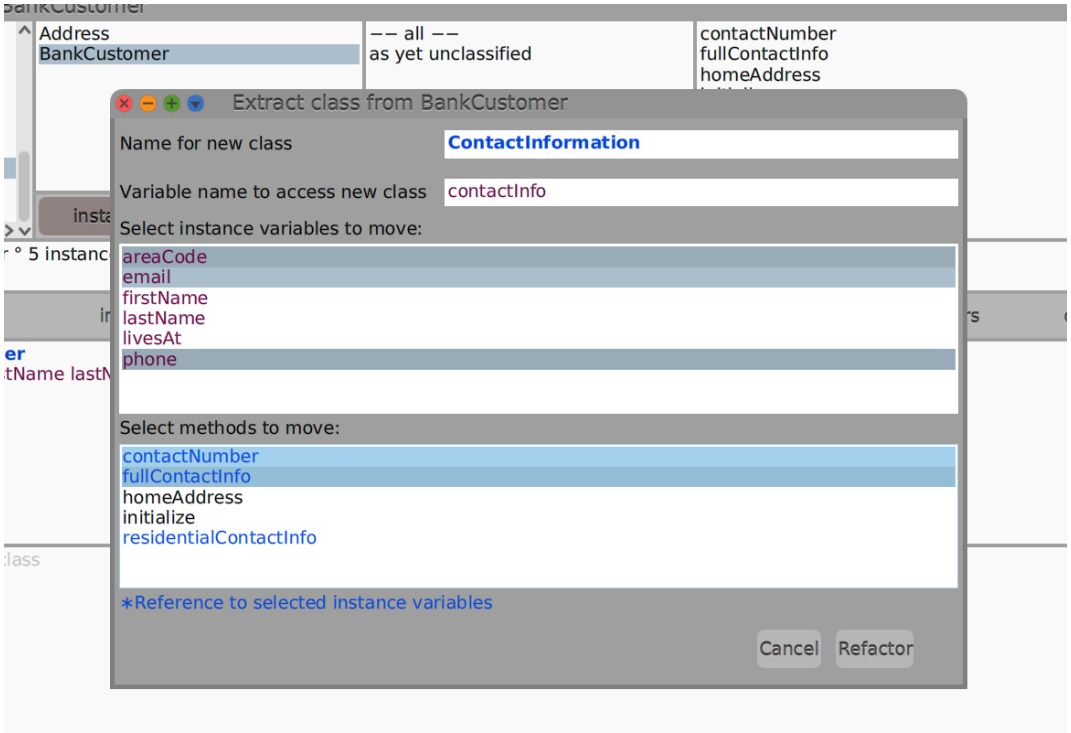
- Para acceder al refactoring hay que hacer click derecho en el browser sobre la clase que uno desea realizar el **Extract class**. Seleccionar la opción «Refactorings» y luego seleccionar la opción «Extract class...»



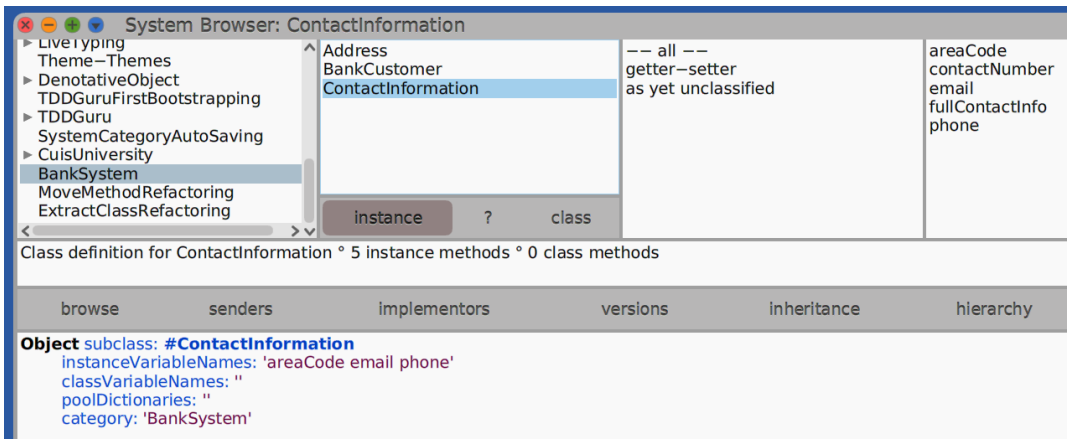
- Luego se abrirá un cuadro en donde debemos ingresar:
  - El nombre de la nueva clase a crear.
  - El nombre de la nueva variable de instancia a crear.
  - Seleccionar las variables de instancia que deseamos mover a la nueva clase.
  - Seleccionar los métodos que deseamos mover a la nueva clase.



- Podemos ver que al seleccionar una variable de instancia la UI nos avisa que métodos referencian las variables seleccionadas.



- Finalmente podemos ver como cada variable de instancia que seleccionamos se ha movido a la clase nueva y cada método que hemos seleccionado ha sido cambiado por una delegación.



6.7. Discusión

Una de las cosas que debemos mencionar de esta implementación es que al reutilizar tanto el **Move instance variable** como el **Move method** hacemos uso de estas dos implementaciones de la misma manera en que fueron hechas. Si bien esto simplifica la complejidad de la implementación, también debemos decir que siempre vamos a estar atados a lo que nos provean estas implementaciones.

Podemos observar que si no se selecciona ninguna variable de instancia y ningún método a extraer, entonces, se creará la nueva clase únicamente.

Una de las dificultades al implementar el refactoring **Extract class** es la de ¿Que sucede cuando extraemos una variable de instancia y un método que tiene el mismo nombre? Hemos mencionado que en Smalltalk es una convención que el selector de un getter y su variable de instancia usen el mismo nombre. Recordemos que esta implementación primero mueve las variables de instancia y luego mueve los métodos. El problema que tenemos aquí es que al mover la variable de instancia primero, el **Move instance variable** es el que va a crear un getter en la clase nueva para que los métodos que leen esa variable puedan seguir funcionando. Pero, luego al intentar mover el método con el **Move method** no se va a poder, por que ya existe un método con el mismo nombre en la clase nueva que acabamos de crear. Para solucionar esto primero detectamos que al ejecutar el **Extract class** hay una variable de instancia y un método con el mismo nombre. Luego, ejecutamos el **Move instance variable** pero con el segundo constructor que nos permite poder generar getters y setters de la variable pero fuera de la convención de Smalltalk. Entonces, en la nueva clase, tenemos la variable de instancia, el getter (con un nombre distinto) y el método que queríamos mover.

6.8. Trabajo futuro

Una de las mejoras que se podría hacer es la de tratar de favorecer el encapsulamiento. Si las variables de instancia que intentamos extraer son seteadas en único método, tal vez se podría prescindir de los setter que necesitamos crear para reemplazar las escrituras a dichas variables.

Otra de las mejoras que podríamos hacer al refactoring **Extract class** es la de extraer variables de instancia y métodos a una clase existente. Si bien el trabajo de mover las variables de instancia y los métodos es muy parecido a los refactorings que se han implementado en las secciones anteriores, debemos mencionar que las validaciones de esos refactorings deben ser cambiadas para que el refactoring **Extract class** pueda tomar distintas decisiones.

7. CONCLUSIONES

En este trabajo hemos visto la implementación de los refactorings **Move instance variable**, **Move method** y **Extract class**. Si bien tratan de mantener el comportamiento como cualquier refactoring, no podemos dejar de mencionar que no son una herramienta infalible y hemos visto que pueden existir casos el los que es inevitable que fallen. Sin embargo, la implementación de un refactoring es un proceso iterativo e incremental, y por lo tanto, estos pueden ser usados aunque no funcionen en la totalidad de los casos y se podrían seguir mejorando de manera progresiva.

El desarrollo de refactorings automatizados es un desafío aún mas grande en ambientes dinámicamente tipados como **Cuis-Smalltalk**. Esto provoca que la cantidad de potenciales bugs o mejoras a futuro sea aun mas alta, ya que no se tiene la información de tipos para determinar como se debe actuar en cada caso.

También vimos como la herramienta **LiveTyping** nos puede dar pistas de que decisiones debemos tomar en base a la información de tipos recolectada. Si bien solo se ha usado para autocompletar el nombre de clases, sería bueno seguir investigando que otros usos se le podría dar para mejorar los refactorings en un lenguaje dinámicamente tipado.

Además, se ha visto que modificar código con refactorings no nos exime de utilizar buenas heurísticas de diseño, como por ejemplo: no romper el encapsulamiento o no utilizar nulls.

A pesar de esto, queremos mencionar que la posibilidad de usar los refactorings presentados en esta tesis son una herramienta más a la hora de hacer que el código sea mas comprensible para las personas. Especialmente el refactoring **Extract class** es el que disminuye la complejidad de una clase delegando algunas funcionalidades a una nueva clase.

Como John Brant dice en su articulo[2], es mucho mas importante tener una herramienta imperfecta que ayude a los desarrolladores a trabajar más eficientemente, a que tratar de lograr la correctitud en lo que es preservar el comportamiento.

Bibliografía

[1] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002.

[2] John Brant and Friedrich Steimann. Refactoring tools are trustworthy enough and trust must be earned. *IEEE Software*, 32(6):80–83, 2015.

[3] Maximiliano Contieri. God object, 2023. `https://blog.devgenius.io/code-smell-14-god-objects-b84b75b702`.

[4] Matias Dinotaw. Tdd-gurú, 2018. `https://github.com/mdinota/TDDGuru`.

[5] Eclipse Foundation. Eclipse ide, 2024. `https://eclipseide.org/`.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[7] JetBrains. Intellij idea, 2024. `https://www.jetbrains.com/`.

[8] Jongwook Kim, Don Batory, and Danny Dig. Scripting parametric refactorings in java to retrofit design patterns. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 211–220, 2015.

[9] Jongwook Kim, Don Batory, and Milos Gligoric. Code transformation issues in move-instance-method refactorings. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 17–22, 2019.

[10] Emerson Murphy-Hill. Programmer-friendly refactoring tools. 07 2012.

[11] Emerson Murphy-Hill, Chris Parnin, and Andrew Black. How we refactor, and how we know it. volume 38, 05 2009.

[12] William Opdyke and Ralph Johnson. Refactoring object-oriented frameworks. 07 1992.

[13] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, pages 3–4, 1997.

[14] Norihisa Suzuki. Inferring types in smalltalk. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 187–199, New York, NY, USA, 1981. Association for Computing Machinery.

[15] Juan Vuletich and the Cuis community. Cuis-smalltalk, 2024. `https://cuis.st/`.

[16] Hernan Wilkinson. Denotative objects, 2017. `https://github.com/hernanwilkinson/Cuis-Smalltalk-DenotativeObject`.

[17] Hernan Wilkinson. Live typing, 2018. `https://github.com/hernanwilkinson/LiveTyping`.

[18] Hernan Wilkinson and the Cuis university community. Cuis-university, 2024. https://sites.google.com/view/cuis-university.

[19] Hernan Wilkinson and Juan Vuletich. Aconcagua, 2005. https://github.com/Cuis-Smalltalk/Measures/tree/master/Aconcagua.

[20] Hernan Wilkinson and Juan Vuletich. Chalten, 2005. https://github.com/Cuis-Smalltalk/Calendars/tree/master/Chalten.

[21] H. Fernandes with K. Dickey and J. Vuletich. *The Cuis Book*. 2024. https://cuis-smalltalk.github.io/TheCuisBook/.