Un lenguaje modelo para la definición y análisis de refactorizaciones

Yania Crespo Valentín Cardeñoso José Manuel Marqués
Departamento de Informática. Universidad de Valladolid *
{*yania, valen, jmmc*}*@infor.uva.es*

Resumen

En este trabajo se propone el lenguaje modelo MOON con suficiente poder expresivo para representar las construcciones abstractas necesarias en la definición y análisis de refactorizaciones. Gracias a este modelo, el esfuerzo de refactorización para un lenguaje particular se reparte entre la tarea de representar en MOON el código y la de refactorización propiamente dicha en el lenguaje modelo. Este último esfuerzo, no obstante, es claramente reutilizable, lo que hace que en conjunto el empleo de nuestro lenguaje modelo reduzca sustancialmente el esfuerzo de refactorización. También se ha definido una arquitectura y un proceso para integrar el modelo en herramientas de refactorización. Como caso concreto se detalla la reducción a MOON de un lenguaje de programación bien conocido y se ilustra la viabilidad de nuestra propuesta describiendo un algoritmo de refactorización concreto.

Palabras clave: transformación de programas, refactorización, abstracción de lenguajes, modelo de familia de lenguajes.

1 Introducción

El propósito fundamental de esta comunicación es presentar brevemente un lenguaje modelo definido con un objetivo muy concreto, facilitar la labor de la definición y análisis de refactorizaciones y avanzar hacia la posibilidad de establecer una cierta independencia del lenguaje en este campo. Fundamentalmente presentaremos el análisis razonado de las decisiones del diseño del lenguaje modelo y la forma en que este apoya las tareas de refactorización. Hemos empleado el modelo en una aplicación concreta que se presenta como caso de estudio de la factibilidad de proseguir por este camino.

En lo que sigue, la presentación se organiza de la siguiente forma. La Sección 2 introduce el problema de la refactorización y la dependencia del lenguaje de los elementos objetos de la transformación. En la Sección 3 se exponen otros trabajos que se relacionan con la aproximación que hemos tomado. Posteriormente en la Sección 4 se define y se analiza el modelo propuesto y en la Sección 5 se explica el papel del modelo en la definición y análisis de refactorizaciones. La Sección 6 es la presentación de un caso de estudio, mientras que en la Sección 7 se concluye.

2 Refactorización de software

El término **refactorización** fue introducido por primera vez por Opdyke en [1]. Las **refactorizaciones** son transformaciones de elementos de software Orientado a Objetos (OO) que, realizando reestructuración y reorganización, preservan el comportamiento. Reestructurar es transformar la estructura interna de

*Este trabajo ha sido parcialmente financiado por el proyect o DOLMEN (CICYT TIC-2000-1673-c06-05).

un elemento de software. Una reestructuración que transforma la manera en que se relacionan diferentes elementos se dice una reorganización.

En el área de la refactorización de software nos encontramos con una intensa actividad actualmente. En este campo se trabaja para abordar diferentes aspectos tales como la propia definición de operaciones de refactorización [1], la construcción de herramientas que apoyen la ejecución de refactorizaciones [2], la elaboración de catálogos de "patrones" de refactorización que ayuden a los desarrolladores a decidir cuándo refactorizar y qué operaciones aplicar [3], la incorporación de técnicas para poder llegar a inferir automáticamente cuándo y qué refactorizar [4]. También se encuentran trabajos centrados en el análisis y demostración de preservación del comportamiento al aplicar una refactorización [5] y en la introducción en el ciclo de vida de métodos de desarrollo [6].

Definir una refactorización pasa por definir las operaciones a realizar, mostrar que las reglas de validez se preservan y proponer cuándo ejecutar la refactorización. Finalmente, para que una refactorización definida pueda ser aplicada eficazmente, es deseable contar con una herramienta que la realice automáticamente o al menos asista en su ejecución.

La herramienta debe garantizar que a partir de elementos correctos, una vez aplicada la transformación, se obtengan elementos correctos. Para esto es necesario conocer de forma precisa la estructura de los elementos a transformar y sus reglas de validez. Esto ha condicionado que los trabajos realizados hasta el momento en refactorización sean dependientes del lenguaje que define a los elementos objeto de las transformaciones, haciendo que se multipliquen los esfuerzos en la definición de las refactorizaciones por cada lenguaje en particular y por cada proceso de desarrollo de las herramientas que las ejecutan.

Sin embargo, la experiencia en el área nos lleva a las siguientes ideas. En primer lugar, las refactorizaciones no modifican el comportamiento sino la arquitectura que se ha diseñado para la solución, de modo que no todas las construcciones de los lenguajes son importantes a la hora de definir e implementar refactorizaciones. Por otra parte, de la experimentación con diferentes lenguajes podemos detectar que aunque dichos lenguajes tengan características diferentes, si enfocamos aquellas que han sido fundamentales para definir e implementar una refactorización, encontramos muchos conceptos comunes. La perspectiva esbozada por las ideas anteriores se resume en que podemos avanzar hacia establecer, al menos, una cierta independencia del lenguaje.

Nuestro enfoque para abordar esta nueva perspectiva consiste en concentrarnos en los aspectos comunes y en las construcciones que interesan desde el punto de vista de la refactorización, definiendo un lenguaje modelo que responda a estas características. De esta forma, contando con dicho modelo, como primer paso, podemos definir las refactorizaciones para las construcciones del modelo y reducir los esfuerzos, de particularizar para un lenguaje, a mirar cada lenguaje por el prisma del modelo.

Otros aspecto muy importante, y muchas veces pasado por alto, en el ámbito de las refactorizaciones es el análisis de las consecuencias que tiene su aplicación para los elementos que dependen de los transformados y para los objetos persistentes que fueron creados previos a la modificación. La definición de las refactorizaciones sobre el modelo también ayuda a abordar esta tarea. En [7] hemos definido un lenguaje modelo para el análisis y definición de refactorizaciones, que posteriormente hemos aplicado en casos de estudio.

3 Trabajos relacionados

En [8] se presenta el modelo FAMIX como un meta-modelo para almacenar información en un repositorio que permita la integración de di-

ferentes entornos de desarrollo de software con soporte para la refactorización y por otro lado se presenta un estudio de factibilidad, a partir del análisis de dos lenguajes, JAVA y SMALLTALK, para validar su propuesta de metamodelo que abstrae las características necesarias para realizar refactorizaciones. JAVA y SMALLTALK son lenguajes con diferencias muy marcadas como la presencia de meta-clases en SMALLTALK y de interfaces en JAVA, y de tipado estático en JAVA y dinámico en SMALLTALK. De esta forma obtienen un modelo que considera la abstracción de estas características.

En cambio en nuestro modelo nos hemos concentrado en abstraer características de lenguajes basados en clases, estática y fuertemente tipados y con genericidad, haciendo especial énfasis en la consideración de características avanzadas en cuanto a herencia y genericidad. Hemos comprobado en la implementación para JAVA de la refactorización que definimos en [9], que la adaptación para considerar interfaces a lo JAVA no es un problema para nuestro modelo, ni la ausencia de tipos. Sin embargo creemos que la consideración de meta-clases es un problema mayor para nuestro modelo, al igual que la inclusión en FAMIX de genericidad al mismo nivel que en el nuestro, es también un problema mayor. Una línea interesante sería intentar fusionar ambos modelos.

Hasta donde conocemos, éste es el único trabajo que se ha publicado con un propósito igual al nuestro: ir hacia un motor de refactorizaciones independiente del lenguaje.

La idea, en el sentido de brindar soporte para múltiples lenguajes en una herramienta de refactorización, fue esbozada en [10], aunque también puede inspirarse en las direcciones de investigación que propone [11] en lo que denomina "tecnologías para lenguajes genéricos".

A continuación presentamos el enfoque de nuestra propuesta mediante la descripción, primeramente, del lenguaje modelo definido y su posterior aplicación a la definición, el análisis y la implementación de refactorizaciones, mostrando finalmente un caso de estudio.

4 El lenguaje MOON

La esencia del lenguaje modelo que hemos propuesto radica en intentar ser **minimal**, en el sentido de enfocar solamente en las construcciones que aportan información fundamental a la hora de definir y analizar refactorizaciones, y a la vez intentar que sea lo más **general** posible para dar cabida a una amplia familia de lenguajes. Con este objetivo se han estudiado un conjunto de lenguajes para abstraer sus características. El resultado de dicho estudio se puede consultar en [7].

Con este objetivo se han definido variantes en el modelo. La variación se centra en las reglas de tipos. Tener bien definidas las reglas de las construcciones del lenguaje y del sistema de tipos permiten analizar si las refactorizaciones, una vez aplicadas, dan lugar a elementos correctos en cuanto a estructura y tipos.

El lenguaje modelo minimal-general que se define se denomina MOON[1]. MOON no intenta ser un lenguaje para programar sino para abstraer cómo se representan estructuralmente los principales conceptos y construcciones a tener en cuenta para definir y analizar refactorizaciones. En este lenguaje las clases son implementaciones implícitas de tipos y se considera la presencia de herencia múltiple. La estructura de los tipos que dichas clases implementan, estará dada por un modelo de especificación de tipos basado en estructura y signatura.

MOON se basa en ISTBOPL, lenguaje definido en [12]. El lenguaje ISTBOPL es una extensión que se hace a partir de otro lenguaje nombrado BOPL[2], añadiendo a este último:

[1] MOON es un acrónimo de *Minimal Object-Oriented Notation*.

[2] BOPL e ISTBOPL son acrónimos de *Basic Object Programming Language* y de *Inheritance, Substitution and Types for BOPL*, respectivamente.

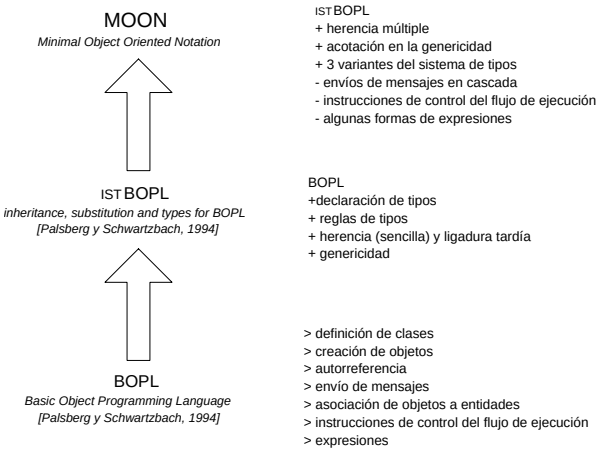

Figura 1: *Precedentes de* MOON.

declaración de tipos, herencia y genericidad. BOPL fue definido como un lenguaje básico a partir del cual se van añadiendo propiedades, hasta llegar a istBOPL. BOPL es un lenguaje basado en clases mientras que istBOPL es un lenguaje OO, según la clasificación definida en [13][3]. Estos lenguajes fueron diseñados para ilustrar construcciones básicas, son lenguajes básicos sin adornos. En la Figura 1 se muestra la relación de las características de MOON respecto a sus antecesores.

Se ha decidido basar MOON en istBOPL porque este último constituye un tronco común a la familia de lenguajes estudiados y porque ya asume simplificaciones que nos ayudan en el intento de buscar minimalidad. La presentación de MOON la haremos en dos partes, la primera refleja la búsqueda de aún más minimalidad a partir de istBOPL y la segunda la obtención de más generalidad a partir de la consideración de más complejidad en la herencia, la genericidad y el sistema de tipos.

[3]Breve resumen de la clasificación de Wegner: 1- lenguaje basado en objetos, ofrece construcciones para definir objetos, 2- lenguaje basado en clases, ofrece construcciones para definir clases y objetos, 3- lenguaje orientado a objetos, ofrece construcciones para definir clases, objetos e incorpora el mecanismo de herencia.

4.1 Simplificaciones a partir de istBOPL

En el modelo MOON se proporciona una gramática de la sintaxis concreta para la descripción de las clases, definiciones de la sintaxis abstracta y reglas formales del sistema de tipos (comunes y variantes). A partir de la gramática de la descripción de una clase se proporciona la definición de la estructura de los objetos y de los objetos persistentes.

La gramática que se presenta en el Apéndice A define la sintaxis concreta de MOON y por tanto determina la estructura de las clases que se pueden definir.

Un módulo en MOON se corresponde con la definición de una clase (regla 1). Cada clase tiene atributos y métodos que pueden ser rutinas o funciones. Una rutina ejecuta un conjunto de instrucciones y una función computa un resultado. Los atributos y los métodos definen las propiedades de las clases. Las signaturas de las propiedades intrínsecas de las clases se declaran en el grupo de signaturas (reglas 5 y 13). La regla 23 define las anotaciones de tipo que se pueden hacer en MOON, a la que conducen las declaraciones de entidades (reglas 19 y 22).

Sintácticamente un envío de mensaje a una entidad se denota, como es habitual, con la notación de punto, es decir: entidad.mensaje. En MOON se define sin pérdida de generalidad, que no se permiten envíos de mensajes (llamadas) en cascada. Todo envío de mensajes en cascada se puede transformar en una sucesión de envíos de mensajes. Las llamadas en cascada, $\mathsf{e}_1.\mathsf{e}_2 \ldots \mathsf{e}_n$ $(n > 2)$, que aparezcan en los elementos de un lenguaje en particular, que se mira desde el prisma de MOON, al margen de los argumentos que pueda tener cada mensaje involucrado en la cascada, se pueden reducir considerando la siguiente secuencia de instrucciones MOON:

$\mathsf{t}_1 := \mathsf{e}_1.\mathsf{e}_2;$
$\mathsf{t}_2 := \mathsf{t}_1.\mathsf{e}_3; \ \ldots$
$\mathsf{t}_{i-1} := \mathsf{t}_{i-2}.\mathsf{e}_i; \ (2 < i < n)$

si $e_1.e_2 \ldots e_n$ es una expresión, $t_{n-1} := t_{n-2}.e_n$
si $e_1.e_2 \ldots e_n$ es una instrucción, $t_{n-2}.e_n$ (1)
Esta reducción del envío de mensajes en cascada no va encaminada a dar una alternativa al tratamiento dinámico ni a la generación de código, sino a hacer un tratamiento estático simplificado de la cascada. De esta forma las dependencias implícitas de tipos que ocasiona una cascada se eliminan, quedando todas las dependencias expresadas de forma clara y explícita como propone la ley de Demeter [14]. La eliminación de las llamadas en cascada está recogida en las reglas 36 y 41.

Por otra parte, los argumentos reales en los envíos de mensajes solamente podrán venir dados por expresiones atómicas (una entidad o una constante manifiesta). Este caso tampoco implica una pérdida de generalidad puesto que si se quiere analizar la presencia de una expresión no atómica $exp \triangleq e_1 \ldots e_n$ $(n > 1)$ como argumento real en un envío de mensaje tal que $a.f(e_1 \ldots e_n)$, el análisis se puede hacer sobre la aplicación de (1) a $e_1 \ldots e_n$ y la sustitución $a.f(t_{n-1})$. Esta consideración se refleja en la regla 44.

En el lenguaje ISTBOPL, que se tomó como punto de partida para definir MOON, las formas de expresiones admitidas son: expresiones atómicas, expresiones de envío de mensajes, expresiones binarias y expresiones unarias. De las formas anteriores de expresiones, en MOON no se incluyen las dos últimas (regla 39). Las expresiones binarias o unarias se pueden considerar como una expresión de envío de mensaje (e.g. a+b como a.+(b)).

También en ISTBOPL el conjunto de instrucciones está formado por: instrucción compuesta, instrucción de asignación, instrucción de creación de objetos, instrucción de envío de mensajes, instrucción condicional (**if** EXPR **then** INSTR **else** INSTR) e instrucción de repetición (**while** EXPR **do** INSTR). De este conjunto de instrucciones, en MOON se eliminan las dos últimas, condicional y de repetición que son las instrucciones de control del flujo de ejecución definidas en ISTBOPL (regla 32). Esto se debe a que desde el punto de vista de los análisis que interesan a la hora de refactorizar, la presencia en un lenguaje particular de instrucciones como estas se miran desde el prisma de MOON como:
c_1: Boolean; c_2: Boolean;
c_1 := EXPR; $c_2 := c_1$.eq(BOOLEAN_CONSTANT);
INSTR
De esta forma se pueden analizar claramente las relaciones de dependencias entre clases, las asociaciones de entidades, la corrección en cuanto a tipos, etc., sin prestar atención a más detalles.

4.2 Reglas y variantes del sistema de tipos de MOON

A partir de ISTBOPL, para definir MOON en la búsqueda de reflejar la riqueza de conceptos de una mayor familia de lenguajes, se añade la capacidad de declarar herencia múltiple de clases y una amplia gama de modificadores de herencia, se añade también la capacidad de acotar los parámetros formales de una clase genérica y se enriquece el sistema de tipos considerando algunas variantes.

En el sistema de tipos de MOON:

- se considera la existencia un tipo universal U que es implementado por una clase predefinida de nombre Object. Toda clase hereda de Object de forma implícita, directa o indirectamente.
- se considera la existencia de clases predefinidas que son implementación de tipos básicos de objetos.
- no se definen relaciones entre los tipos básicos.
- se define un núcleo de reglas comunes y variantes.

Las reglas comunes se definen de forma similar al sistema de tipos de FUN descrito en [15], teniendo en cuenta las observaciones aportadas en [16], y la correspondencia de renombramien-

to que se utiliza en la definición de subtipo de [17]. En la variante *1)* las reglas de la herencia y la asignación polimórfica se basan en las reglas de subtipado (contravarianza) y los tipos paramétricos se acotan mediante subtipado, un ejemplo de lenguaje particular que se adapta a esta variante de MOON es TRELLIS/OWL. En la variante *2)* las reglas de la herencia se basan en las reglas de subtipado (contravarianza) y los tipos paramétricos se acotan mediante cláusulas *tal que*[4], un ejemplo de lenguaje que se adapta a esta variante de MOON es THETA. En la variante *3)* las reglas de la herencia se basan en reglas de conformidad (covarianza) y los tipos paramétricos se acotan mediante conformidad, un ejemplo de lenguaje que se adapta a esta variante de MOON es EIFFEL.

Un lenguaje particular que no incluya genericidad se puede mirar fácilmente desde el prisma de MOON, igualmente lenguajes como C++ que incluyen una forma muy rudimentaria de genericidad en la que no hay acotación. En estos casos lo único que sucede es que se simplifica aún más el modelo. Lo mismo podemos decir de lenguajes sin anotaciones de tipo.

La definición de MOON tiene como limitación que no permite fácilmente el tránsito a poder modelar lenguajes con nivel meta avanzado, como SMALLTALK. Pero este es un aspecto que podremos atacar en un futuro tomando como base lo que se ha desarrollado en FAMIX para ese caso.

4.3 Estructura de los objetos

La definición de la estructura de los objetos depende en última instancia de la estructura de las clases y de la definición de los tipos completamente instanciados que dichas clases implementan. Conociendo la estructura de los objetos de un tipo dado, se podrá analizar las consecuencias de la aplicación de las refactorizaciones para evaluar si son *respetuosas con los objetos*, lo que conduciría a saber si es necesario hacer, lo que en bases de datos se denomina, migración de poblaciones.

Los objetos forman parte de la extensión de algún tipo no paramétrico o de algún tipo paramétrico completamente instanciado. El modelo de tipos asumido define que la estructura de un tipo está dada por un record cuyos campos se corresponden con las signaturas de las propiedades del tipo, atributos y métodos. Por eso es inmediato pensar en la estructura de un objeto como el valor de un record. El valor de un record contiene los valores de cada campo definido en el record.

Digamos entonces que en el record que representa el objeto tenemos los valores de los campos que se corresponden con atributos, el valor correspondiente al identificador del tipo del objeto, un campo cuyo valor será el identificador único del objeto al que se le suele denominar oid y un campo que indica dónde encontrar las implementaciones de los métodos. Este campo se conoce como referencia a la tabla de métodos.

Los valores de los atributos de un objeto en MOON pueden ser: una referencia a otro objeto o constantes manifiestas (regla 40, Apéndice A). Como referencia a otro objeto bastaría tener el valor de su oid. Esto hace homogéneo el valor de los atributos en la estructura de los objetos persistentes y de los objetos en ejecución. Por claridad asumimos que los valores son homogéneos, es decir, las referencias a objetos son oids.

La estructura de los objetos MOON, en presencia de herencia de sus clases, es una estructura basada en concatenación [18]. Esta es la forma más frecuente, según hemos detectado, en las implementaciones de los lenguajes revisados.

Se define que los objetos persistentes tienen la misma estructura que los objetos en ejecución excepto la referencia a la tabla de métodos pues esta se actualiza cuando el objeto se carga

[4] cláusulas *tal que*, del término en inglés *where clauses*.

desde el soporte externo durante una ejecución, utilizando el valor del identificador del tipo del objeto.

El análisis de las consecuencias para los objetos de la aplicación de las refactorizaciones, se hace a partir de la estructura de los objetos persistentes. Puede parecer que para este análisis también haría falta definir la estructura de la tabla de métodos pues, evidentemente, cuando se aplica una refactorización, esta puede cambiar. Una vez que se realice la refactorización, hay que volver a "compilar" (o realizar una acción con un efecto similar) las clases transformadas. De esta forma la tabla de métodos se regenera. Esto no se refleja en la estructura del objeto pues solamente tiene una referencia a la tabla. Para lo que sí tiene consecuencias es para los mensajes que el objeto es capaz de responder. Pero determinar esto se corresponde con el análisis de las consecuencias de la refactorización para los clientes. A partir de la estructura del objeto lo que se analiza es si los objetos persistentes, que están dados fundamentalmente por los valores de sus atributos, siguen siendo válidos en una sesión posterior a la refactorización.

5 MOON y refactorización

Se ha definido una arquitectura y un proceso para integrar el modelo en herramientas de refactorización. En este apartado presentamos ambos aspectos.

5.1 Arquitectura

En el marco de MOON definimos una arquitectura de refactorizadores en la que entran clases a un repositorio pasando por un analizador que extrae los elementos relevantes (definidos por MOON) y almacena el resultado en forma de un objeto persistente cuya estructura responde a un modelo de clases que describe las partes más relevantes del árbol sintáctico que se genera del análisis de una clase MOON. A este formato le llamamos formato de clases preanalizadas. La entrada de una clase al repositorio implica la actualización de las relaciones entre las clases preanalizadas. Las relaciones entre las clases preanalizadas se han definido formalmente en el modelo de MOON como un Grafo de Dependencias que tiene en cuenta las relaciones de herencia y cliente entre las clases, así como las relaciones de herencia y cliente con sustitución (para los parámetros formales) en presencia de clases genéricas. El modelo formal está definido mediante un grafo dirigido etiquetado.

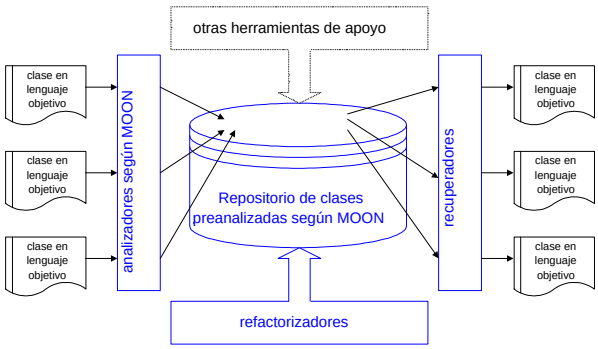

Figura 2: *Arquitectura de refactorizadores basada en* MOON.

Se establecen reglas de formación del Grafo de Dependencias a partir de la estructura de las clases. Las reglas de formación consideran la presencia de relaciones con sustitución para expandir las dependencias. También se definen consultas que permiten extraer información del repositorio a partir de la estructura de las clases preanalizadas y del Grafo de Dependencias (e.g. clases de las que depende una clase, clases que dependen de una clase, origen de una propiedad, etc.).

A partir de estas definiciones, en la arquitectura propuesta, la estructura de la nueva clase que llega al repositorio determina la forma en que se actualiza el Grafo de Dependencias.

Las herramientas de refactorización actúan

sobre las clases preanalizadas que se encuentran en el repositorio. De los objetos persistentes que representan las clases preanalizadas del repositorio se debe poder extraer el texto equivalente en el lenguaje de partida. De esta forma se obtienen los resultados de la refactorización. La Figura 2 muestra un esquema de dicha arquitectura.

Con este enfoque se disminuye la complejidad de la definición de una herramienta de refactorización pues ésta se centra en elementos que responden a una estructura que contiene sólo lo que tiene que considerar la herramienta para refactorizar, apartándole de lidiar con los detalles propios del análisis de un lenguaje particular.

Como ya hemos dicho, es muy importante analizar si la refactorización obtiene elementos correctos, cómo se afectan las clases que dependen de las clases modificadas y cómo se afectan los objetos persistentes que hayan sido creados previamente a la refactorización. De la misma forma que la definición, el análisis se ve simplificado por el modelo. El Grafo de Dependencias nos permite analizar las clases que deben ser transformadas y la forma en que se afectan las clases que dependen de las transformadas. La corrección de los nuevos elementos y las consecuencias para los objetos se analiza desde la estructura y las reglas definidas en MOON.

5.2 Proceso

El proceso que guía la inserción de MOON en el ámbito de las refactorizaciones tiene dos partes fundamentales: definición y análisis, e implementación, que se resumen a continuación.

1. Definición y Análisis
   a) Poner en correspondencia el lenguaje objetivo con MOON
   b) Definir en MOON la refactorización
   c) Analizar sus consecuencias
4. Implementación
   a) Construir un analizador del lenguaje objetivo que obtiene como resultado para cada clase un objeto cuya estructura viene dada por el modelo de clases que se deriva del árbol sintáctico si se analizaran textos MOON. Estos objetos están enlazados por un grafo que representa el Grafo de Dependencias del Repositorio.
   b) Construir un recuperador de textos en el lenguaje objetivo a partir dicho Grafo. Este paso y el anterior se condicionan.
   c) Implementar la refactorización definida, actuando sobre el Grafo de Dependencias del Repositorio.

En este escenario, la intención de agregar una nueva refactorización para el mismo lenguaje objetivo implica trabajar en 1.b), 1.c) y 2.c), mientras que aplicar la misma refactorización para elementos de un nuevo lenguaje implica trabajar en 1.a), 2.a) y 2.b).

6 Un caso de estudio

En esta sección se presenta un caso de estudio del modelo en dos partes. En primer lugar un caso de estudio del modelo como abstracción de lenguajes, y en segundo lugar un caso de estudio del modelo como base para la definición y análisis de refactorizaciones. Para esto se han elegido el lenguaje Eiffel y una refactorización definida por nosotros y presentada en [19] y [20].

La selección de Eiffel viene dada por sus recursos en cuanto a genericidad (incluyendo acotaciones) y de la herencia que permiten explotar bien el modelo. Además, la claridad de la sintaxis, el tratamiento uniforme de las entidades, la ausencia de punteros y el manejo automático de la memoria, facilitan la definición e implementación de los analizadores necesarios.

6.1 EIFFEL visto desde MOON

El lenguaje seleccionado (EIFFEL) se corresponde con la variante *3)* de MOON, teniendo en cuenta además algunos aspectos particulares como los tipos ancla y expandidos, su conjunto de instrucciones y sus formas de expresiones.

Todas las construcciones se pueden reducir, a la hora de realizar un análisis estático de las clases EIFFEL, de la misma forma que se presentaron las simplificaciones de MOON en la Sección 4. Las instrucciones de creación, asignación, y envío de mensajes y las formas de expresiones de envío de mensajes y constantes manifiestas son la base de las demás.

Evidentemente, hay que tener en cuenta, tal y como se analizó, en la presentación de MOON (al no incluir las instrucciones **if** y **while**) que ciertas instrucciones exigen un tipo determinado de expresiones, e.g. expresiones de tipo *Boolean*, como en la instrucción **if**, o de tipo *Integer*, como en la parte **variant** de una instrucción **loop** de EIFFEL. La solución que se presentó en las simplificaciones de MOON: asumir que existe una asociación con una constante manifiesta del tipo requerido, es válida para este caso.

Los aspectos relativos al conjunto de instrucciones de EIFFEL se vuelven a reducir a la instrucción de creación, la instrucción de asignación (el intento de asignación ?= en EIFFEL se analiza desde MOON como una asignación), y la instrucción de envío de mensajes. Los envíos de mensajes en cascada se simplifican. En cuanto a las formas de expresiones que se corresponden con expresiones con operadores binarios y unarios en EIFFEL, se tratarán como expresiones de envío de mensajes, tal y como se expuso en Sección 4. Esto se puede hacer directamente en EIFFEL, sin tener que cambiar ninguna consideración, porque todos los tipos básicos se corresponden con una clase de la biblioteca KERNEL.

6.2 Desarrollo de una refactorización: parameterize

En este apartado se presenta una refactorización que hemos definido y denominado parametrización. El objetivo de esta refactorización es obtener clases genéricas a partir de clases que no lo son y transformar el software basado en estas clases para utilizar las nuevas clases genéricas y sus instanciaciones.

Dada una orden de parametrización

C.**parameterize**(e **as** T)

donde C es la clase objetivo, la clase que se quiere hacer genérica; e es la entidad guía, la entidad cuyo tipo específico va a pasar a estar dado por el nuevo parámetro formal T; la ejecución de la refactorización pasa por:

1. Determinar el grafo que forman las clases que participarán en la obtención de las entidades genérico-dependientes a partir de la entidad guía y las relaciones entre dichas clases. A este grafo se le denomina universo de trabajo y se denota G^I.
   Las entidades genérico-dependientes son aquellas que deben modificar su tipo a genérico producto del cambio del tipo de e. El grafo del universo de trabajo se especifica muy fácilmente según el modelo del Grafo de Dependencias establecido a partir de MOON. Este grafo estará determinado por los condicionantes de C (ancestros y proveedores), y por sus descendientes.
2. Determinar las entidades genérico-dependientes directas e indirectas.
   Las entidades genérico-dependientes pueden ser directas o indirectas, en alusión a la forma en que cambia su tipo. El tipo de la entidad e cambia para ser el nuevo parámetro genérico formal, algunas entidades (las directas) cambiarán de la misma forma, mientras que otras (las indirectas) cambian para que su tipo pase a ser un tipo genérico instanciado con el nuevo parámetro (si su tipo era A ahora sería A[T].

La clara especificación de las reglas del sistema de tipos de MOON y la abstracción en sus construcciones simplifican en gran medida la definición de cómo obtener estas entidades y el análisis de la corrección de su selección. Hay que garantizar corrección de tipos una vez ejecutada la transformación. Las reglas fundamentales a tener en cuenta son las que guían la corrección de las asociaciones entre entidades y expresiones. Las reglas 34, 35, 42 y 44 son las que determinan, y la ausencia de envíos de mensajes en cascada, así como la restricción sobre las expresiones que pueden ser argumentos reales en un envío de mensaje, clarifican la definición formal y el análisis de la formación de los conjuntos de entidades genérico-dependientes.

3. Determinar si es posible proseguir con la parametrización y con qué clases.
   Con esto se obtiene un subgrafo del grafo G^I. Este subgrafo pasa a constituir el grafo de las clases candidatas a participar en la parametrización y se denota G^C.
   Nuevamente las reglas de tipos y las asociaciones determinan. Se hace necesario un análisis exhaustivo de los cambios que pueden ocurrir y de las reglas que se deben cumplir para determinar qué circunstancias conducirían a obtener elementos incorrectos, y prohibirlas. Por ejemplo, no se puede obtener una entidad cuyo tipo es un parámetro formal asociada con una constante manifiesta.
4. Si del paso anterior se determina que no se puede proseguir o si no se desea: Terminar
5. Eliminar de G^C las clases a las que no es necesario propagar la operación (las clases que no contienen ninguna de las entidades genérico-dependientes determinadas).
   El grafo resultante se denomina grafo de clases finales y se denota G^F. consultar si se desea proseguir y con qué clases, rearmando G^F cada vez que se elimine una clase interactivamente.
   Las entidades que tienen que cambiar su tipo determinan las clases que tienen que ser modificadas y las relaciones de dependencia de éstas determinan si la eliminación de una clase conduce a la eliminación de otras.
6. Analizar la genericidad acotada.
   Se analiza la necesidad de acotar los nuevos parámetros formales. Se determina cuáles deben ser las restricciones. La forma de expresarlas varía de acuerdo a la variante de acotación modelada en MOON. La que mejores resultados da es la variante *2)* (acotación mediante cláusulas *tal que*).
7. Comenzar la parametrización a partir de las clases ancestros en el grafo de herencia que subyace en G^F hasta las descendientes (considerando el análisis de genericidad acotada).
   Las estructuras que pueden ser afectadas son las dadas por las siguientes reglas:
   regla 3, porque de no haber sido genérica la clase, habría que modificar la construcción que da paso a los parámetros formales.
   regla 7, porque habría que añadir el nuevo parámetro formal, y en el caso de la variante *2)*, en esta regla se da paso a la construcción para acotar el nuevo parámetro formal si es necesario (regla 49W).
   regla 8, porque en las variantes *1)* y *3)* es aquí donde hay que dar paso a la construcción para acotar el nuevo parámetro formal si es necesario (regla 48S).
   reglas 49W y 50W, en el caso de la variante *2)*.
   regla 48S, en las variantes *1)* y *3)*.
   regla 10, porque hay que verificar si el padre, tiene como clase determinante una clase que ha sido modificada por la parametrización. En ese caso, el tipo del padre debe modificarse añadiéndole el parámetro real correspondiente a la sustitución.
   reglas 19 y 22, porque las entidades genérico-dependientes declaradas a partir de estas reglas deben cambiar a variable

su tipo.
Los cambios en las reglas 48S, 10, 19 y 22 condicionan que hay que reformar las dependencias expresadas en el Grafo de Dependencias del Repositorio. Las dependencias cambian en las siguientes formas: una relación puede pasar a ser de sustitución, puede cambiar la sustitución de una relación, puede aparecer una nueva relación producto de la inclusión de acotación para los nuevos parámetros formales en el caso de las variantes *1)* y *3)*

8. Reajustar las clases que no están en G^F y que dependen directamente (hijos o clientes directos) de las clases parametrizadas para que pasen a depender de las nuevas clases genéricas instanciadas.

Esto último indica que la refactorización **parameterize** no respeta a los clientes (directos) de las clases transformadas. El otro aspecto significativo a analizar sobre las consecuencias de la refactorización es su impacto en los objetos persistentes. El modelo de objetos basado en concatenación, la estructura de los objetos persistentes con el identificador de su tipo (ausencia de referencia a los métodos), y la gestión del repositorio sobre la identificación de los tipos hace que la transformación no tenga consecuencias para los objetos persistentes.

Esta refactorización se implementó en el caso de estudio de EIFFEL, siguiendo la arquitectura diseñada (Figura 2). Los únicos aspectos que hubo que tener en cuenta por encima del modelo de clases preanalizadas según MOON fueron: no transformar el tipo de una entidad declarada como tipo ancla pues la declaración como ancla es más significativa que tener las dos entidades declaradas del mismo tipo; y tener en cuenta la presencia de tipos expandidos en el análisis de genericidad acotada, pues si el resultado de la acotación es un tipo expandido, debido a las reglas de conformidad de tipos, se limitan las posteriores instanciaciones.

Se construyeron las siguientes herramientas:

- un preanalizador de clases EIFFEL: genera objetos persistentes representando las clases preanalizadas según el modelo de MOON
- un gestor de repositorio que mantiene el Grafo de Dependencias y permite extraer información de los objetos que representan las clases preanalizadas.
- un refactorizador que implementa la operación **parameterize**

Con este caso de estudio hemos comprobado la viabilidad del modelo y de la arquitectura diseñada para la definición y análisis de refactorizaciones. Estamos trabajando actualmente en una versión de JAVA con genericidad para migrar automáticamente bibliotecas de clases no genéricas. El modelo y arquitecturas MOON está mostrándose muy efectivo en la reducción del esfuerzo de implementar la refactorización para otro lenguaje, lo que era uno de los objetivos fundamentales de la propuesta.

7 Conclusiones

Hemos presentado un lenguaje modelo, MOON, para la definición y análisis de refactorizaciones, así como una arquitectura para su integración en herramientas de refactorización. El propósito fundamental es disminuir el esfuerzo de definición y desarrollo de herramientas para refactorizar elementos de diferentes lenguajes. Se ha realizado un caso de estudio de la viabilidad de este propósito dando resultados muy positivos.

Nuestro trabajo parece estar en la misma línea que el iniciado por Tichelaar et al. [8] (Sección 3). A diferencia de este último, que define un modelo de intercambio de información mediante un esquema entidad-relación y su especificación se basa en CDIF, nosotros definimos un lenguaje modelo dado por una gramática de atributos y las reglas de tipos del lenguaje, entre otros aspectos, que condu-

ce a una representación de las clases analizadas según MOON como objetos persistentes cuya estructura viene dada por el modelo de clases que se deriva del árbol sintáctico. Derivar de esta estructura un modelo de intercambio resulta inmediato. Por otra parte, creemos que esta forma de abordar el problema nos establece un punto de partida inmejorable para abordar la refactorización independiente del lenguaje. En esta sentido, nuestras actividades apuntan ya a la generación automática de herramientas de refactorización partiendo de las gramáticas y del sistema de tipos de MOON y del lenguaje de programación objetivo.

Referencias

[1] W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[2] W.G. Griswold and D. Notkin. Architectural trade offs for a meaning preserving program restructuring tool. *IEEE Transactions on Software Engineering*, 21(4):275–287, 1995.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[4] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. *Proceedings of the 6th International Symposium on the Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes*, 23(6):99–110, Nov. 1998.

[5] D. Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[6] K. Beck. *Extreme Programming explained: embrace change.* Addison-Wesley, 1999.

[7] Y. Crespo. *Incremento del potencial de reutilización del software mediante refactorización.* PhD thesis, Dpto. de Informática, Universidad de Valladolid, 2000.

[8] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000.* IEEE, 2000.

[9] J.J. Rodríguez, Y. Crespo, and J.M. Marqués. Transformación de jerarquías de herencia múltiple en jerarquías de herencia sencilla. Technical Report TR-GIRO-03-98, Dpto. de Informática, Universidad de Valladolid, 1998.

[10] M. O Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *Proceedings of the International Conference on Software Maintenance ICSM'99*, 1999.

[11] M. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2), Feb. 1997.

[12] J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[13] P. Wegner. Dimensions of object-based language design. In *Proceedings of OOPSLA'87*, 1987.

[14] K.J. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *Conference Proceedings of OOPSLA'88*, pages 323–334, Sept. 1988.

[15] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–523, Dec 1985.

[16] K.B. Bruce and P. Wegner. An algebraic model of subtypes in object-oriented languages. In *ACM SIGPLAN Notices. Object-Oriented Programming Workshop.*, Oct. 1986.

[17] B. Liskov and J.M. Wing. Family values: A semantic notion of subtyping. Technical Report MIT-LCS-TR-562, MIT, Dec. 1992.

[18] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, Sept. 1996.

[19] Y. Crespo, J.J. Rodríguez, F.J. García, and J.M. Marqués. Obtención automática de clases genéricas a través de una operación de parametrización. In *Actas JISBD'99*, pages 343–354, Nov. 1999.

[20] Y. Crespo, J.J. Rodríguez, and J.M. Marqués. Obtaining generic classes automatically through a parameterization operator. A focus on constrained genericity. In *Proceedings of TOOLS 31th, Asia'99.* IEEE CS Press, Sept.

A Sintaxis concreta de MOON

```
1  MODULE              ≜ CLASS_DEF
2  CLASS_DEF           ≜ [deferred] class CLASS_NAME HEADER SIGNATURES CLASS_BODY end
3  CLASS_NAME          ≜ CLASS_ID [FORMAL_PARAMETERS]
4  HEADER              ≜ INHERITANCE_LIST
5  SIGNATURES          ≜ signatures SIG_LIST
6  CLASS_BODY          ≜ body { METHODS_IMPL ';' ...}
7  FORMAL_PARAMETERS   ≜ '[' { FORMAL_PAR ',' ...}+ ']' [ BOUND_W ]          variante w
8  FORMAL_PAR          ≜ FORMAL_GEN_ID [ BOUND_S ]                           variante s
9  INHERITANCE_LIST    ≜  { INHERITANCE_CLAUSE ';' ...}
10 INHERITANCE_CLAUSE  ≜ inherit CLASS_TYPE OPLUS                            hereda
11 OPLUS               ≜ { MODIFIER ';' ...}
12 MODIFIER            ≜ rename PROP_ID as PROP_ID | redefine PROP_ID | makedeferred PROP_ID |
                         makeeffective PROP_ID | select PROP_ID
13 SIG_LIST            ≜ ATTRIBUTE_DECS METHOD_DECS
14 ATTRIBUTE_DECS      ≜ attributes { ATT_DEC ';' ...}
15 METHOD_DECS         ≜ methods { METH_DEC ';' ...}
16 ATT_DEC             ≜ VAR_DEC
17 METH_DEC            ≜ ROUTINE_DEC | FUNCTION_DEC
18 ROUTINE_DEC         ≜ WITHOUT_RESULT
19 FUNCTION_DEC        ≜ WITHOUT_RESULT ':' TYPE                             es cliente
20 WITHOUT_RESULT      ≜ [deferred] METHOD_ID [FORMAL_ARGUMENTS]
21 FORMAL_ARGUMENTS    ≜ '(' { VAR_DEC ',' ...}+ ')'
22 VAR_DEC             ≜ VAR_ID ':' TYPE                                     es cliente
23 TYPE                ≜ FORMAL_GEN_ID | CLASS_TYPE
24 CLASS_TYPE          ≜ CLASS_ID [REAL_PARAMETERS]                          ¿de qué clases?
25 REAL_PARAMETERS     ≜ '[' { TYPE ',' ...}+ ']'                            ¿de qué clases?
26 METHOD_IMPL         ≜ NONE_DEFERRED_R | NONE_DEFERRED_F
27 NONE_DEFERRED_R     ≜ MSIG [LOCAL_DECS] METHOD_BODY
28 NONE_DEFERRED_F     ≜ MSIG ':' TYPE [LOCAL_DECS] METHOD_BODY
29 LOCAL_DECS          ≜ { VAR_DEC ';' ...}+
30 METHOD_BODY         ≜ do INSTR end
31 MSIG                ≜ METHOD_ID [FORMAL_ARGUMENTS]
32 INSTR               ≜ COMPOUND_INSTR | CREATION_INSTR |
                         ASSIGNMENT_INSTR | CALL_INSTR          no hay instrucciones de control
33 COMPOUND_INSTR      ≜ {INSTR ';' ...}
34 CREATION_INSTR      ≜ create VAR_ID
35 ASSIGNMENT_INSTR    ≜ VAR_ID ':=' EXPR
36 CALL_INSTR          ≜ CALL_INSTR_LONG1 | CALL_INSTR_LONG2        no hay llamadas en cascada
37 CALL_INSTR_LONG1    ≜ METHOD_ID [REAL_ARGUMENTS]
38 CALL_INSTR_LONG2    ≜ CALL_EXPR_LONG1'.'CALL_INSTR_LONG1
39 EXPR                ≜ MANIFEST_CONSTANT | CALL_EXPR   se reducen las formas de expresiones
40 MANIFEST_CONSTANT   ≜ nil | REAL_CONSTANT | INTEGER_CONSTANT | BOOLEAN_CONSTANT |
                         CHAR_CONSTANT | STRING_CONSTANT
41 CALL_EXPR           ≜ CALL_EXPR_LONG1 | CALL_EXPR_LONG2          no hay llamadas en cascada
42 CALL_EXPR_LONG1     ≜ ENTITY [REAL_ARGUMENTS]
43 CALL_EXPR_LONG2     ≜ ENTITY'.'CALL_EXPR_LONG1
44 REAL_ARGUMENTS      ≜ '(' {EXPR_ATOM ',' ...}+ ')'
45 EXPR_ATOM           ≜ MANIFEST_CONSTANT | CALL_EXPR_LONG1
```

continúa en la página siguiente ...

... continuación de la página anterior

46	ENTITY	$\triangleq$ VAR_ID | **result** | **self**
47	PROP_ID	$\triangleq$ VAR_ID | METHOD_ID

Los símbolos BOUND_S y BOUND_W que aparecen en las reglas anteriores, dependen de qué variante de acotación de parámetros genéricos formales esté en consideración. Las reglas correspondientes a estos símbolos se definen a continuación de acuerdo a la variante de lenguaje a la que dará lugar.

48S	BOUND_S	$\triangleq$ '->'CLASS_TYPE	*es cliente*
48W	BOUND_S	$\triangleq \epsilon$	
49S	BOUND_W	$\triangleq \epsilon$	
49W	BOUND_W	$\triangleq$ **where** { WHERE_CLAUSE ',' ... }+	
50W	WHERE_CLAUSE	$\triangleq$ FORMAL_GEN_ID **has** SIG_LIST	

Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante subtipado o conformidad, las reglas que se utilizan son las marcadas con S (reglas 48S y 49S).

Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante cláusulas *tal que*, las reglas que se utilizan son las marcadas con W (reglas 48W, 49W y 50W).