Genericidad inversa

Yania Crespo* José Manuel Marqués† Juan José Rodríguez‡

17 de febrero de 1998

Resumen

Este documento constituye una propuesta de trabajo en el marco de la reutilización en lenguajes y ambientes orientados a objeto. Esta propuesta consiste en definir una operación inversa de genericidad. Esto es, disponer de un mecanismo con el que dada una clase en un entorno orientado al objeto se pueda obtener una clase genérica, producto de aplicar un operador de parametrización. En este documento se describe de forma general la propuesta de sintaxis y semántica para dicho operador, así como algunas consideraciones a ser tenidas en cuenta para introducirlo en un entorno de programación concreto.

Palabras claves: reutilización, orientación al objeto, lenguajes de programación, genericidad.

1 Introducción

A lo largo del desarrollo de la Programación Orientada a Objetos, como se muestra en la figura 1, aunque con diferentes matices, ha quedado claro cómo construir objetos según la descripción de una clase, cómo obtener clases concretas a partir de clases abstractas, definiendo la implementación de los métodos (funciones, etc.) que quedaron diferidos según las especificidades de los distintos lenguajes y entornos de POO [10, 12]. Así mismo está visto cómo obtener clases instancias a partir de clases genéricas, proporcionando los parámetros de tipo necesarios [8]; u obtener subclases a partir de clases, a través de la herencia [4, 13]. También se han realizado trabajos para obtener superclases (abstractas o no) para clases ya existentes generalizando [11], refactorizando [8] o aplicando algoritmos de transformación de jerarquías de clases [6] e incluso, para obtener clases a partir de un conjunto de objetos representativos [9].

En el esquema anterior las clases aparecen clasificadas siguiendo la propuesta del modelo ACTI, formulado por Edwards en [7], de forma que estas son abstractas (A) o concretas (C) y plantillas (clases genéricas, T de *template*) o instancias (I), estableciendo 4 categorías a partir de las combinaciones AT, AI, CT, CI. Uno de los problemas que presenta esta clasificación es su rigidez, pues puede aludirse que en diferentes lenguajes, o en última instancia modelos objeto, se pudieran establecer grados intermedios entre cada una de las clasificaciones. Así por ejemplo, una clase abstracta en Eiffel [10] o C++ [12] puede tener todos, varios o un método diferido (función virtual pura en los términos de C++). En Java [2], las interfaces pudieran clasificarse exactamente en la categoria de Abstracta mientras

*Universidad de La Habana, Cuba. e-mail: yania@infor.uva.es
†Universidad de Valladolid, Spain. e-mail: jmmc@infor.uva.es
‡Universidad de Valladolid, Spain. e-mail: juanjo@infor.uva.es

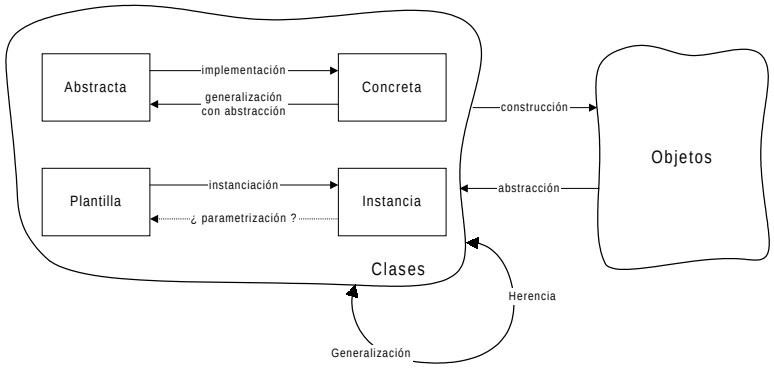

Figura 1: Transformaciones

que las clases están en el mismo caso de una clase Eiffel o C++ y por otra parte ninguna clase en Java aparecería en la categoría de Plantilla. No obstante la variedad, siempre pueden establecerse criterios para enmarcar estos casos en las 4 categorías descritas. En general esta clasificación nos vale para dar la idea de las transformaciones entre elementos de software orientado a objeto (clases y objetos, básicamente) que han sido desarrolladas hasta el momento.

La propuesta de este trabajo consiste en cerrar dicho esquema presentando un operador de parametrización, que lleva desde clases instancia a clases plantilla. Es decir, que dada una clase generalmente no paramétrica (no genérica) en un lenguaje orientado al objeto se pueda obtener una clase genérica producto de la aplicación de dicho operador. Esto es, obtener su plantilla y convertirla a ella misma en instancia de la plantilla resultante. Esto no quita el caso particular en que una clase ya genérica quiera parametrizarse aún más.

Como puede analizarse, la existencia del operador *parameterize* no sólo amplía las posibilidades de reutilización y facilita la localización de nuevas clases en una jerarquía de clases existente, sino que también se convierte, para los casos en que es aplicable, en una operación alternativa de las transformaciones de jerarquías de herencia. Con la ventaja de que no tiene implicaciones para los objetos ya creados ni para el código existente. Algunas clases susceptibles de ser blanco de parametrizaciones para reutilizarse pueden ser estructuras de almacenamiento, clases que encapsulen algoritmos, etc.

Por otra parte, muchas veces como recomendaciones de programación se dice que al programar una clase genérica, primero se implemente un prototipo de la clase para un tipo básico en particular que permita probar extensivamente la clase y, cuando se tenga esta por correcta, re-escribirla como genérica. Contar con este operador mantiene dicha recomendación y da la posibilidad de que, con una indicación al entorno, en este se realicen las modificaciones necesarias automáticamente. Incluso, algunas metodologías de Análisis y Diseño Orientado a Objetos como BON [13] incluyen un proceso de revisión de las clases para ser reutilizadas en otros dominios, como parte de este proceso entra hacer las clases más generales, lo cual incluye convertir algunas en genéricas. Teniendo implementado el operador *parameterize*, una herramienta CASE puede realizar esta transformación de forma automática.

Si, finalmente, en un lenguaje como Java se adopta una forma de genericidad de las muchas propuestas [1, 3], contar con un operador como este será muy útil para actualizar las clases de las bibliotecas existentes.

2 Descripción general del operador

El operador *parameterize* se concibe para un nivel de entorno de programación, incluso pudiera considerarse como a nivel de lenguaje de programación para algún lenguaje en específico. En la descripción que se realizará en este trabajo se hablará del operador como un operador del entorno, que es el caso más general. Esto se debe fundamentalmente a que en un lenguaje orientado a objetos puro cualquier construcción lingüística debe formar parte de una clase, describiendo una propiedad o como parte del código de una función (un método). Un operador como este resultaría en una construcción extra lingüística. Por otra parte, en nuestra opinión, el entorno y el lenguaje deben estar estrechamente vinculados para que el entorno potencie la capacidad de reutilización del lenguaje.

2.1 Sintaxis

El operador que nos ocupa pudiera indicarse en forma de comando o quizás como opción "gráfica" del entorno. Como comando su sintaxis general se describe de la siguiente forma:

<C_id>.**parameterize**(<E_id> **as** <fgen > [,<E_id> **as** <fgen>]*)

Donde,

C_id es el identificador de una clase que se encuentra en el repositorio de clases del entorno de programación. La forma de identificar las clases puede variar según el entorno, generalmente bastaría con el nombre de la clase y, en caso de ambigüedad, la especificación del *cluster* al que pertenece la clase [10, 13]. Obsérvese que no hay restricciones para el tipo de clase que es C_id, esta puede ser lo mismo una clase abstracta, concreta, una instancia o incluso una plantilla que se quiere parametrizar aún más.

E_id es el identificador de una entidad de la interfaz de la clase C_id, es decir, un atributo, un parámetro o el resultado de una función. Para identificar un atributo de la clase bastaría con su nombre mientras que identificar un parámetro o el resultado de una función estará condicionado por el nombre de esta. Para cada caso particular de lenguaje (o entorno) se verá de qué forma es mejor denotar el resultado de una función. En general un atributo atrib puede verse también como (_,atrib) pero para simplificar la notación se escribirá sólo como atrib.

fgen corresponde al identificador del parámetro genérico formal que se utilizará en la nueva plantilla como tipo de la entidad E_id.

El hecho de que en el operador se indique el identificador de una entidad y no el tipo que se quiera parametrizar es evidente que está dado por la posibilidad de existencia en la misma clase de varias entidades no vinculadas pero que tengan el mismo tipo. Un ejemplo claro puede ser el caso en que se tenga una estructura de almacenamiento, pongamos por caso, un tipo de lista de enteros. Esta lista tiene elementos de tipo entero y también tiene un atributo de tipo entero que representa el total de elementos en la lista. Naturalmente, no se quiere que al parametrizar dicha lista para almacenar otros tipos de elementos, por ejemplo una lista de estudiantes, se obtenga que el total de elementos en la lista es de tipo estudiante.

2.2 Semántica

En esta sección se describirá lo relacionado con la semántica del operador *parameterize* de forma general para no caer en especificidades ligadas a un lenguaje o entorno de programación particular.

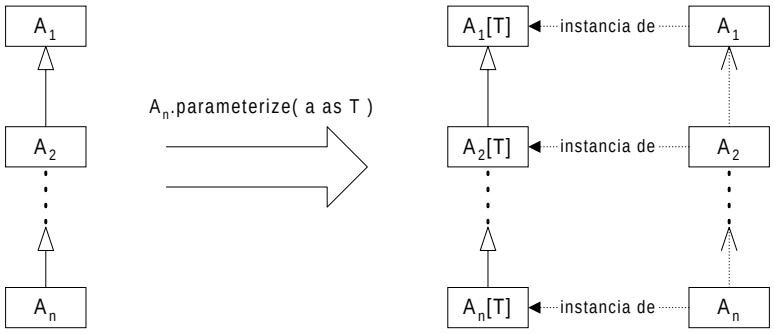

Figura 2: Resultado de aplicar el operador en A_n cuando a se introdujo en A_1

Antes de pasar a describir la semántica del operador se dará un poco informalmente la definición de una relación que se ha estado nombrando hasta ahora como "entidad vinculada con".

Para introducir el operador *parameterize* en un lenguaje de programación debe darse una definición específica de dicho lenguaje para la relación de vinculación entre entidades, que define qué entidades están relacionadas de forma tal que un cambio en el tipo de una puede implicar un cambio en el tipo de la otra. Dicha definición deberá garantizar que la relación de vinculación es una relación de equivalencia. Sobre la base de dicha relación se podrán implementar los algoritmos necesarios con los que se determine qué entidades deben cambiar su tipo junto con las entidades que se especifican en la declaración del operador.

2.2.1 Restricciones

En general, dada la operación de parametrización de una clase, se propone establecer las siguientes restricciones:

1. Si alguna de las entidades en la lista de parametrización es ya genérica en C_id, la expresión es incorrecta. Este operador transforma el tipo para ser genérico de aquellas entidades que no lo son.
2. Si existen dos entidades objetivo de la lista de parametrización que están vinculadas en C_id, entonces deben estar denotando el mismo parámetro genérico formal, en cuyo caso sería una expresión redundante pero no incorrecta.

Otro aspecto importante es definir qué efectos debe considerar quien introduzca el operador *parameterize* en un entorno específico. ¿Cómo se afectan las clases ancestros y herederas de la clase a parametrizar? ¿Cómo se afectan las clases de las que es cliente? ¿Qué problemas hay que dejar resueltos?

2.2.2 Efectos en la jerarquía de herencia

En caso de que algunas entidades de aquellas que deben cambiar su tipo por uno de los genéricos formales especificados en la declaración del operador, sea una propiedad heredada, se propone determinar (si es necesario, por los caminos de herencia de la clase) quién introduce cada entidad y lanzar el proceso de parametrización a partir ahí bajando con la propagación por los caminos de herencia hasta alcanzar las hojas (ver Figura 2).

2.2.3 Propagación a las clases servidoras

Cuando se parametriza una clase y ya se han determinado todas las entidades que cambiarán su tipo (aquellas que se encuentren en la clase de equivalencia, por la relación "vinculada con", de cada entidad especificada en el operador) para los genéricos formales, debe analizarse la propagación de los cambios a las clases servidoras.

Si existe alguna entidad, en el cuerpo de la clase parametrizada, a la cual se le solicita un servicio pasando como parámetro una entidad de las que han modificado su tipo como genérico, para garantizar la correctitud de tipos de la nueva clase genérica hay que revisar si es necesario propagar la parametrización a la clase servidora (en caso de que esta no lo sea ya), a partir de dicho parámetro y luego, consecuentemente, actualizar el tipo de la entidad en cuestión.

2.2.4 Genericidad Acotada

En [5] se define la cuantificación acotada. Esta dota a la genericidad (polimorfismo universal paramétrico) de la restricción necesaria para garantizar que un módulo que depende de un tipo genérico T no puede ser instanciado con cualquier tipo, sino sólo con aquellos que cumplan las condiciones impuestas. Esto es muy útil no solamente para expresar mejor las carácterísticas del tipo definido, sino también, en un lenguaje estática y fuertemente tipado, para garantizar la seguridad de tipos. En un entorno orientado a objetos, si a través de una entidad se invocan determinados servicios, debe garantizarse que la clase con la que se declara dicha entidad los tiene.

Al introducir el operador de parametrización deben analizarse las carácterísticas del lenguaje en cuanto a tipos y a genericidad acotada y actuar en consecuencia. El caso peor, cuando la restricción resulte en toda la funcionalidad de la clase que era originalmente el tipo de la entidad antes de la parametrización, esta aún sigue teniendo sentido a pesar de que parezca que se ha obtenido una clase equivalente a la que se tenía. En [5] Cardelli y Wegner explican la importancia y beneficios de la cuantificación universal acotada.

En el caso de que el entorno particular en el que se define el nuevo operador no soporte este recurso, deberán tomarse las decisiones adecuadas (aumentar las restricciones en la semántica del operador, etc.).

3 Conclusiones

En este trabajo, hasta la versión actual, está definido de forma general el operador, los efectos que hay que considerar al introducirlo en un entorno particular y aquellas cuestiones que traerían problemas de no tenerse en cuenta. La implementación de estas tienen algunas limitaciones que pueden intentar resolverse. En este momento se trabaja en dar una definición de los detalles de introducir este operador en el entorno de programación de Eiffel. Evidentemente, sería interesante realizar otros trabajos analizando diferentes lenguajes, pero más aún, en nuestra opinión sería el estudio de esta operación como alternativa, cuando es aplicable, de las reorganizaciones de jerarquías de herencia.

4 Agradecimientos

Este trabajo ha sido realizado con el apoyo del proyecto CICYT TIC97-0593-C05-05. Yania Crespo es estudiante de doctorado gracias al Instituto de Cooperación Iberoamericana (ICI). Los autores agradecen al resto de los integrantes del grupo GIRO por su apoyo y por los frutos que han salido de nuestras eternas discusiones.

Referencias

[1] D. Aranha and P. Borba. Parameterized packages and java. In L.E. Buzato and C.M. Fisher, editors, *Anais II Simposio Brasileiro de Linguagens de Programação.* Sociedade Brasileira de Computação, 1997.

[2] K. Arnold and J. Gosling. *The Java Programming Language.* Java Series. Sun Micosystems, 1996.

[3] J. Bank, B. Liskov, and A. Myers. Parameterized types and java. Technical Report MIT LCS TM-553, Massachusets Institute of Technology, May 1996.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. Uml. Technical report, Rational Software Corporation, 1996.

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

[6] E. Casais. Automatic reorganization of object-oriented hierarchies: a case study. *Object-Oriented Systems*, 1:95–115, 1994.

[7] S.H. Edwards. *A formal model of software subsystems.* PhD thesis, Ohio State University, 1995.

[8] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1(2):22–35, 1988.

[9] Lieberherr. K.J., P. Bergstein, and N. Silva-Lepe. From objects to classes: algorithms for optimal object-oriented design. *Journal of Software Engineering*, July 1991.

[10] B. Meyer. *Eiffel: the language.* Prentice-Hall International, 1989.

[11] C.H. Pedersen. Extending ordinary inheritance schemes to include generalization. In N. Meyrowitz, editor, *Conference Proceedings of OOPSLA'89*, pages 407–418. Special issue of SIGPLAN Notices: 24(10), ACM Press, October 1989.

[12] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[13] K. Waldem and J.M. Nerson. *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems.* Object-Oriented Series. Prentice Hall, 1995.