Transformación de clases para obtener clases genéricas. Implicaciones en las jerarquías de herencia y de agregación/composición

Yania Crespo
Dpto. de CC. de la Computación
Universidad de La Habana
Cuba

José Manuel Marqués
Dpto. de Informática
Universidad de Valladolid
España

Juan José Rodríguez
Dpto. de Informática
Universidad de Valladolid
España

{yania, jmmc, juanjo}@infor.uva.es

TR-GIRO-04-98

Resumen

La genericidad puede describirse como la capacidad de parametrizar módulos. Estos módulos en el caso de los lenguajes orientados al objeto se concretan en clases. Aunque la genericidad no es un recurso intrínseco de la Programación Orientada a Objetos complementa adecuadamente este paradigma, permitiendo la definición de plantillas de clases también nombradas clases genéricas. A partir de las clases genéricas se pueden obtener otras clases (instancias de dichas plantillas). En este trabajo se presenta un operador de parametrización automática. La esencia del operador radica en que de su aplicación a una clase en un lenguaje orientado a objetos se pueda obtener una clase genérica. Se presenta una descripción general y una definición más específica del operador **parameterize** como resultado del estudio de su introducción en un entorno de programación Eiffel. La selección de Eiffel viene dada por la forma en que se pueden combinar todas las propiedades que tiene la genericidad y otros recursos que aparecen en el lenguaje como los tipos anchor y la herencia. No obstante, las ideas generales pueden aplicarse a cualquier otro lenguaje orientado a objetos que incluya genericidad. Se argumenta la utilidad del operador en reutilización y como alternativa en las transformaciones de jerarquías de herencia.

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

1 Introducción

La Programación Orientada a Objetos (POO) ofrece un marco adecuado para la reutilización de software. El principio abierto-cerrado que promueve la POO está soportado en primer lugar por la definición de la clase como un módulo: cerrado para utilizarse; y por

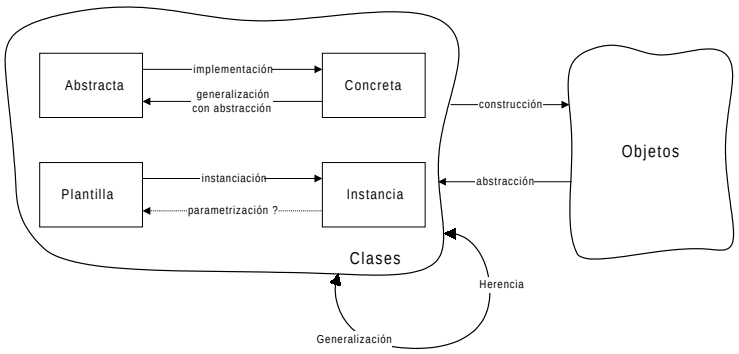

Figura 1: Transformaciones entre elementos de software Orientado a Objetos.

la posibilidad de extender y especializar estos módulos a través de la herencia: abierto para adaptar y mejorar. Estos son los pilares de la reutilización basada en mecanismos de herencia.

Las clases se organizan en jerarquías de herencia y están sujetas a cambios con el fin de mejorar su capacidad para ser reutilizadas. Esto usualmente conduce a procesos de reorganización de jerarquías [7, 9] algunas veces bastante complejos, que producen clases artificiales y/o ocasionan la necesidad de adaptar a poblaciones de objetos persistentes.

La genericidad brinda otra forma de reutilización debido a que hace posible tener módulos paramétricos. Esto permite la definición de plantillas de clases, también llamadas clases genéricas, que constituyen generadores de otras clases, instancias de dichas plantillas. El cliente que obtiene una clase utilizando este recurso solamente debe suministrar algunos parámetros de tipo que se especifican en la definición de la clase genérica. Dichos parámetros se denominan parámetros genéricos formales en la clase genérica y parámetros genéricos reales en la clase instancia.

Aunque la genericidad no es un recurso intrínseco de la Programación Orientada a Objetos complementa adecuadamente este paradigma. La herencia y la genericidad se convierten en la concreción en un Lenguaje Orientado a Objetos del polimorfismo universal definido por Cardelli y Wegner [5]: polimorfismo de inclusión en el caso de la herencia y polimorfismo paramétrico en el caso de la genericidad.

La genericidad sólo tiene sentido en lenguajes estáticamente tipados, como se explica en [17], y propone una forma importante para expresar mejor ciertas dependencias entre los tipos de diferentes elementos de la estructura de una clase. Muchos lenguajes Orientados a Objetos estáticamente tipados soportan genericidad. Java [2] no, pero se han presentado un número creciente de trabajos para proponer extensiones al lenguaje que permitan la parametrización de clases [3] e incluso de paquetes (*packages*) [1].

A lo largo del desarrollo del paradigma de Programación Orientada a Objetos, ha quedado claro (aunque con algunas particularidades) cómo construir objetos a partir de descripciones de clases, cómo obtener clases concretas a partir de clases abstractas (definiendo la implementación de los métodos diferidos, de acuerdo a las características específicas de los diferentes lenguajes de POO [16, 21]). También se ha visto cómo obtener clases instan-

cias a partir de clases genéricas (suministrando los parámetros de tipo necesarios [12]); o subclases a partir de otras clases (utilizando la herencia [4, 22]). Se han desarrollado trabajos para obtener superclases (abstractas o no) a partir de clases existentes: generalizando [19], refactorizando [13, 18] o aplicando algoritmos de transformación sobre jerarquías de clases [8]; e incluso para obtener clases a partir de un conjunto representativo de objetos [14]. La figura 1 ilustra estas transformaciones.

Los cuatro tipos diferentes de clases que se muestran en la figura 1 se han tomado del modelo ACTI [12]. En este modelo, los componentes software, clases en este caso, se clasifican en: abstracta (A) o concreta (C) y plantilla (T, del término inglés *template*) o instancia (I), estableciendo 4 categorías a partir de sus combinaciones permitidas: AT, AI, CT, CI. La rigidez de esta clasificación es discutible porque se pueden encontrar en diferentes lenguajes grados intermedios entre cada posición A–C, T–I. Por ejemplo, una clase abstracta en Eiffel [16] o en C++ [21] puede tener uno, todos o alguno de sus métodos diferidos. En Java una interfaz puede clasificarse exactamente como Abstracta mientras que una clase se encuentra en el mismo caso que las clases Eiffel o C++. Sin embargo siempre se pueden establecer algunos criterios para colocar estos casos en las cuatro categorías descritas. A pesar de esto, la clasificación del modelo ACTI permite expresar esquemáticamente, como se hace en la figura 1, las principales transformaciones que se han descrito hasta el momento entre elementos básicos del software orientado a objetos (clases y objetos) ya sea mediante construcciones de lenguajes o herramientas de Ingeniería de Software que realizan meta-operaciones.

Este trabajo es una propuesta que cierra dicho esquema con la introducción de un operador de parametrización, como se muestra en la figura con una línea de puntos. De la aplicación de este operador se puede obtener una clase genérica a partir de otra, generalmente no paramétrica (no genérica). En otras palabras, se puede obtener su plantilla y convertir a la clase misma en una instancia de la plantilla resultante. En este planteamiento no se rechaza el caso de una clase genérica que quiera ser parametrizada más aun. Esta operación se enmarca entre aquellas que son meta-operaciones a ser realizadas por una herramienta de Ingeniería de Software como mismo lo son las refactorizaciones y los algoritmos de reorganización de jerarquías.

Como consecuencia de que algunas características particulares para la genericidad varían según el lenguaje, se dará una descripción detallada del operador enfocado al caso de su introducción en una herramienta para Eiffel. A pesar de esto, las ideas generales que se presentan en este trabajo podrían aplicarse a otros lenguajes orientados a objetos con genericidad. La forma en que está presente la genericidad como construcción del lenguaje en Eiffel es muy satisfactoria y, por otra parte, su combinación con otras propiedades del lenguaje como la declaración por asociación a través de tipos anchor (**like**) y la herencia, permitirá ajustar el proceso de parametrización en una forma más elegante.

El resto del trabajo está organizado como sigue. En la sección 2 se motiva a través de un ejemplo la posibilidad de contar con este operador. La sección 3 presenta una definición general de la sintaxis y semántica del operador independiente del entorno de programación concreto para el que se utilice. Más adelante, en la sección 4, se describe de forma más detallada las características de la parametrización en una herramienta dirigida a Eiffel. Por último la sección 5 expone algunas conclusiones y enumera ejemplos para subrayar la

```
class GRAPHIC_WINDOW
 –...
 background : COLOR;
 –...
 change_background(new_background : COLOR);
 paint; – Pinta todos los elementos de la ventana
        – En particular pide al background que
        – se pinte (background.paint)
 –...
end - GRAPHIC_WINDOW class
```

Tabla 1: Fragmento de la forma *flat* de la clase GRAPHIC_WINDOW.

utilidad de contar con un operador como este.

2 Presentando el operador a través de un ejemplo

Se asume la existencia de cierta clase, GRAPHIC_WINDOW en el ejemplo, en un repositorio de clases de algún entorno de programación como sería un entorno para asistir a la construcción de programas Eiffel. En el ejemplo 1 se presenta un fragmento de la forma *flat* [17, pp.541-543] de dicha clase (ver figura 2). En este caso la clase GRAPHIC_WINDOW es una clase que probablemente se pre-compiló y almacenó en el repositorio del entorno de programación.

En el caso específico de Eiffel, lamentablemente, la estructura pre-compilada de una clase y la forma en que se almacena es diferente dependiendo del entorno específico de programación. Véase por ejemplo la propuesta de *Eiffel Units* (*EU*) [10] para Eiffel/UH, los *Eiffel Gen* de ISE Eiffel, etc. Todas estas estructuras pre-compiladas, aunque con sus diferencias, deben cubrir las mismas necesidades, de forma que permitan, por medio de ciertas herramientas, recuperar información para ayudar a localizar clases con propósitos de reutilización. Por ejemplo estas estructuras deben ser la base para extraer, con las herramientas adecuadas, las formas *short* y *flat* de una clase, sus ancestros, así como permitir operar con ella para que se pueda interpretar, verificar y depurar.

De vuelta a la situación inicial, supóngase que se desea localizar una clase de ventana gráfica con el ánimo de reutilizarla. Se localiza la clase GRAPHIC_WINDOW y a través de su *EU* se consulta su forma *flat*. A partir de la información que se reporta en dicha forma *flat* se llega a que esta clase se asemeja a lo que se necesita pero la ventana gráfica que se quiere construir requiere que su background pueda contener un *bitmap*.

¿Cuáles son las opciones posibles con las que se cuenta?

a) Tratar de reutilizar el código de la clase GRAPHIC_WINDOW a través de herencia. Si se define la nueva ventana, de nombre MY_WINDOW, como una clase descendiente de GRAPHIC_WINDOW será necesario redefinir el atributo background como una entidad de tipo BITMAP. Para poder hacer esto, el lenguaje de programación debe soportar la

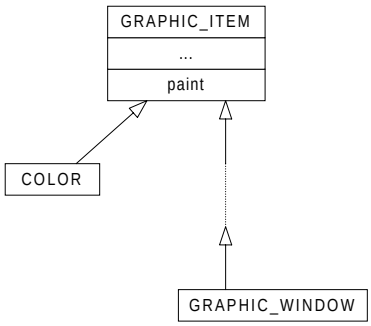

Figura 2: Jerarquía que representa el ejemplo 1.

redefinición de tipos para los recursos cuando se hereda. En C++ no puede hacerse pero sí en Eiffel. Así y todo, Eiffel es un lenguaje con redefinición covariante y por tanto se requiere que BITMAP herede de COLOR (en realidad la regla de validez dice que conforme con COLOR). Por el contrario, en el caso de que el lenguaje permita redefinición contravariante COLOR debe heredar de (o conformar con) BITMAP.

Esta decisión conduce a una jerarquía de herencia forzada que no tiene sentido ya que BITMAP no es un heredero natural de COLOR (y por supuesto tampoco lo contrario). En realidad, un BITMAP está formado por múltiples puntos de colores y lo único que tiene en común con un COLOR es su capacidad de ser pintado.

b) Se tiene disponible una herramienta, o se puede realizar manualmente, que modifica la jerarquía de herencia. Existen varias reorganizaciones posibles de la jerarquía. Una de ellas se muestra en la figura 3. En esta reorganización se introduce una clase GRAPHIC_WINDOW_G cuyo atributo background tiene un tipo más general, por ejemplo GRAPHIC_ITEM. De acuerdo a esto la jerarquía queda organizada de forma que COLOR y BITMAP hereden de GRAPHIC_ITEM, GRAPHIC_WINDOW hereda de GRAPHIC_WINDOW_G y redefine background como COLOR. Finalmente, MY_WINDOW puede convertirse en un heredero de GRAPHIC_WINDOW_G redefiniendo background como BITMAP. En estas reorganizaciones debe tenerse en cuenta cómo repercuten en las clases clientes y la necesidad de considerar de qué forma proceder respecto a la evolución de los posibles objetos persistentes creados a partir de las versiones anteriores de estas clases [6].

c) Si no es posible aplicar ninguna de las opciones anteriores se construirá una nueva ventana y se deja entonces de lado la posibilidad de reutilizar.

Sin embargo, supóngase que se cuenta en el entorno de trabajo con una herramienta que pueda ejecutar una meta-operación de nombre operador *parameterize* y se aplica de la siguiente forma (esto representa una línea de comandos pero podría obtenerse gráficamente en la herramienta):

GRAPHIC_WINDOW.**parameterize**(background **as** T)

Como consecuencia de aplicar esta operación se obtienen dos resultados importantes:

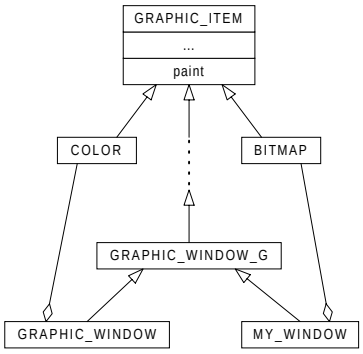

Figura 3: Resultado de una de las posibles transformaciones de la jerarquía.

```
class GRAPHIC_WINDOW_GEN[T]
 –...
 background : T;
 –...
 change_background(new_background : T);
 paint; – Pinta todos los elementos de la ventana
        – En particular se pide al background que
        – se pinte (background.paint)
 –...
end - GRAPHIC_WINDOW_GEN[T] class
```

Tabla 2: Forma *flat* (fragmento) de la clase resultado de aplicar el operador.

- una clase genérica GRAPHIC_WINDOW_GEN[T] donde T es el parámetro genérico formal que identifica el tipo del atributo background (ver el tipo estático del ejemplo 2).

- una clase
  GRAPHIC_WINDOW = GRAPHIC_WINDOW_GEN[COLOR].

  Entonces se puede obtener la nueva ventana como
  MY_WINDOW = GRAPHIC_WINDOW_GEN[BITMAP]

De la exposición anterior se puede analizar que la existencia de un operador como *parameterize* no solamente incrementa las posibilidades de reutilización y facilita la ubicación de nuevas clases en una jerarquía de clases existentes, sino que también se convierte en una operación alternativa de reorganización de jerarquías de clases en los casos en que pueda aplicar. La ventaja de esta operación reside en sus leves implicaciones para el código existente pero más aun, si la implementación de este operador se lleva a cabo en una herramienta con una total integración con el compilador, se puede asegurar que no habrá ninguna implicación para aquellos objetos persistentes que hayan sido creados con anterioridad. En la sección 5 se discuten otras ventajas de disponer de una realización del operador **parameterize**.

3 Descripción general del operador

El operador **parameterize** se concibe para un nivel meta de entorno de desarrollo o una herramienta de ingeniería de software que apoye estas actividades. La descripción que se hace en esta presentación es breve para el caso general y más extensa para el caso particular de su introducción en una herramienta para Eiffel. Esto es así porque en definitiva **parameterize** es una refactorización y como se expone en [18], las refactorizaciones siempre son de muchas formas dependientes del lenguaje objetivo. En este sentido el proyecto ESPRIT FAMOOS[1] trabaja actualmente en herramientas de re-ingeniería (como pueden ser las refactorizaciones) independientes del lenguaje que se basan en un modelo que nombran como FAMOOS Information Exchange Model [20]. Por el momento puede considerarse una buena aproximación que el entorno-herramienta y el compilador del lenguaje estén estrechamente integrados de forma que el entorno fortalezca las capacidades de reutilización del lenguaje y permita, por ejemplo, refactorizaciones y reorganizaciones.

3.1 Sintaxis

Lo que aquí se describe es la sintaxis de la forma textual del operador de parametrización como un comando. Teniendo esta base se pueden definir otras formas visuales de indicar la misma operación en una herramienta. Como un comando textual la 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 del repositorio de clases del entorno. Las formas de identificar clases pueden variar de acuerdo al entorno particular, generalmente el nombre de la clase es suficiente y en caso de ambigüedad puede ser necesaria alguna aclaración sobre el contexto, lugar de almacenamiento, etc. (por ejemplo en Eiffel podría ser el *cluster* al que pertenece la clase [16, 22]). Nótese que no hay restricciones para el tipo de clase que sea C_id, puede ser tanto una clase abstracta, concreta, una instancia o incluso una plantilla que se quiera parametrizar más.

E_id es el identificador de una entidad de la interfaz de la clase C_id: un atributo, un parámetro de un método o el resultado de una función. Para identificar un atributo generalmente es suficiente con su nombre, mientras que el identificador de un parámetro o el resultado de una función estará condicionado por el identificador del método (función). Por ejemplo en Eiffel puede ser (F_id, param) o (F_id, Result) respectivamente. Result es una palabra reservada en Eiffel para identificar la variable cuyo valor retorna la función una vez finalizada. Para cada herramienta específica se vería cual es la mejor forma de denotar el resultado de una función. Si se quiere homogeneizar, un atributo atrib puede representarse como (_, atrib) pero con el objetivo de simplificar la notación se escribirá solamente como atrib. Un ejemplo de utilizar diferentes tipos de entidades de la interfaz es (el ejemplo está basado en el caso que se presentó en el ejemplo 1):

[1] ESPRIT Project 21975

GRAPHIC_WINDOW.**parameterize**((change_background,new_background) **as** T)

más adelante se verá que esta forma resulta equivalente a tener:

GRAPHIC_WINDOW.**parameterize**(background **as** T)

puesto que background y (change_background, new_background) son entidades relacionadas de la clase GRAPHIC_WINDOW (ver las secciones 3.2 y 4.1).

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

En la sintaxis del operador se hace referencia al identificador de una entidad y no al tipo que se quiere pasar a genérico. La razón de esto está dada por la posibilidad de la existencia en la misma clase de varias entidades no relacionadas que tengan el mismo tipo. Por ejemplo, si se tiene una lista de enteros como estructura de almacenamiento esta lista tendrá elementos de tipo entero y un contador de tipo entero que representa el total de elementos almacenados en la lista. Si la declaración del operador considerara tipos en lugar de entidades y se quiere parametrizar la lista con el objetivo de almacenar otros tipos de elementos la indicación sería algo como LIST.**parameterize**(INTEGER **as** T). En este caso se estaría pidiendo que también el contador de elementos en la lista fuera de tipo genérico formal T.

3.2 Semántica

En esta sección se establece una descripción semántica del operador **parameterize** de forma general.

Los efectos operacionales de **parameterize** estarán condicionados por las dependencias entre entidades y expresiones en la clase objetivo. La siguiente definición expresa esta dependencia como una relación.

Definición 1 *(Entidad Relacionada): Sean x e y entidades de una clase. Si existe una dependencia entre ellas tal que un cambio en el tipo de x implique la necesidad de cambiar el tipo de y o vice versa, se dice que x está relacionada con y y vice versa o en general que x e y son entidades relacionadas. Esta relación se denotará como* $x \sim y$.

Para introducir el operador **parameterize** en una herramienta relacionada con un lenguaje específico, se debe definir una especificación de la relación "entidad relacionada" de acuerdo con dicho lenguaje objetivo. Sobre la base de esta relación se podrán implementar los algoritmos necesarios para determinar qué entidades deben cambiar su tipo junto con aquellas que se declararon como argumentos del operador.

3.2.1 Restricciones

Dado una orden de parametrizar una clase,

c_id.**parameterize**(e_id$_1$ **as** T$_1$,...,e_id$_n$ **as** T_n), $n \geq 1$

se establecen las restricciones generales siguientes:

1. En caso de que la clase actual c_id sea ya una plantilla, los identificadores de los nuevos parámetros genéricos formales que aparecen en la declaración tienen que ser distintos a cualquiera de los identificadores de los actuales parámetros genéricos formales de c_id.

2. Si una entidad que se declara en la lista de argumentos del operador ya tiene como tipo en la clase actual c_id, un parámetro genérico formal ($\exists$ e_id$_i$, $1 \leq i \leq n$ t.q. e_id$_i$ tiene como tipo en c_id un parámetro genérico formal), la orden dada mediante el operador es incorrecta. No se permite transformar una plantilla para que cambie el tipo de una entidad que ya tiene como tipo un parámetro genérico formal.

3. Si dos o más entidades declaradas en la lista de argumentos del operador son entidades relacionadas en la clase c_id ($\exists$ e_id$_i$ y e_id$_j$, $1 \leq i, j \leq n, i \neq j$, t.q. e_id$_i \sim$ e_id$_j$) entonces en la orden dada mediante el operador ambas deben estar asociadas al mismo identificador para el nuevo parámetro genérico formal. En este caso puede que la orden sea redundante pero no incorrecta. En el caso contrario se tiene que la orden dada mediante el operador es incorrecta.

4. Si dos o más entidades declaradas en la lista de argumentos del operador se asocian al mismo identificador del nuevo parámetro genérico formal y *no* son entidades relacionadas en la clase c_id ($\exists$ e_id$_i$ y e_id$_j$, $1 \leq i, j \leq n, i \neq j$ t.q. T$_i$=T$_j$ y e_id$_i \not\sim$ e_id$_j$) se llama demanda explícita de relación y requiere que ambas entidades estén declaradas en la clase actual c_id con el mismo tipo estático, en otro caso se tiene que la orden dada mediante el operador es incorrecta.

Otro punto importante es definir qué efectos producirá el operador **parameterize** cuando se ejecuta una vez que se ha introducido en un ambiente específico. ¿Cómo cambian ancestros y herederos de la clase a ser parametrizada? ¿Cómo cambian las clases clientes y servidoras? ¿Qué problemas aparecen?

3.2.2 Efectos en la jerarquía de herencia

Se propone que cuando se tengan entidades que deben cambiar su tipo para un parámetro genérico formal y estas sean propiedades que la clase objetivo hereda de sus ancestros. Las acciones a seguir pasan por:

- Determinar a través de los caminos de herencia de la clase cuál(es) es(son) la(s) clase(s) origen de la entidad y comenzar el proceso de parametrización a partir de ahí hacia las clases herederas.

- Propagar los cambios de la parametrización por todas las clases que se encuentran en los caminos de herencia hasta que se alcancen las clases minimales de la jerarquía (ver figura 4 y sección 4.2).

El segundo de los puntos anteriores podría cambiarse por:

- Propagar los cambios de la parametrización por todas las clases que se encuentran en los caminos de herencia que conducen a la clase objetivo hasta alcanzarla.

```
class GRAPHIC_WINDOW
 inherit WINDOW
   redefine paint
 –...
 paint; – Pinta todos los elementos de la ventana
 –...
end - GRAPHIC_WINDOW class
```

Tabla 3: Fragmento de la forma *short* de la clase GRAPHIC_WINDOW

Pero en este caso sería necesario introducir como tercer punto:

- Buscar los descendientes directos de aquellas clases que se hayan parametrizado y cambiar su cláusula de herencia de forma que cada clase herede de una instancia de la nueva plantilla cuyo(s) parámetro(s) genérico(s) actual(es) se corresponda(n) con el tipo que tenían las entidades objetivo antes de ser parametrizadas.

De acuerdo con esto, se prefirió definir un proceso homogéneo como el que se describe en los dos puntos iniciales.

En la forma *flat* de una clase no aparece información de herencia. Los recursos de la clase, heredados e intrínsecos, se unifican para ofrecer sin distinciones la información de todas las propiedades y operaciones con las que se cuenta en los objetos de dicha clase. Por el contrario la forma *short* de una clase muestra las relaciones de herencia de la clase y sus recursos intrínsecos. El ejemplo 1 se considerará de forma que la clase GRAPHIC_WINDOW herede de una clase WINDOW que es la que introduce el atributo background. Esto puede verse en el ejemplo 3 que muestra la forma *short* de la clase GRAPHIC_WINDOW.

Cuando se aplica la operación GRAPHIC_WINDOW.**parameterize**(background **as** T) según los puntos que se describieron anteriormente se obtendría:

- una clase genérica WINDOW_GEN[T], donde T es el parámetro genérico formal correspondiente al atributo background.
- una clase WINDOW = WINDOW_GEN[COLOR].
- una clase genérica GRAPHIC_WINDOW_GEN[T] que hereda de WINDOW_GEN[T], donde T sigue siendo el parámetro genérico formal correspondiente al atributo background (idem sección 2)
- una clase GRAPHIC_WINDOW = GRAPHIC_WINDOW_GEN[COLOR] (idem sección 2).

Si fuera el caso de que GRAPHIC_WINDOW tuviera descendencia, el proceso de parametrización se propagaría hasta alcanzar las clases minimales como se decidió anteriormente.

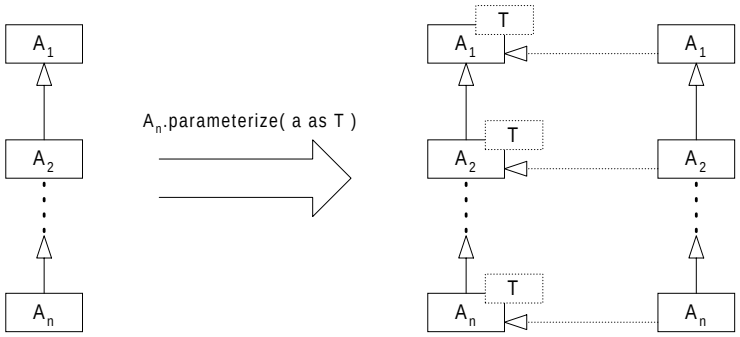

Figura 4: Resultado de aplicar el operador **parameterize** para la entidad a de la clase A_n cuando a se introdujo realmente en A_1

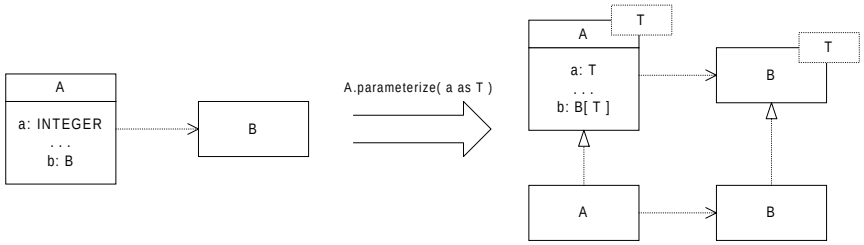

Figura 5: Resultado de la aplicación del operador **parameterize** para la entidad a de la clase A cuando B es una clase servidora que tiene elementos de su interfaz relacionados en A con entidades parametrizadas.

3.2.3 Propagación a las clases servidoras

Cuando se parametriza una clase A y se determinan las entidades que deben cambiar su tipo para convertirse en genérico, se debe analizar la propagación de los cambios de parametrización a las clases servidoras de A. Las entidades que cambian su tipo se obtienen mediante la relación "entidad relacionada".

Si en la clase objetivo existe alguna entidad en estas condiciones, es decir, que es una entidad representando una clase servidora debe verificarse si alguna entidad de su interfaz está relacionada con aquellas que deben cambiar su tipo para ser genérico. De acuerdo con esto, y con el objetivo de garantizar la correctitud de tipos de la nueva clase genérica, el proceso de parametrización debe propagarse a la clase servidora especificando como lista de argumentos aquellas entidades de su interfaz que resultaron estar relacionadas con la parametrización. Luego consecuentemente se procede a actualizar el tipo de la clase servidora con respecto a la entidad de la clase objetivo que originó la propagación (ver figura 5 y sección 4.3).

3.2.4 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 características 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 será capaz de responder ante solicitudes de dichos servicios.

Cuando se introduce el operador de parametrización en una herramienta para un lenguaje en particular, se debe tener en cuenta las características en cuanto a tipos y a genericidad acotada del lenguaje y actuar en consecuencia. Por ejemplo, en el caso de Eiffel, la forma en que se introduce como recurso del lenguaje la genericidad restringida (o acotada) se basa en restringir el tipo del parámetro genérico formal a ser descendiente de (-conformar con- es más general) una determinada clase. Esta clase garantiza la funcionalidad mínima requerida para las entidades de dicho tipo que aparecen en la clase genérica. Es decir, la acotación significa: todo tipo con el que se instancie la clase debe ser heredero de (debe conformar con) *la clase restricción*. Cuando se parametriza una clase debe determinarse qué servicios se invocan a partir de las entidades que forman la clase de equivalencia, por la relación "entidad relacionada", de cada entidad especificada en el operador. Entonces se puede detectar la mínima clase que incluye todos esos servicios. De esta manera al concluir la parametrización cada parámetro genérico formal que lo requiera estará restringido a dicha clase.

El caso peor, cuando la clase restricción resulta la misma que la clase original antes de la parametrización, 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.

Si el lenguaje específico con el que se corresponde la herramienta en la que se define el nuevo operador no soporta este recurso, deberán tomarse las decisiones adecuadas (aumentar las restricciones en la semántica del operador, considerar otras transformaciones que simulen la presencia y control de las restricciones, ignorar la necesidad de restringir, etc.).

4 Definición del operador: el caso de Eiffel

En esta sección se presentarán de forma más detallada los aspectos que se discutieron previamente acerca de la introducción del operador de parametrización en una herramienta para un lenguaje específico. Eiffel es el lenguaje que se ha seleccionado.

4.1 Entidades relacionadas

En primer lugar se debe obtener una definición detallada y particular de la relación "entidad relacionada" que aproveche y se base en las especificidades de las construcciones del lenguaje Eiffel.

Definición 2 *(Entidad relacionada en el caso de Eiffel):*
Sean a y b entidades de la misma clase Eiffel. Se dice que a está relacionada con b, y se denota como $a \sim b$*, si se cumple cualquiera de las siguientes situaciones:*

1. *a y b son la misma entidad.*

2. *a y b tienen el mismo tipo estático y están en la lista de argumentos de la declaración del operador* **parameterize** *señalando el mismo identificador de un nuevo parámetro genérico formal (demanda explícita de relación* A.**parameterize**(...,a **as** T,...,b **as** T,...)

3. *a tiene tipo estático* **like** b *o vice versa (declaración por asociación con tipos anchor [16, 17])*

4. *a y b tienen el mismo tipo estático y participan en los siguientes tipos de expresiones (tanto en el cuerpo de los métodos como en aserciones):*

   - a.f(...,b,...)
   - b.f(...,a,...)

   Esta es una notación general en forma de punto ("dot form") pero abarca también el caso en que f es un operador (+,=,...). Note que el concepto de entidad abarca también los resultados de funciones.

5. *a y b tienen el mismo tipo estático y existe una instrucción* a:=expression *como parte del cuerpo de algún método de la clase donde b está presente en* expression*, idem* b:=expression *donde a está en* expression*. Esto cubre el caso particular donde* a:=b *(*b:=a*).*

6. *existe c tal que* $a \sim c$ *y* $c \sim b$

La relación "entidad relacionada" es una relación de equivalencia: reflexiva, $a \sim a$ por 1), simétrica por definición de 2), 3), 4) y 5), y transitiva por 6).

Los puntos 4) y 5) determinan una definición excesivamente pesimista de la relación. Esta forma evita un proceso recursivo complejo para analizar en profundidad cualquier llamado a función en el cual la entidad objetivo está involucrada.

Una suposición menos pesimista podría consistir en considerar a.f(b) (a.+(b)) y a:=b como una relación fuerte entre a y b. Por otra parte a.f(...,b,...), a:=expression podrían considerarse como signos de una relación débil entre a y b. Si a y b están débilmente relacionados se puede declarar explícitamente lo contrario cuando se utiliza el operador de parametrización.

La aproximación menos pesimista podría ser cambiar los puntos 4) y 5) por: a es un *alias* de b [11].

4.2 Efectos en la jerarquía de herencia

De acuerdo a lo explicado en 3.2.2, este apartado expone las operaciones de propagación de la parametrización en una jerarquía de herencia. Se explica de forma detallada para el caso de Eiffel.

Dada la orden A.**parameterize**(e$_1$ **as** T$_1$, ..., e$_m$ **as** T$_m$)

e_i $(1 \leq i \leq m)$ son las entidades objetivo de la parametrización, T_i el nombre de los parámetros genéricos formales para e_i en la clase genérica resultante y PT_i es el tipo que e_i tiene en A antes de comenzar el proceso de parametrización.

Primeramente, para cada entidad e_i que aparece como argumento de la orden **parameterize**, se obtiene $\overline{e_i}$ como
$\overline{e_i} = \{e$, entidad en el "aplastamiento"[2] de A$|e \sim e_i\}$. $\overline{e_i}$ representa la clase de equivalencia de e_i respecto a la relación $\sim$ (sección 4.1).

Se chequean las restricciones del operador (sección 3.2.1). Si estas se satisfacen entonces el proceso continúa.

Con el objetivo de simplificar la exposición, se determinará una forma normalizada del operador de manera que queden eliminadas las redundancias que podrían presentarse. Si existen e_i y e_j que son argumentos de **parameterize**, donde $e_i \sim e_j$ y $T_i = T_j$, entonces e$_j$ **as** T$_j$ se elimina de la lista de argumentos del operador.

Sea A.**parameterize**(e$_1$ **as** T$_1$, ..., e$_n$ **as** T$_n$), $(n \leq m)$ la forma normalizada.

CE denota el conjunto de aquellas entidades cuyo tipo debe cambiar para ser genérico. Las entidades en
$CE = \cup_{i=1}^{n} \overline{e_i}$ se denominarán entidades conflictivas.

Dada una entidad e, la operación $e.Clase$ determina las clases que primero introducen a e en la jerarquía. Note que pueden ser varias las clases que introduzcan e ya que esta entidad pudo haberse unificado en una sola propiedad en algún punto de herencia múltiple.

Entonces se determinan las clases origen del proceso de parametrización. Es decir, se determinan aquellas clases a partir de las cuales se desencadenará el proceso. Sea OC el conjunto de las clases origen. $OC = \cup_{e \in CE} e.Clase - \{C | C$ hereda de D, C y $D \in \cup_{e \in EC} e.Clase\}$.

El subgrafo S (del grafo que forman las clases en la jerarquía de herencia), es el subgrafo afectado al propagar la parametrización de clases a través de la herencia. S contiene las clases derivadas de las clases origen. En este punto se deben completar las clases de equivalencia para cada e_i y el conjunto CE mediante el análisis de aquellas clases en S que no están situadas en los caminos de herencia que parten desde las clases que pertenecen a OC hasta A.

S se organiza por generaciones en forma descendente según [15]. Los niveles que se obtienen son $G_1, \ldots, G_k$. Las clases en G_1 son clases maximales con respecto a S $(G_1 = OC)$ y las clases en G_k son las clases minimales. El proceso de parametrización comienza en las clases maximales.

A partir de este punto los pasos a seguir se describen en el siguiente algoritmo.

[2] *flattening*

1) Desde $g = 1$ hasta k hacer // *recorrer las generaciones del subgrafo afectado // desde las clases maximales hasta las minimales.*

2) Para cada clase C en la generación g hacer

   2.1) Si C es maximal en S // *se analizan las clases origen y por tanto // no se tienen en cuenta los cambios realizados en los padres*

      a) Cada entidad en C que pertenezca a alguna clase de equivalencia $\overline{e_i}$ cambiará su tipo por un parámetro genérico formal T_i, a menos que su tipo estático PT_i sea un tipo anchor (**like**). En este caso se mantiene el tipo como un tipo anchor.

      b) Analizar las entidades servidoras (sección 4.3)

      c) Analizar genericidad acotada (sección 4.4)

   Se obtienen las clases genéricas C_GEN[...,T_i,...], con los parámetros genéricos formales acotados si fuera necesario, de acuerdo con el resultado del análisis en b). C se transforma en la clase instancia C = C_GEN[...,PT_i,...]

   2.2) Sino // *se deben considerar los cambios realizados en los padres*

      a) La cláusula de herencia de la nueva clase C_GEN cambia con respecto a la de C con el objetivo de considerar que las modificaciones de aquellos padres de C que han pasado a ser clases genéricas.

      b) Determinar las entidades de $\overline{e_i}$ que se introducen o se redefinen en C.
      Si es el caso de que alguna entidad conflictiva se redefina en C para cambiar su tipo, dicha redefinición se reflejará acotando el parámetro genérico formal.
      Si es el caso de que hay entidades conflictivas que se introducen en C, cambiar su tipo para ser T_i, a menos que su tipo estático PT_i sea un tipo anchor.

      c) Analizar las clases servidoras (sección 4.3)

      d) Analizar genericidad acotada (sección 4.4). Se deben tener en cuenta los parámetros genéricos formales que se definieron anteriormente en los ancestros de C. Si como resultado de este análisis se obtiene una restricción menos fuerte que la restricción propuesta en la superclase (de haberla): no hacer nada; en caso contrario: en la declaración de C_GEN se especificará la nueva restricción.

   Se obtienen la clase genérica C_GEN[...,T_i,...], con los parámetros genéricos formales acotados si fuera necesario, de acuerdo con el resultado del análisis en b). Los T_i incluyen los parámetros genéricos formales que ya se encuentran en las superclases de C y los nuevos que aparecen en C (si es el caso). C se transforma en la clase instancia C = C_GEN[...,PT_i,...]

4.3 Propagación a las clases servidoras

En el algoritmo anterior se indica la propagación a las clases servidoras. En ese punto debe chequearse para cada entidad a, cuyo tipo estático se corresponde con un parámetro genérico formal resultado de la parametrización, si existe alguna entidad cuyo tipo estático

no es genérico (sea el tipo PT) y está presente en una expresión como e.f(..., a, ...) o a:=e.f(...). La entidad e es una entidad servidora. Se tienen que considerar en consecuencia los siguientes aspectos:

1) • Si es el caso que a es un argumento de la llamada e.f(...).
Sea par identificador del parámetro formal en la definición del método f de PT correspondiente a la aplicación de a como parámetro real en el llamado e.f(...,a,...).
Se debe hacer:
PT.**parameterize**((f,par) **as** T).

• Si es el caso que el resultado de la aplicación de la función e.f(...) se asigna a la entidad a, se debe hacer:
PT.**parameterize**((f,Result) **as** T).

2) Cambiar en la clase actual el tipo estático de e como PT_GEN[T_a].

4.4 Genericidad acotada

Cada identificador de parámetro de tipo genérico formal de la lista de argumentos de una orden **parameterize**, determina un conjunto CT con la influencia de la relación $\sim$. Se reúnen en CT todas las entidades que cambian su tipo al tipo genérico T en la clase genérica resultante.

Sea el conjunto F_{CT} definido como sigue
$F_{CT} = \{f | f$ se invoca a través de $e \in CT$ en la clase a parametrizar$\}$,
donde f denota también operadores ($+$, $<$,...). Si F_{CT} no es vacío, el parámetro genérico formal que identifica T debe restringirse a una clase que tenga (al menos) la funcionalidad que indica F_{CT}. Se asegura la existencia de una clase minimal y una clase maximal. La maximal es la clase que se corresponde con el tipo estático de dichas entidades antes de la aplicación del operador (dígase PT). La minimal es la clase ANY [16, 17]. Siguiendo los caminos de herencia desde ANY hasta PT, se encontrará una clase R cuyo conjunto de propiedades incluya a las propiedades que se indican en el conjunto F_{CT}.

De esta forma la clase paramétrica resultante tendrá el parámetro de tipo genérico formal T restringido a R,
A_GEN[...,T→R,...] (ver [16, 17]). Si R=ANY no es necesario hacer la restricción. Si R=TA se tiene el caso extremo.

De acuerdo con los ejemplos 1 y 3, en el cuerpo del método paint de la clase GRAPHIC_WINDOW a la entidad background se le pide pintarse (background.paint). Por tanto la clase genérica GRAPHIC_WINDOW_GEN[T] debe tener su parámetro genérico formal T restringido al menos a la clase GRAPHIC_ITEM. Esta clase contiene el método paint garantizando de esta forma que todas las clases con las que se instancie el parámetro genérico formal correspondiente al tipo de background incluirán el método paint. La clase genérica que se obtiene sería:

GRAPHIC_WINDOW_GEN[T→GRAPHIC_ITEM]

4.5 Limitaciones de implementación

La implementación de este operador en un entorno real de asistencia a la programación, para el lenguaje Eiffel fue el caso analizado aquí, trae consigo algunas dificultades importantes. En una aproximación más profunda y compleja podría intentarse resolver estos problemas pero hasta el momento este trabajo llega a presentarlos como limitaciones.

La primera de estas limitaciones proviene de las complicaciones de implementar una detección completa de las entidades relacionadas. De esta forma se podría dar el caso en el cual algunas entidades `a` y `b` de una clase no se hallen relacionadas y sí lo sean, mientras que una de ellas, por ejemplo `a`, deba cambiar su tipo para pasar a ser el identificador de un parámetro genérico formal. Esta es una limitación que se deriva de realizar la definición de "entidad relacionada" para el caso concreto de un lenguaje objetivo específico. En este caso la entidad `b` mantendría el mismo tipo estático que tenía en la clase original cuando en realidad debería cambiarlo. Es posible entonces que surjan errores de compilación cuando se definan nuevas clases instancias de la plantilla obtenida. Esta es una consecuencia de que en compilación aparezca, por alguna vía no considerada, la necesidad de que los tipos de ambas entidades conformen (note que en ningún caso esto ocurrirá para una clase instancia definida con el parámetro real correspondiente al tipo que tenía la entidad conflictiva en la clase original). En caso de que estos errores de compilación ocurran podría reaplicarse el operador pero demandando explícitamente la relación entre `a` y `b` (demanda explícita de relación: secciones 3.2 y 4.1).

La segunda es una consecuencia de las características particulares del lenguaje Eiffel. En una clase genérica no se permite aplicar instrucciones de creación sobre entidades que tienen un tipo correspondiente al identificador de un parámetro genérico formal. Entre otras razones, esta restricción se justifica por si la clase genérica se instancia con una clase abstracta. En este caso, se evita un intento de construir objetos a partir de una clase abstracta lo cual es un error. Si se encuentra el caso en que se aplica una instrucción de creación a una entidad cuyo tipo deberá cambiar a un parámetro genérico formal producto de la parametrización, en la presente versión se rechazará la posibilidad de efectuar dicha parametrización. Recientemente una nota técnica[3] de ISE[4] describe una propuesta con un conjunto de extensiones al lenguaje Eiffel enviadas al comité del standard del lenguaje (NICE = Nonprofit International Consortium for Eiffel). Dichas extensiones se centran en cómo permitir instrucciones de creación cuando el tipo de la entidad objetivo es un parámetro genérico formal. El objetivo principal se consigue con una extensión a la cláusula de restricción de los parámetros genéricos formales en la clase genérica y consecuentemente cambiando las reglas de validez. Si se aprueba la propuesta, el operador de parametrización para una herramienta que trabaje a partir de Eiffel se puede adaptar a la nueva situación y quedaría resuelta la presente limitación.

[3] disponible en http://www.eiffel.com/doc/manuals/language/generic-creation

[4] Interactive Software Engineering

4.6 Una breve comparación con C++

En esta sección se presenta una breve comparación entre algunos aspectos específicos relacionados con la introducción del operador de parametrización en una herramienta que trabaje a partir de Eiffel y otra para el lenguaje C++.

El primer aspecto a considerar es el concerniente a la definición de la relación "entidad relacionada". En C++ realizar dicha relación se torna más complejo y menos claro que para el caso de Eiffel. Esto es consecuencia fundamentalmente de la coexistencia de construcciones C y Orientadas a Objetos híbridas, el manejo explícito de memoria, las posibilidades de coerción de tipos (*type cast*), etc. Por otra parte, una particularidad de Eiffel que hace más clara la realización de "entidad relacionada" es la presencia de declaraciones por asociación con tipos anchor (**like**) como recurso del lenguaje. No existe en C++ un recurso equivalente.

Las repercusiones de la parametrización para las jerarquía de herencia en C++ es menos compleja que en Eiffel porque en C++ no se permiten redeclaraciones de tipos cuando una clase se deriva de otra a través de la herencia.

Otras diferencias significativas residen, por ejemplo, en que en C++ nunca se tendría el problema (descrito para Eiffel en la sección 4.5) que surge cuando en la clase a parametrizar existe alguna instrucción de creación sobre una entidad cuyo tipo debe pasar a ser el identificador de un parámetro genérico forma en la clase genérica resultante. Otro caso está dado porque en C++ no se puede considerar, como se hizo en el caso de Eiffel, la posibilidad de restringir las características de los parámetros de tipo genéricos en una forma estáticamente chequeable. En la tabla 4 se muestra un esquema comparativo.

5 Conclusiones

El ejemplo seguido para la presentación del operador **parameterize** (sección 2), es sólo un ejemplo juguete, pero en todo caso sirve para mostrar la utilidad fundamental que brinda el poder contar con dicho operador. En este sentido se pueden formular ejemplos de familias de clases tales como estructuras de almacenamiento, clases que encapsulan algoritmos, entre otras, que son susceptibles de ser objetivo de parametrizaciones con vista a incrementar su potencial de reutilización.

Como se puede deducir, la existencia de un operador como **parameterize** incrementa las capacidades de reutilización y hace más fácil la localización de nuevas clases en jerarquías de herencia previamente existentes. En adición a esto, se convierte en una operación alternativa, en los casos en que sea aplicable, para transformaciones de jerarquías de herencia, con una ventaja visible: provoca pocas implicaciones en código existente y prácticamente ninguna para aquellos objetos que sean persistentes, creados previamente al cambio.

Por otra parte, algunas recomendaciones de programación proponen implementar primero, antes de construir una clase genérica, una clase prototipo para un tipo básico particular que permita probar extensivamente dicha clase. Cuando se considere esta clase como correcta, entonces reescribirla como genérica. Contar con un operador como el que aquí se discute brinda la posibilidad, con las indicaciones apropiadas a la herramienta de asistencia a la programación, de ejecutar las modificaciones necesarias de forma automáti-

	Eiffel	**C++**
entidades relacionadas	• depende del grado de pesimismo que se asuma: del mayor pesimismo a los alias • declaración por asociación (tipos anchor) hace más claro el declarar entidades relacionadas	• oscurecido por la existencia de construcciones híbridas de C y OO, el manejo explícito de la memoria, la posibilidad de cohersión de tipos, etc. • no hay un similar para las declaraciones **like**
efectos de la herencia	• deben considerarse las redefiniciones de tipos. Esto se lleva a cabo restringiendo los parámetros genéricos formales	• no se permite redefinición de tipos en la herencia, entonces el proceso es menos complejo
genericidad acotada	• está presente como recurso del lenguaje. La genericidad acotada se expresa mediante *clases restricción* • la existencia de clases restricción está garantizada pero se obtiene teniendo en cuenta estructura (*signatura*) y no semántica	• no hay un similar para genericidad acotada • las restricciones en los tipos instancia se detectan en tiempo de "instanciación", por supuesto dependiendo de la estructura (*signatura*) de la instancia
instrucciones de creación	• no se permite aplicar una instrucción de creación sobre una entidad de tipo genérico formal (asegurando el chequeo estricto de tipos) • posibles soluciones: desde el rechazo de esta situación hasta realizar complejas transformaciones de código e interfaz. Puede esperarse la posibilidad de que se aprueben extensiones al lenguaje que si permiten esta situación, en ese caso habría que adaptarse a ello	• no hay restricciones para la creación relacionada con los tipos genéricos (debido a la detección en tiempo de "instanciación")

Tabla 4: Resumen comparativo: la introducción de **parameterize** desde el punto de vista de Eiffel y de C++

ca. En otros casos se tiene que algunos métodos de Análisis y Diseño Orientado a Objetos, como por ejemplo BON [22], incorporan un proceso de revisión de clases para modificarlas con el objetivo de prepararlas para su reutilización en otros dominios. Como parte de este proceso se encuentra el intento de hacer dichas clases más generales, esto incluye considerar su transformación en clases genéricas. Teniendo implementado en una herramienta CASE el operador **parameterize** este tipo de transformaciones puede realizarse automáticamente.

Otro caso que puede ser interesante resulta de si un lenguaje como Java, finalmente, opta por incluir alguna de las tantas propuestas que introducen genericidad como recurso del lenguaje [1, 3]. En este sentido podría ser útil la construcción de traductores que automaticen la actualización de bibliotecas existentes y para esto podría ser útil las consideraciones que aquí se hacen respecto al operador de parametrización.

Este trabajo hasta el momento define de forma general la propuesta de un operador de parametrización; los efectos que deben considerarse cuando se introduce en un entorno específico que trabaja para un lenguaje en particular (aquí se analizó el caso de una herramienta para Eiffel); y algunos problemas que se deben tener en cuenta para la trasformación apropiada de las clases con el objetivo de respetar las propiedades del lenguaje (la fortaleza de tipos, por ejemplo). Aparecen limitaciones cuando se quiere implementar esta operación en una herramienta específica. Sin embargo algunas pueden intentarse resolver. Evidentemente resulta muy interesante el desarrollo de otros análisis para diferentes lenguajes objetivo, pero más aún en nuestra opinión e interés, es el estudio de esta operación como alternativa, cuando es aplicable, para reorganizaciones de jerarquías de herencia.

6 Agradecimientos

Este trabajo ha sido parcialmente financiado por el proyecto CICYT TIC97-0593-C05-05. Yania Crespo es estudiante de doctorado gracias al Instituto de Cooperación Iberoamericana (ICI). Los autores agradecen especialmente al resto de los miembros del grupo GIRO (Grupo de Investigación en Reutilización y Orientación al Objeto) por su preocupación y apoyo y por todo lo que nos aportan en nuestras ricas e interminables discusiones cada viernes.

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 Microsystems, 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. available on the web: http://www.rational.com/ot/uml.html.

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

[6] E. Casais. Managing class evolution in object-oriented systems. Technical report, Centre Universitaire d'Informatique, University of Geneve, 1990.

[7] E. Casais. An incremental class reorganization approach. In O.L. Madsen, editor, *Proceedings of ECOOP'92, LNCS:615*, pages 114–132. Springer-Verlag, 1992.

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

[9] J-B. Chen and S.C. Lee. Generation and reorganization of subtype hierarchies. *Journal of Object Oriented Programming*, Jan 1996.

[10] T. Couso, M. Katrib, and M.A. Mateo. Smart compiling for Eiffel. *Journal of Object Oriented Programming*, 11(2):33–42, May 1998.

[11] Y. Crespo and M. Katrib. More about system level-validity in Eiffel. *Journal of Object Oriented Programming*, 11(4):40–49, July/August 1998.

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

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

[14] 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.

[15] J.M. Marqués. *Jerarquías de herencia en el diseño de software orientado al objeto.* PhD thesis, Universidad de Valladolid, 1995.

[16] B. Meyer. *Eiffel: the language.* Prentice-Hall Object-Oriented Series, 1991. second revised printing 1992.

[17] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.

[18] W.F. Opdyke. *Refactoring: a program restructuring aid in designing object-oriented application frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[19] 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.

[20] FAMOOS Esprit Project. FAMOOS: Framework based Approach for Mastering Object-Oriented Software evolution. Newsletter Number 04, September, 1998.

[21] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

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