Generación del cuerpo de los métodos a partir de la semántica de las operaciones del diagrama de clases

Generating the body of the methods from class diagram operation semantics

Carlos Mario Zapata J.[1] y Andrés Felipe Muñetón[2]

RESUMEN

Para la generación automática de código fuente a partir de los diagramas de UML, las herramientas CASE convencionales sólo generan el encabezado de los métodos, y algunos esfuerzos adicionales generan parcialmente el cuerpo de los métodos, pero empleando elementos que se alejan de los estándares de modelado y, en todo caso, muy cercanos a la elaboración manual de código fuente. Buscando superar esas limitaciones, en este artículo se propone un proceso para generar el cuerpo de los métodos del código fuente a partir de las operaciones del diagrama de clases. Para ello se define la "semántica de las operaciones", que es una manera de ligar las operaciones del diagrama de clases y los métodos implementados en la plataforma de desarrollo, tomando como entradas las pre y poscondiciones de las operaciones y el metamodelo de las librerías de la plataforma de desarrollo. Finalmente, el proceso se ejemplifica con un caso de estudio, para el cual fue necesario elaborar una instancia en UML del metamodelo del paquete java.sql.

Palabras clave: generación de código, UML, precondición, poscondición, plataforma de desarrollo.

ABSTRACT

Well-known CASE tools only generate the heading of the methods to automatically generate source code from UML diagrams. Some proposals partially generate the body of the methods; however they use non-standard modeling elements or hand-made source code elements. This paper proposes a process for generating the body of the methods from class diagram operations in an attempt to overcome such constraints. "Semantics of class operations" was thus defined as a way of linking class diagram operations to development platform implemented methods. These kinds of semantics use pre- and post-conditions belonging to the operations and the development platform library meta-model. This process is also exemplified by giving a case study. An UML instance of the java.sql package meta-model was created for developing the case study.

Keywords: code generation, UML, pre-condition, post-condition, development platform.

Recibido: mayo 13 de 2008
Aceptado: octubre 21 de 2008

Introducción

La ingeniería de *software* provee herramientas CASE como una manera de asistir a los analistas en el proceso de desarrollo de *software*. Una de las tareas que tradicionalmente se aducen para el uso de tales herramientas es la generación automática de código fuente desde diagramas. En particular, se destacan algunas herramientas como Together® (Borland Software Corporation, 2008) y Rational Rose® (IBM Corporation, 2008), que generan algo de código a partir del diagrama de clases, pero que, en términos de los métodos correspondientes a las clases de implementación, sólo genera el encabezado de los mismos. La herramienta CASE Fujaba® (University of Paderborn, 2008; Geiger y Zündorf, 2005) genera automáticamente el cuerpo de los métodos, pero para ello recurre a elementos no estándar de UML.

Ahora, algunas propuestas, como rCOS (Liu y Jifeng, 2005) y Método-B (Laleau y Mammar, 2005), efectivamente obtienen el cuerpo de los métodos de las clases, pero utilizan lenguajes formales para la representación de los diagramas UML y su posterior transformación a un lenguaje de programación. Además, tales lenguajes no se implementan en las herramientas CASE convencionales y se acercan más al código fuente que al modelado.

Como una solución a los inconvenientes anotados, en este artículo se presenta un proceso para la generación automática de código a partir de diagramas, que define el concepto de "semántica de las operaciones" como un vínculo entre los modelos de diseño del usuario y la plataforma de desarrollo. Se emplean, en este proceso, las pre y poscondiciones de las operaciones e información semántica adicional que indica el objetivo que cumple cada operación, además del metamodelo de las librerías correspondientes a los lenguajes de programación.

El resto de este artículo se estructura así: inicialmente ofrece una vista general de los avances logrados en la generación automática de código; luego se propone el proceso para la generación del cuerpo de los métodos a partir de la denominada "semántica de las operaciones", específicamente para Java® como lenguaje destino; posteriormente presenta un caso de estudio en el que se aplica el planteamiento teórico de este artículo en la generación del código de una operación que recupera datos de la base de datos.

[1] Ingeniero civil, Especialista, en Gerencia de Sistemas Informáticos, M.Sc., en Ingeniería de Sistemas y Ph.D., en Ingeniería, Universidad Nacional de Colombia, Medellín. Profesor asociado, Escuela de Sistemas, Universidad Nacional de Colombia, Medellín. Líder, grupo de Investigación en Lenguajes Computacionales. cmzapata@unal.edu.co

[2] Ingeniero de sistemas e informática. M.Sc., en Ingeniería de Sistemas. Profesor de cátedra, Universidad Pontificia Bolivariana. Ingeniero de proyectos, PRAGMA .SA. Integrante, grupo de Investigación en Lenguajes Computacionales, Universidad Nacional de Colombia, Medellín. andresfelipeml@gmail.com

Finalmente, se dedica a las conclusiones y el trabajo futuro que se desprenden de esta propuesta.

Un panorama general de la generación automática de código

Considere el siguiente método en el lenguaje de programación Java®:

```
public void guardarUsuario(u:Usuario) {
Connection con = DriverManager.getConection("datos
de la conexión");
PreparedStatement ps = con.preparedStatement("insert
into usuarios values(?,?);");
ps.setString(1,u.getID());
ps.setString(2,u.getNombre());
ps.execute();}
```

La Figura 1 muestra un posible diagrama de secuencias para este método.

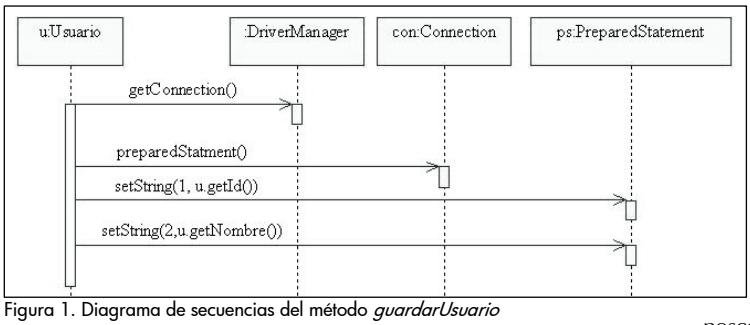

Figura 1. Diagrama de secuencias del método *guardarUsuario*

Existe un gran número de herramientas CASE que permiten crear diagramas como el de la Figura 1. Entre ellas están Together®, Rational Rose ® y Fujaba®. Sólo en el caso de Together® y Fujaba® es posible obtener código fuente en Java® a partir del diagrama de secuencias, aunque adicionándole ciertos elementos, como los *story diagrams* en el caso de Fujaba® (Geiger y Zündorf, 2006), que son combinaciones de los diagramas de colaboración y máquina de estados de la versión 1.4 de UML. La especificación estándar de UML no cobija los *story diagrams*. En el caso de Together®, la generación de código se produce desde el diagrama de secuencias, pero incorporando los denominados "bloques de notación", que son elementos no estándar de dichos diagramas que incluyen fragmentos de código en el lenguaje destino. Las demás herramientas sólo permiten generar código a partir del diagrama de clases.

Aunque el diagrama de la Figura 1 permite generar el código deseado, el diseñador debe, además de crear sus diagramas, tener disponible o crear él mismo un modelo de las librerías de la plataforma de desarrollo para que se puedan utilizar clases como: *DriverManager*, *Connection* y *PreparedStatement,* propias de Java®. Un modelo de plataforma que sea útil en la generación de código deberá ser tan detallado como para incluir las asociaciones entre clases, atributos y operaciones, lo cual implica los tipos de datos y visibilidad en ambos casos, y parámetros en el caso de las operaciones.

Algunos trabajos, como Método-B (Laleau y Mammar, 2005) y rCOS (Liu y Jifeng, 2005), se enfocan en la formalización de UML utilizando un lenguaje formal intermedio. La generación del código en estos trabajos, implica la transformación del modelo al lenguaje formal y de este al lenguaje de programación deseado; es un proceso similar al de las herramientas CASE, pero se supone que el paso intermedio de formalización permite reducir o eliminar la ambigüedad de UML (Aronson y Grossman, 2005).

En síntesis, los trabajos actuales transforman diagramas UML de diseño en código con dos limitaciones apreciables: la incorporación de elementos no estándar de los diagramas, que además requieren fragmentos de código en el lenguaje destino, y la utilización de lenguajes formales intermedios, que también tienen parecido con el lenguaje destino.

Un proceso para la generación automática del cuerpo de los métodos en Java®

La semántica de las operaciones

Toda operación requiere para su ejecución que el sistema al que pertenece esté en un estado particular y, luego de su ejecución, puede ocasionar que el sistema continúe en su estado actual o haga una transición hacia un nuevo estado. El estado requerido para la ejecución de la operación es su precondición, y el estado final es su poscondición. La operación correspondiente al diagrama de secuencias de la Figura 1 tiene como precondición que el usuario *u* no exista en el conjunto *usuarios,* y su poscondición es agregar el usuario *u* a *usuarios*, que es el conjunto que representa a los usuarios almacenados en la base de datos.

guardarUsuario (u:Usuario){
pre: $\neg\exists u \in usuarios$
post: $usuarios = usuarios \cup \{u\}$ }

Según Morgan (1998), una especificación basada en pre y poscondiciones se puede refinar a un código entendible por una máquina, es decir, un código compilable y ejecutable. Aplicar esta idea a la especificación de *guardarUsuario* para obtener la representación descrita anteriormente implica conocer la secuencia de instrucciones en Java® que permiten alcanzar la poscondición de esta operación. La secuencia es la siguiente: *crear la conexión con la base de datos, preparar la consulta* y *ejecutar la consulta*. Para ejecutar la instrucción 3 es necesario que el sistema se encuentre en el estado dado por la poscondición de la instrucción 2, es decir, que se ejecute exitosamente esta instrucción. De igual manera, la instrucción 2 requiere el estado dado por la poscondición de la instrucción 1 para que se pueda ejecutar. Para generar automáticamente el código Java® del método correspondiente a una operación se requiere, inicialmente, comprender la estructura de las librerías que se emplean para generar el código de forma manual. Con este fin, se propone en este artículo una instancia del metamodelo de Java® que incluye algunas clases del paquete java.sql, utilizadas en el código de la operación *guardarUsuario*; dicha instancia se muestra en la Figura 2. En el metamodelo de Java® las operaciones de los diagramas de clases se implementan como Métodos. Nótese que algunos métodos de la Figura 2 se relacionan por medio de su parámetro de retorno, identificado con el *rol returnParameter,* y el tipo de este parámetro, identificado con el *rol type*.

La Figura 3 muestra un conjunto de especificaciones con pre y poscondiciones que representan los métodos de la Figura 2. Se aprecia que, en la relación entre los métodos, además de las poscondiciones intervienen las precondiciones. La relación que existe entre las pre y las poscondiciones que enlazan varios métodos Java® se denomina, en este artículo, "semántica de la operación". Con base en la semántica de la operación *guardarUsuario*, consignada en la Figura 3, es posible obtener el siguiente código:

Connection con = DriverManager.getConnection();
PreparedStatement ps = con.preparedStatement("");
ps.execute();

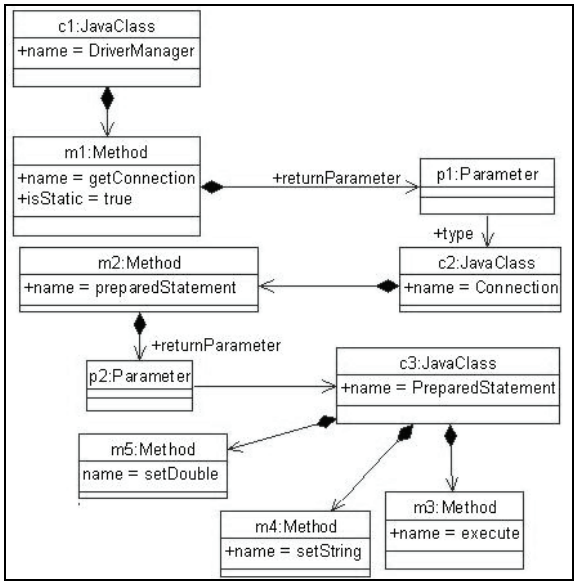

Figura 2. Diagrama de algunas clases del paquete java.sql de Java® como instancia del metamodelo de UML

Para generar este código se pueden establecer las siguientes reglas: si la operación no es estática entonces tiene como precondición la creación de un objeto de la clase que la contiene; si la operación es estática, no requiere un objeto de la clase que la contiene para que se pueda invocar; el tipo del parámetro de retorno de una operación es la poscondición de esa operación.

Inicialmente, se identifica la operación *execute* como aquella que permite almacenar elementos en la base de datos; *execute* no es estática, entonces tiene como precondición la creación de un objeto de su clase contenedora *PreparedStatement*. A su vez, la operación *preparedStatment* tiene como poscondición un objeto de *PreparedStatement*. La operación *preparedStatement* no es estática, así que tiene como precondición un objeto de su clase contenedora *Connection*. La operación *getConnection* tiene como postcondición un objeto de tipo *Connection*. La operación *get Connection* es estática, por lo cual no requiere un objeto de su clase contenedora y no tiene alguna otra precondición.

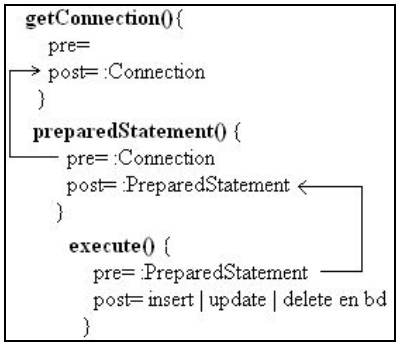

Figura 3. Relaciones de pre y poscondiciones entre métodos Java®

Al comparar el código que se espera obtener (véase el inicio de la Sección 2) con el generado, este último parece incompleto, pues no incluye las sentencias que invocan la operación *setString*. Con las reglas seguidas para generar el código no se tienen las relaciones completas para generar un código que incluya todas las sentencias necesarias para llegar a una poscondición dada. Por ejemplo, el método *setString* no se asocia con *execute*, excepto que comparte la misma clase dueña, *PreparedStatement*. Tampoco es claro cómo se puede relacionar el código obtenido con el método que contendrá el código, pues no hay una relación entre los modelos de usuario que contienen la operación y la estructura de la plataforma.

Adición de asociaciones de pre y poscondiciones al metamodelo de la plataforma de desarrollo

Para determinar la secuencia completa de sentencias para un método, el desarrollador, normalmente, consulta la documentación de cada plataforma y encuentra las relaciones explícitas entre las operaciones dadas por sus tipos de retorno o sus parámetros y las relaciones implícitas especificadas textualmente en la documentación. Por ejemplo, la documentación de Java® incluye un ejemplo del uso de la operación *execute* de *PreparedStatement*, el cual indica que antes de esta operación se deben invocar las operaciones *setX*. Si el proceso no fuera manual, sería necesario complementar las pre y poscondiciones de las operaciones de la plataforma con el fin de generar secuencias de sentencias, en las cuales cada instrucción prepare el sistema en el estado que requiera para su ejecución. Siguiendo con el caso de la operación *guardarUsuario*, además de un objeto de tipo *PreparedStatement*, la operación *execute* tiene como precondición el estado alcanzado por operaciones *setX()*, donde *X* puede ser *String, Double, Int*, entre otras, de la clase *PreparedStatement*. Así, si se pretende automatizar el proceso de generación de código, toda relación entre las operaciones de la plataforma se debe declarar explícitamente.

La Figura 2 contiene las relaciones entre los elementos de Java® especificados en el metamodelo de ese lenguaje. Al modificar este metamodelo para que acepte el uso de pre y poscondiciones se obtienen instancias que incluyen todas las relaciones entre las operaciones de la plataforma. Con este fin, se debió modificar el metamodelo de Java® para complementarlo con las asociaciones pre y post en las metaclases *Method* y *PreconditionGroup*. El metamodelo modificado se presenta en la Figura 4. Adicionalmente, se agrega la siguiente restricción en OCL para evitar que un método sea precondición o postcondición de sí mismo:

self.precondition -> forAll(m | m <> self) and self.postcondition -> forAll(m | m <> self)

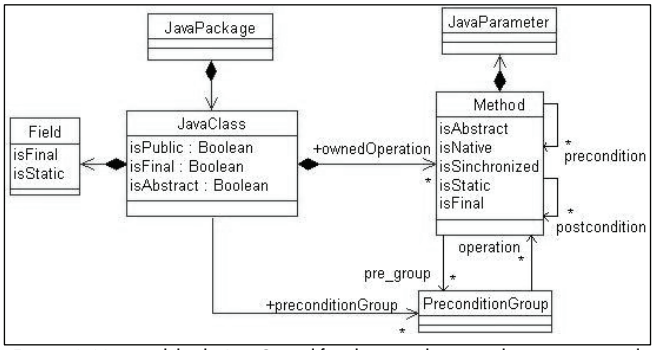

Figura 4. Metamodelo de Java® modificado con relaciones de pre y postcondiciones

Una instancia de la metaclase *Method* puede tener como precondición o poscondición una o más instancias de *Method*. Las relaciones pre y post se emplean para complementar las precondiciones y poscondiciones que no se establecen originalmente en la estructura de cada plataforma, en este caso Java®.

La Figura 5 muestra las operaciones *setString* y *setDouble* de *PreparedStatement* contenidas en una instancia de la metaclase *PreconditionGroup,* cuya representación gráfica es similar a la metaclase *CombinedFragment* del metamodelo de UML. El *preconditionGroup* se utiliza para indicar que cualquiera de las operaciones que contiene puede ser precondición de la operación *execute*. La elección de una u otra operación dependerá del tipo de dato que se pretende almacenar en la base de datos.

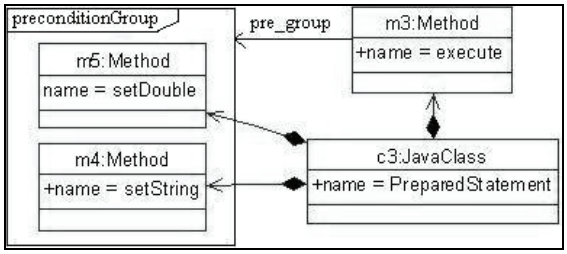

Figura 5. Precondiciones de execute vistas desde una instancia del metamodelo de UML

Relación entre los modelos de usuario y la plataforma de desarrollo

La relación entre las operaciones de los modelos de usuario y los métodos de la plataforma de desarrollo es el último paso necesario para lograr la generación automática de código. Nótese que, todavía, no se indica cómo identificar un método en la plataforma que lleve a la misma poscondición de la operación que se está traduciendo a código.

Ya que las poscondiciones de las operaciones de los modelos de usuario pertenecen a un dominio específico, no es posible crear relaciones como las identificadas para las operaciones de la plataforma. Por ejemplo, no hay una relación clara entre *guardar Usuario* y *execute*, pero se está asumiendo que sus poscondiciones permiten alcanzar el mismo objetivo.

Si se deja de lado el dominio y se generaliza el objetivo de cada operación, es posible establecer una relación, en este caso semántica, entre las operaciones de los modelos de usuario y las de la plataforma. Este artículo se enfoca en el manejo de datos persistentes, pero la propuesta se puede extender a cualquier otro tipo de operación.

En la Figura 6 se construye un *profile* de especificaciones que permiten relacionar semánticamente, mediante estereotipos, las operaciones de usuario y los métodos de la plataforma. Si el diseñador puede identificar la semántica de las operaciones de su modelo, entonces puede establecer una asociación semántica con alguna clase de especificación como las de la Figura 6. Si, adicionalmente, los métodos de la plataforma también tienen una asociación con especificaciones que indiquen cuál es su semántica, entonces se contará con una relación entre las operaciones de los modelos de usuario y los métodos de la plataforma. En la Figura 6, las especificaciones relacionadas con el manejo de objetos en bases de datos se agrupan en la metaclase abstracta *DBSpecification*.

La especificación *DCLSpecification* se utiliza para indicar que una operación permite hacer consultas en la base de datos. Una operación que se estereotipe como *DDLSpecification*, que es una metaclase abstracta, de acuerdo a la relación de generalización y la propiedad de polimorfismo del paradigma orientado a objetos, tendrá como semántica la inserción (*InsertSpecification*), actualización (*UpdateSpecification*) o eliminación de elementos (*Delete Specification*) de la base de datos. La operación *guardarUsuario* se puede estereotipar con una especificación de tipo *Insert Specification*, como se aprecia en la Figura 7.

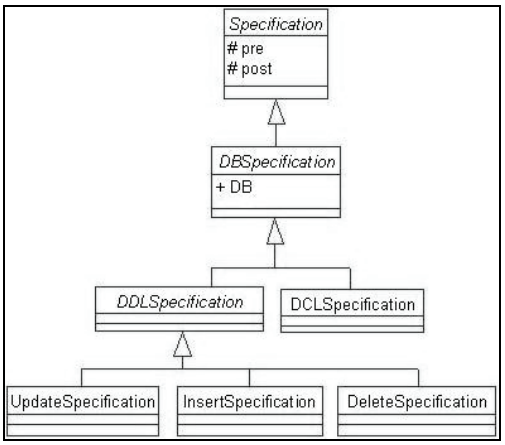

Figura 6. Especificaciones para relacionar modelos de usuario y plataforma

Figura 7. Clases estereotipadas con especificaciones semánticas

La Figura 7 también muestra el método *execute* y su estereotipo de tipo *MSpecification*. La relación entre las dos operaciones de la Figura 7 se puede apreciar en la Figura 8, donde se muestran las mismas clases pero como instancias del metamodelo.

La especificación de la operación de usuario permite ligar un tipo específico con las operaciones que tienen especificaciones abstractas. En el caso de *execute*, se selecciona la especificación *Insert Specification*, pues es la relacionada con la operación *guardar Usuario* y hace parte de la jerarquía de *DDLSpecification*, metaclase de especificación para *execute*.

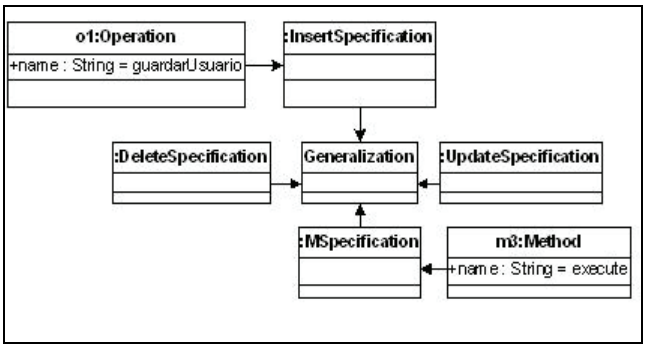

Figura 8. Instancia de relación por estereotipos entre operaciones

Caso de estudio

El diagrama de clases de la Figura 9 contiene la clase *Usuario*, estereotipada como una entidad de acuerdo al *profile* de bases de datos de UML (OMG, 2008), y la clase *AdministradorUsuarios*, que tiene dos operaciones: *guardarUsuario*, estereotipada como una operación que almacena usuarios en la base de datos, y *get Usuario*, estereotipada como una operación que recupera un usuario de la base de datos a partir de la identificación del mismo.

Para obtener la implementación de la operación *getUsuario* se debe encontrar un método implementado que se relacione con la misma especificación de *getUsuario*, o que lo esté transitivamente

en una relación de generalización. Luego, se obtiene, iterativamente, una secuencia de operaciones relacionadas por sus pre y poscondiciones que permita cumplir la poscondición de la operación *getUsuario*.

A continuación se presenta, paso a paso, el proceso de generación de código para la operación *getUsuario*. El código se complementa a medida que se encuentran los elementos que lo componen. Todos los objetos se nombran como su clase, pero iniciando con minúscula.

Como se aprecia en la Figura 10, la operación *executeQuery* de la clase *PreparedStatement* se relaciona con la misma especificación, *DCLSpecification*, de *getUsuario*, lo que en esencia significa que poseen la misma semántica de operación; *executeQuery* tiene como poscondición un objeto de tipo *ResultSet* y, al no ser estática, tiene como precondición un objeto de tipo *PreparedStatement*. Estas características generan el siguiente código:

```
ResultSet resultSet = preparedStatement.executeQuery();
```

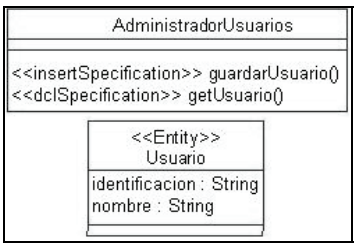

Figura 9. Diagrama de clases de manejo de usuarios

La operación *preparedStatement* de la clase *Connection* permite crear objetos de tipo *PreparedStatement*. Esta operación no es estática, por lo que requiere de la creación de un objeto de su clase contenedora. En términos de código Java®, esta aserción se traduce en:

```
PreparedStatement preparedStatement =
connection.preparedStatement();
ResultSet resultSet = preparedStatement.executeQuery();
```

La operación estática *getConnection* de la clase *DriverManager* retorna objeto de tipo *Connection*. Esta operación no tiene precondiciones. La implementación de la operación es:

```
public Usuario getUsuario(Usuario u){
Connection connection =
DriverManager.getConnection();
PreparedStatement preparedStatement =
connection.preparedStatement();
ResultSet resultSet =
preparedStatement.executeQuery();}
```

Nótese que el método generado permite la recuperación de un registro de la base de datos, pero no de un objeto de tipo *Usuario*. El método esperado podría ser como el siguiente:

```
Connection connection =
DriverManager.getConnection();
PreparedStatement preparedStatement =
connection.preparedStatement();
ResultSet resultSet = preparedStatement.executeQuery();
Usuario u = new Usuario();
if(next()){
u.setIdentificacion(resultSet.getString("identificacion"));
u.setNombre(resultSet.getString("nombre"));}
return u;
```

El procedimiento propuesto en este artículo parte de la poscondición de la operación para la generación del código. En ese caso, la poscondición es retornar un objeto de tipo *Usuario* que se almacena como un registro en la base de datos. Sin embargo, la poscondición de la operación implementada *executeQuery* sólo considera el caso más abstracto: recuperar el registro de la base de datos sin importar a qué objeto pertenece. Es responsabilidad del programador agregar el código necesario para asignar los datos del registro a los atributos del objeto que espera. Para el método de la operación *getUsuario* el programador debe utilizar las operaciones de la clase *ResultSet* para obtener el resultado de la consulta y crear el objeto.

En este artículo se utilizan los comentarios como complemento al código que se puede generar analíticamente. En la Figura 10 el comentario informa sobre la necesidad de utilizar las operaciones de *ResultSet* una vez ejecutada la consulta. Este comentario se agrega al código, también como comentario, como se aprecia a continuación.

```
public Usuario getUsuario(String identificacion){
Connection connection =
DriverManager.getConnection();
PreparedStatement preparedStatement =
connection.preparedStatement();
ResultSet resultSet = preparedStatement.executeQuery();
/* Generated Note:
* Use the ResultSet methods to obtain the data from the
resultSet object
*/}
```

De esta manera, aunque el código no es completo, al haber alcanzado una poscondición general se indica al desarrollador que debe utilizar determinadas operaciones para completar el código. En un futuro se espera agregar elementos a cada plataforma que permitan deducir qué operaciones complementarias se requieren para alcanzar poscondiciones específicas.

Conclusiones y trabajo futuro

En el artículo se presentó una propuesta para la generación automática del código fuente del cuerpo de los métodos, a partir de la relación entre operaciones de modelos de diseño y métodos implementados en Java®, relacionados por sus pre y poscondiciones. Esas relaciones se denominaron "semántica de las operaciones".

La estructura de la plataforma Java® se complementó con relaciones de pre y poscondiciones que permitieron obtener un código más completo que el que se puede generar a partir de las herramientas y propuestas actuales. Sin embargo, aún queda por incorporar mayor información a la estructura que indique cómo se pueden agregar al código sentencias complementarias, una vez se alcanza la poscondición del método implementado.

Es posible que una operación no se pueda asociar con una especificación en particular, debido a que algunas de las sentencias que conforman su implementación tengan poscondiciones que requieran generar código como si se tratase de una operación más. Este es el caso de operaciones que en su implementación combinen operaciones de inserción, actualización, eliminación y consulta. El análisis de esta situación es uno de los asuntos que requieren trabajo futuro. Otros asuntos que necesitan mayor análisis, son: la extensión de la semántica de las operaciones a otros tipos de operaciones, por ejemplo algebraicas o definiciones de funciones, que

no se ocupan de datos persistentes, como los que se trabajaron en este artículo; la generalización del enfoque propuesto a otras plataformas de desarrollo, como C++ o C#; y la incorporación de este trabajo en herramientas CASE convencionales.

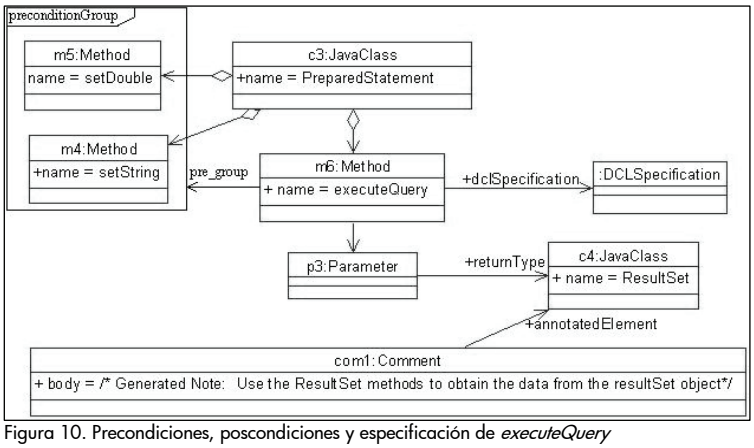

Figura 10. Precondiciones, poscondiciones y especificación de *executeQuery*

Bibliografía

Aronson, J., Grossman, M., Does UML Make the Grade? Insights from the software development community., Inf. and Soft. Tech., Vol. 47, No. 6, 2005, pp. 383-397.

Borland Software Corporation., Borland Together Architect®. En: http://www.borland.com/us/products/together/index.html (Consultado Abril de 2008).

Geiger, L., Zündorf, A., Statechart Modeling with Fujaba., Electronic Notes in Theoretical Computer Science, Vol. 127, 2005, pp. 37-49.

Geiger, L., Zündorf, A., Tool Modeling with Fujaba, Electronic Notes in Theoretical Computer Science., Vol. 148, 2006, pp. 173-186.

IBM Corporation., Rational Rose Architect®., En: http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html (Consultado Abril de 2008).

Laleau, R., Mammar, A., From a B formal specification to an executable code: applicational to the relational database domain., Inf. and Soft. Tech., Vol. 48, No. 4, 2005, pp. 253-279.

Liu, Z. y Jifeng, H., Towards a Rigorous Approach to UML-Based Development., Electronic Notes in Theoretical Computer Science, Vol. 130, 2005, pp. 57-77.

Morgan, C., Programming from Specifications., Second Edition, Hempstead, Prentice Hall International, 1998.

OMG., Object Management Group. UML data modelling profile., En: http://www.omg.org/cgi-bin/doc?ab/05-12-02 (Consultado Abril de 2008).

University of Paderborn., Software Engineering Group. FUJABA Tools Suite., En: http://wwwcs.uni-paderborn.de/cs/fujaba/index.html (Consultado Abril de 2008).