A proposal to generate the method code based on class diagram and java® meta-model[1]

Una propuesta para la generación del código de los métodos desde el diagrama de clases y el metamodelo de java®

Carlos Mario Zapata Jaramillo,
Ph.D. en Ingeniería

Andrés Felipe Muñetón Lopera,
M. Sc. en Ingeniería de Sistemas

Abstract

Code generation, one of the final phases of software development lifecycle, has been made using CASE tools. However, the method code has not been successfully accomplished in the well-known CASE tools. To meet this challenge, in several proposals researchers employ UML diagrams in order to generate some parts of the method code, even though UML diagrams are given in a non-standard use. We introduce, in this paper, a proposal to generate the method code from class diagrams (complemented with pre- and post-conditions) and Java® meta-models. We exemplify the use of this proposal with a case study.

Keywords: CASE tools, class diagram, code generation, meta-modeling, pre- and post-conditions.

[1] This paper is an English version of the paper "Generación del cuerpo de los métodos a partir de la semántica de las operaciones del diagrama de clases", published on the journal *Ingeniería e Investigación*, volume 28, issue 3, 2008, pp. 58-63. The authors state that, besides translation, several amendments and minor additions have been included.

[2] The authors belong to the Computational Language Group, Universidad Nacional de Colombia. E-mail: cmzapata@unal.edu.co, afmuneto@unal.edu.co

Recibido: 11 de junio 2011
Aprobado: 26 de octubre 2011

Resumen

La generación de código, una de las fases finales del ciclo de vida del software, se viene realizando mediante herramientas CASE. Sin embargo, el código de los métodos no se realiza de manera exitosa con las herramientas CASE tradicionales. Como respuesta a este problema, existen propuestas que emplean algunos de los diagramas de UML para generar porciones del código de los métodos, pero esas propuestas emplean, con este fin, modelos o elementos que se alejan del estándar de UML. En este artículo se introduce una propuesta para generar el código de los métodos a partir del diagrama de clases (complementado con pre y post condiciones) y el metamodelo de Java®. El uso de esta propuesta se ejemplifica con un caso de estudio.

Palabras clave: diagrama de clases, generación de código, herramientas CASE, Metamodelado, pre y postcondiciones.

Introduction

Software development lifecycle has the following phases: definition, analysis, design, construction, transition, and production. During the construction phase, the models defined in previous stages are used to generate an executable code of the final application. This process is partially assisted by the well-known CASE tools. Some of these tools, like Rational Rose® (Quatrani, 2000), Together® (2011), Poseidon® (2011), and ArgoUML® (Robbins *et al.*, 1997) are capable of generating code in several languages like Java® or C++. Also, some platforms are used to perform such task; for example, NetBeans 6.8 (2011).

The resulting code obtained from the above mentioned CASE tools is still immature, and there are two reasons for this: most of the tools only generate code from class diagram—given that this diagram is structurally similar to the source code—and the method contents from the generated code classes lack the method body—these CASE tools only include the head of the code. In the case of the NetBeans 6.8 platform (2011), we can use a set of predefined “templates” included inside the “code generator” feature, but only in the development phase. We need, instead, tools to generate code from the design phase.

The software engineering community has reacted to these problems with new proposals that use other diagrams or elements in the code generation process. Three examples of these proposals are Fujaba® (Niere & Zündorf, 1999), rCos (Liu & Jifeng, 2005), and B-method (Mammar & Laleau, 2006).

Fujaba® (Niere & Zündorf, 1999) uses class diagram, and a non-standard combination of state machine and sequence diagram. Keeping in mind that the use of non-standard elements in UML standard diagrams poses a major usability problem in this CASE tool, the entire process is completely linked to customized versions of the UML standard, and changes to this standard will not be reflected upon it. Also, the device selected to represent the body of the methods is a text with the same structure of the source code to be obtained. The invested effort in the task of model building with these additions is comparable to the invested effort in manually writing code.

The rCos (Liu & Jifeng, 2005) and the B-method (Mammar & Laleau, 2006) are similar proposals in that they both use UML diagram formalizations in order to generate code. Some of the formalized diagrams are class and sequence diagrams. These proposals, as in the case of Fujaba®, pose some problems:

- The method body is represented by a formal language originated before the code generation process (with an invested time substantially longer than the time employed in the manual encoding process).
- The process itself is restricted to methods with database semantics (for example, the insertion of a record in a database).
- The use of non-standard elements makes the code generation process dependent of customized versions of the artifacts.
- There are no CASE tools that use this approach.

As a way to improve the code generation process and surpass the listed problems of previous proposals, we argue, in this paper, that the method code can be obtained from class diagram and pre- and postconditions belonging to the operations of such a diagram. We also complement this approach by adding

semantic information (represented by new elements) to the Java® meta-model expecting to match the cited post-conditions. This approach is shown by means of a case study.

The rest of the paper is structured like this: first, we discuss the state-of-the-art in automated code generation; second, we propose a method to link class operations to code methods from pre- and post-conditions; third, we propose a modification to the Java® meta-model in order to include pre- and post-condition associations; next, we present the relationship between user models and Java® development platform; then, we outline a case study illustrating this proposal; and finally, we provide conclusions and future work perspectives.

Automated code generation

A code fragment belonging to a method in the Java® programming language looks like this:

```
public void saveUser(u:User) {
Connection con = DriverManager.getCo-
nection(“”); PreparedStatement ps = con.
preparedStatement(“insert into usuarios values(?,?);”);
ps.setString(1,u.getID());
ps.setString(2,u.getName());
ps.execute();
}
```

This method is used to insert a new user into an existing table, as a way to deal with database management. Figure 1 shows a possible sequence diagram matching this method.

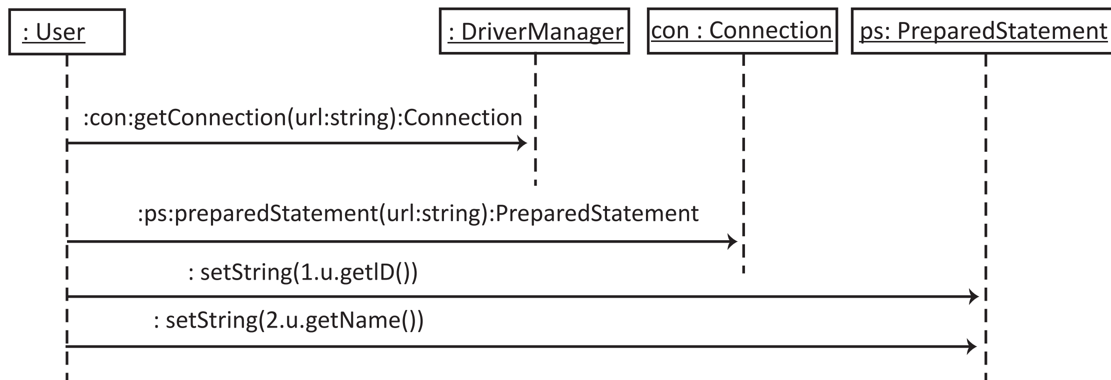

Figure 1. A possible sequence diagram for the *saveUser* method.

Most of well-known CASE tools exhibit capabilities to draw diagrams like the one in figure 1, but only Together® (2011) and Fujaba® (Niere & Zündorf, 1999) are able to generate Java® code from this diagram. The remaining tools might only generate incomplete source code from class diagram. When source code generation from sequence diagram is possible, the designer must add some special features to this code in order to complete it. In the previous example, some classes like *DriverManager*, *Connection*, and *PreparedStatement*, belonging to the Java® language, must be manually added to the resulting code. On the other hand, NetBeans 6.8 (2011) is unable to automatically generate such a code, since it does not match the predefined templates, which are: sample generator, constructor, override method, and add property.

On the other hand, Model-Driven Architecture (MDA) is an initiative of the Object Management Group (Kepple *et al.*, 2003) intended to develop standards based on the idea that modeling is a better foundation for developing and maintaining systems. MDA suggests the use of profiles to adapt the business area models (platform independent models—PIM—in the jargon of MDA) to the programming language (platform specific models—PSM—as understood by MDA practitioners). Some of the well-known CASE tools, for example Rational Rose® (Quatrani, 2000), exhibit capabilities to include

profiles in order to complement the models with information about the targeted programming language. However, even such capabilities are not enough to include the required information in the method to be generated. In the *saveUser* method, from the previous example, the information the designer must add belongs to Dynamic Link Libraries (DLL) of the Java® language, and this information is not supported by any Java® profile.

Different to the source code previously stated, other kind of source code is related to calculus; for example, the computation of the absolute value of a number or the result of an algebraic operation. Looking at the following expression of source code:

```
x:[ x = abs y]
```

If we have to program a method to explain this expression, we have to know the sense of the implicit operation; for example:

```
if
y>= 0 --> x:=y
|| y < 0 --> x:= -y
endif
```

The process of recognizing the procedure to calculate the result of an operation, called refinement, is not supported by most of CASE tools. Also, other proposals as rCos (Liu & Jifeng, 2005) and B-method (Mammar & Laleau, 2006) employ a formalization of this refinement process, but this process must be explicitly defined to complete the code generation. In these proposals, formalization is an intermediate representation of the source code, and it poses some drawbacks for the entire code generation process:

- Formal languages are more difficult to manage than programming languages. The time the designer must invest in the formalization process is longer than the time devoted to manual code writing.
- Formal languages do not have standard guidelines for well-known CASE tools. Also, there are multiple notations of this kind of languages. If standardization is not possible in the code generation process, there will be a dependency on customized versions of the formal language.
- Both rCos (Liu & Jifeng, 2005) and B-method (Mammar & Laleau, 2006) are theoretical approaches, and there is no CASE tool *supporting* these theories.

Relationship among operations, from pre- and post-conditions

Every operation requires—as an executing initial condition—a set of states in the particular system it belongs to. After the execution of the operation, the system can remain the same or can become a new set of states. The initial set of initial conditions is named pre-conditions and the final set of states is named post-conditions. The *saveUser* operation, previously defined, has the following discussed conditions:

```
saveUser (u:User){
pre: not exist u:User
post:u=User +{u}
}
```

User is a set of users that has been stored in the database.

According to Morgan (1998), the specs are based on pre- and post-conditions. Those specs can be refined up to machine-understandable code, that is, a text that can be compiled and executed. When applying this concept to the *saveUser* operation, we must write down the following sequence of Java® commands to reach the post-condition (in this case, to store a new user in the database): (1) to create a connection to the database, (2) to prepare the query, and (3) to execute the query. To accomplish the initiation of the execution of the query (3rd step), the system needs to reach the states demanded by the post-condition in the second step. Similarly, the preparation of the query (2nd step) can only be initiated when the post-condition in the first step is reached.

Due to the fact that we need to express pre- and post-conditions in the Java® language, we propose an instance of the Java® meta-model, in order to describe classes belonging to the *java.sql* package. Figure 2 shows the part of this instance, including the classes used by the *saveUser* operation.

We must highlight the fact that one parameter, in figure 2, acts as a relationship connector between one method (belonging to a class) and another class. In the same line of reasoning, one role acts as a navigation connector; for example, by means of *returnParameter* the system is capable of executing a new method. If we assume that *returnParameter* is

the method post-condition, then all the methods represented in Figure 2 are linked by their post-conditions. Figure 3 shows the pre- and post-condition specs to the *saveUser* operation. In this Figure, pre- and post-conditions participate in the relationships among methods.

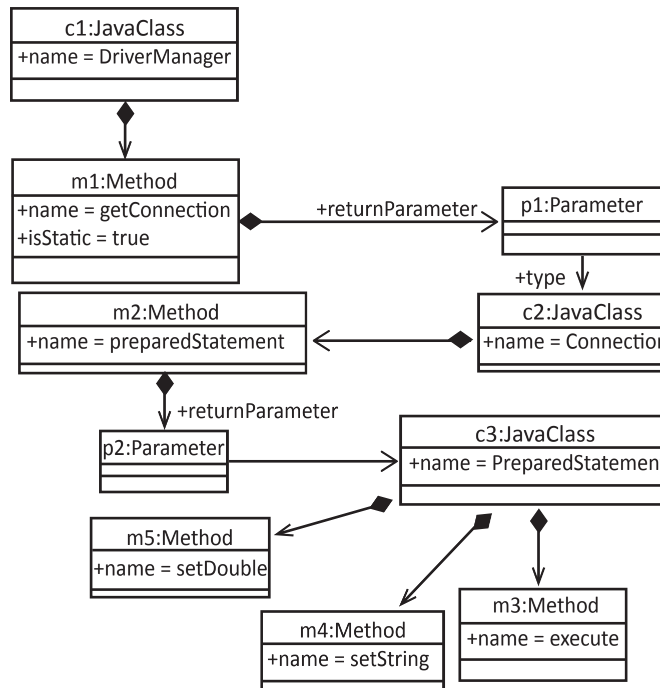

Figure 2. Part of the Java® meta-model instance used by the *saveUser* operation.

```
getConnection( ) (
  pre= ()
  post= :Connection
  }
     preparedStatement() {
      pre= con:Connection
      post= :PreparedStatement
     }
           execute() {
            pre= :PreparedStatement
            post= :objects u { :Object}
           }
```

Figure 3. Pre- and post-condition specs for the *saveUser* method.

Figure 3 is the base to generate the following Java® code:

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

The general process followed in order to generate this code was:

- To identify the *execute* operation as the one for saving elements in the database.
- To outline the fact that the *execute* operation is non-static, and, consequently, it has (as a pre-condition) an object creation of the class *PreparedStatement*. In addition, we must outline that *preparedStatement* operation has, as a post-condition, the creation of a *PreparedStatement* object.

- To note that *preparedStatement* operation is non-static, and it has, as a pre-condition, an object creation of the class *Connection*. Again, we have to note that the *getConnection* operation has, as a post-condition, the creation of a *Connection* object.
- Finally, to evidence that *getConnection* operation is static, so it does not require any pre-condition.

If we compare the suggested code with the one defined earlier, we discover some missing facts. For example, the *setString* operation belonging to the *PreparedStatement* class is not generated by the described process.

To summarize, we can establish that an O' operation subset can be created from an O operation set—and the O set has a post-condition named P. All the operations belonging to O' subset are linked by means of pre- and post-conditions, in such a way that is possible to reach a P post-condition. However, as we already present, the O' subset does not include all the needed operations to reach the post-condition P. As a consequence, the number of relationships among Java® operations might not be enough to generate the entire code, and we need to supply additional information to the process.

Adding pre- and post-condition associations to the Java® meta-model

The process of code generation, when manually made, is full of previous knowledge assertions—on the part of the designer—about the encoding platform. These assertions are commonly found by the designer in the platform documentation, using the revision of the examples included on this documentation.

The presence of these assertions makes difficult to automate the encoding process. In other words, if we want to automatically generate code from models, we will perhaps need to increase the contents of the meta-model with some of the designer's previous knowledge. For example, in the *saveUser* method that we describe, the designer knows that the use of the *execute* method, from the *PreparedStatement* class, needs as a pre-condition the results of a setX() operation, where X can be *String*, *Double*, and *Int*, among others.

Figure 4 shows the modifications proposed for the Java® meta-model. In order to make explicit all the possible relationships among operations, we add pre- and post-condition associations to the *Method* class and the *PreconditionGroup* class. We also add the following OCL restriction, as a way to avoid the fact of a method being itself pre- or post-condition.

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

Pre- and post-condition associations are used by the designer to store conditions that are not explicitly defined in the Java® platform structure. In addition, *PreconditionGroup* class is used to store groups of pre-conditions needed to initiate a method. Figure 5 shows a portion of the refined Java® meta-model instance (the entire meta-model is illustrated in Figure 2). In Figure 5, we see the *PreconditionGroup* package belonging to *m3* method of *c3* JavaClass. In this case, *m3* method can be initiated by the result of either *m5* method or *m4* method.

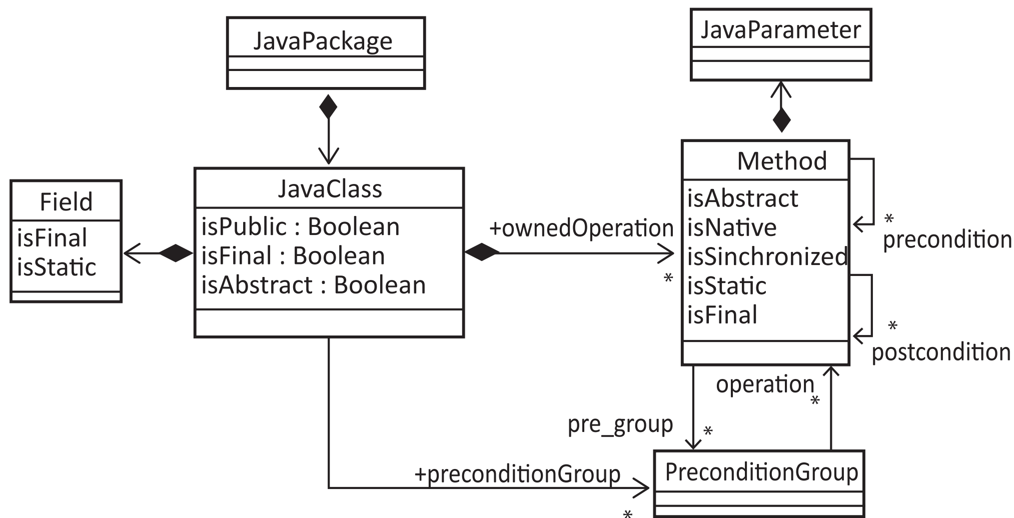

Figure 4. Modifications proposed to the Java® meta-model.

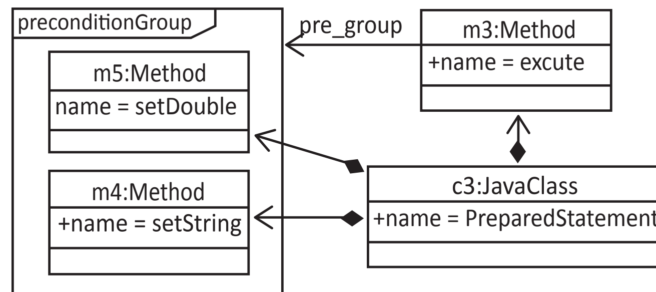

Figure 5. A portion of the Java® meta-model instance of Figure 2.

Relationship between user models and the Java® meta-model platform

As a final step in the code generation process, we propose to establish a relationship between the user model and the Java® meta-model instance. In this stage, a profile is provided in order to associate the user specs to the several kinds of database management specs. This profile can be generalized to meet another kind of specs, and figure 6 shows how it looks. In this figure, a Specification has pre- and post-conditions, represented by means of the attributes *pre* and *post*.

The *DBSpecification* class is the generalization of all possible user specs, in terms of database management, and includes the database (represented by the *DB* attribute) and the Power Set (represented by the *P(X)* attribute). This class can be specialized into *DDLSpecification* class (to modify the database) and *DCLSpecification* class (to query the database). Finally, *DDLSpecification* class can be specialized into three classes: *UpdateSpecification*, *InsertSpecification*, and *DeleteSpecification*.

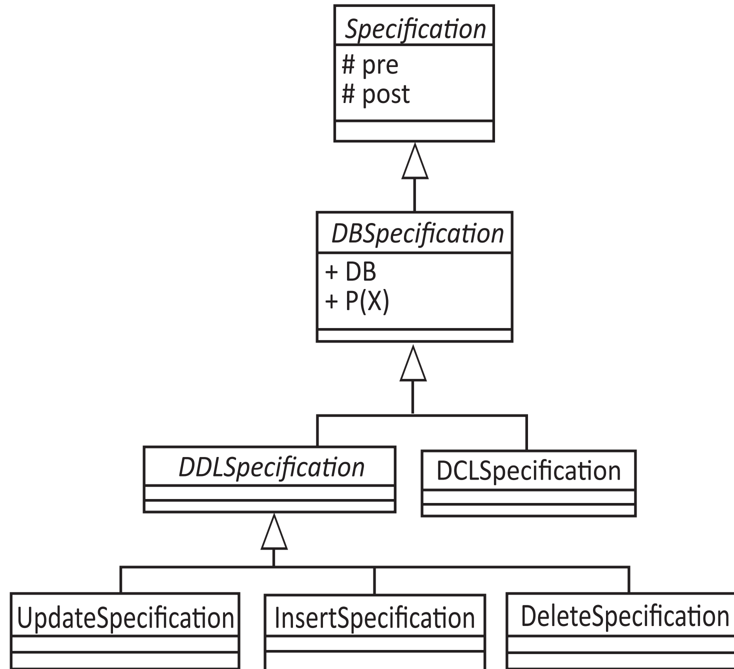

Figure 6. A *profile* to associate user specs to database management specs.

The aforementioned profile can be handled using UML stereotypes, a special feature of UML. Figure 7 shows the *ManageUser* class, belonging to the user model, which includes the *saveUser* operation. This figure also shows the *java.sql.PreparedStatement* class, which includes the *execute* method.

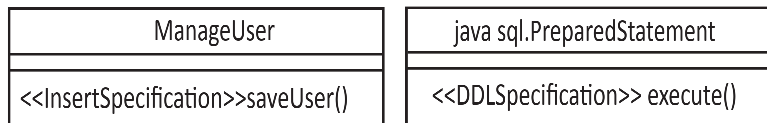

Figure 7. UML stereotypes of the operations belonging to two classes.

Figure 8 shows the relationship between the user model and the Java® meta-model instance. Due to the fact that *InsertSpecification* class is invoked by the *o1* operation, and simultaneously belongs to the specialization of the *DDLSpecification* class, we have to select this class to represent the semantics of the *saveUser* operation.

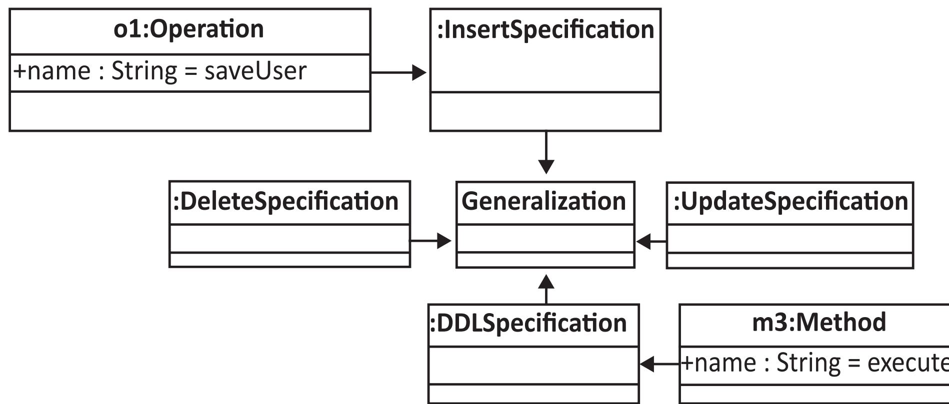

Figure 8. A meta-model instance to explain the relationship between user model and Java® meta-model instance.

A case study

Figure 9 shows two classes of a user management class diagram. In this figure, *User* and *UserManager* classes have their components stereotyped, according to the profile provided before. *UserManager* class has two operations: *saveUser*, the previously discussed stereotyped operation, and *getUser*, which has a stereotype intended to retrieve a user from database by means of his/her user identification. *User* class has only one stereotype belonging to the name of the class (<<Entity>> in figure 9.)

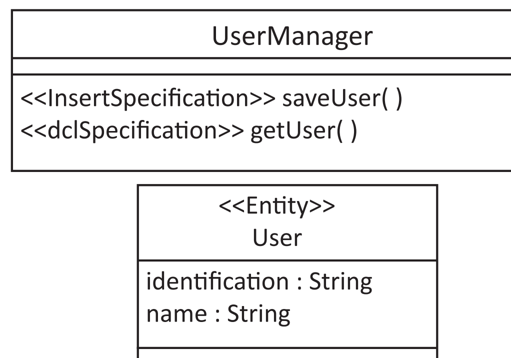

Figure 9. Two classes of the user management class diagram.

Following is the step-by-step encoding process of the *getUser* operation. The code is completed as its components are being found. Figure 10 shows an instance of the Java® meta-model with the pre- and post-conditions of the *executeQuery* method. The *DCLSpecification* of this method is also included.

As we can see in figure 10, the *executeQuery* method is associated with *DCLSpecification* object, which has the same semantics that the *getUser* operation of the *UserManager* class. Furthermore, *executeQuery* method has a *ResultSet* object as a post-condition and a *PreparedStatement* object as a pre-condition (because is non-static). Consequently, we obtain the following resulting code:

ResultSet resultSet = preparedStatement.executeQuery();

Objects of the *PreparedStatement* class are created by means of the *preparedStatement* method of the *Connection* class, as discussed earlier. This method requi-

res (because it is non-static) the creation of an object belonging to the *Connection* class. At this stage, the code of the *getUser* method is:

PreparedStatement preparedStatement = connection.preparedStatement();

ResultSet resultSet = preparedStatement.executeQuery();
getConnection method is non-static (consequently, it has no pre-conditions) and belongs to the *DriverManager* class. This method also returns a *Connection* type object. After this analysis, the code of the *getUser* method is:

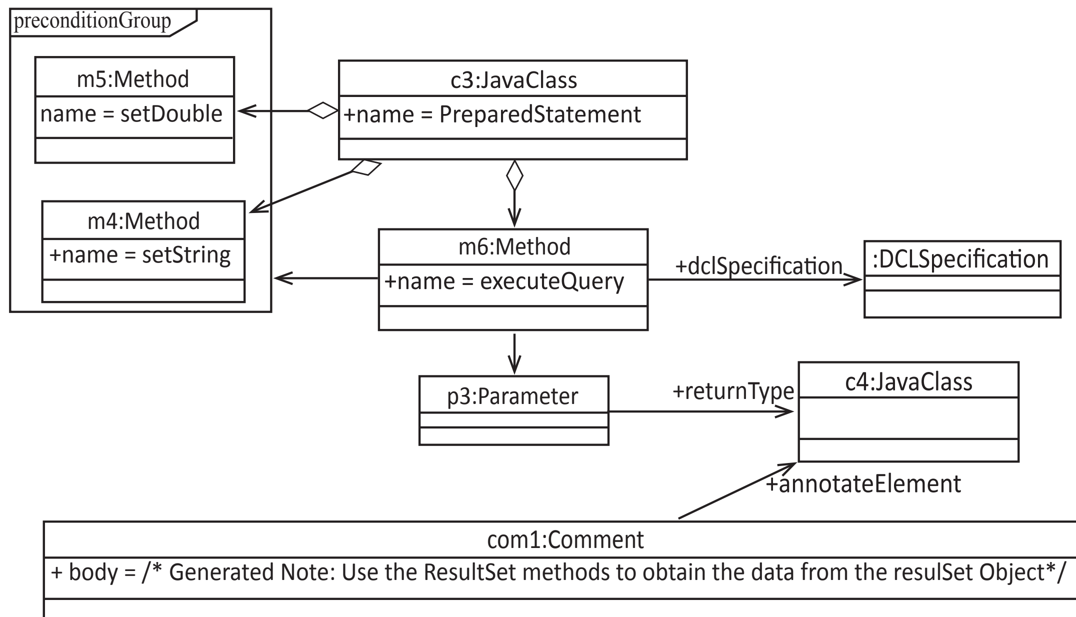

Figure 10. Java® meta-model instance of the *executeQuery* method.

public User getUser(User u){
Connection connection = DriverManager.getConnection();
PreparedStatement preparedStatement = connection.preparedStatement();
ResultSet resultSet = preparedStatement.executeQuery();
}

This is the final result of our proposal, but it is incomplete. A message is added to the code in order to warn the designer about the need for complement the method code. The source of this message is the object *com1* in Figure 10. The final code of the *getUser* method is:

public User getUser(User u){
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*
**/*
}

We can expect—from the aforementioned code—that the *getUser* method can get a record from the database, but it is not stated that this record is a *User* object. The expected code of the *getUser* method will be:

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;
```

As we previously mentioned, the designer must complete the code to have it ready to run. Before, we explored the connection between the *resultSet* method and the *setX* method, and we believe that this gap can be bridged by means of a careful analysis of the API Java® documentation. The automation of the rest of the proposal still needs some additional analysis.

Conclusions and future work

We discussed an approach to automate the code generation process by means of pre- and post-conditions of the user's model and the addition of some elements to the Java® meta-model (specifically, pre- and post-condition associations and the *PreconditionGroup* class). We also used UML stereotypes to define the semantics of the user class method. The mentioned approach has shown that it can be useful to generate the body of the methods, one of the never-accomplished promises of CASE tools.

There is some work to be done, as a way to improve the approach discussed:

- To implement a method to select the right semantics of the method, in situations where there is a "many-to-many" relationship between the stereotype of the user model and the stereotype suggested by the Java® meta-model instance.
- To extend the suggested approach to non-database management operations.
- To generalize this approach to other programming platforms.
- To incorporate this approach into a CASE tool.

References

Kepple, A., Warmer, J. & Bast, W. (2003). MDA Explained, *The Model Driven Architecture: Practice and Promise*. Indianapolis: Addison-Wesley.

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

Mammar, A. & Laleau, R. (2006). From a B formal specification to an executable code: applicational to the relational database domain. *Information and Software Technology*, Vol. 48, No. 4, pp. 253-279.

Morgan, C. (1998). *Programming from Specifications,* 2nd Edition. Hampstead: Prentice Hall International.

NetBeans Platform. "Code Generator Integration Tutorial". http://platform.netbeans.org/tutorials/nbm-code-generator.html [Consulted July 11th, 2011].

Niere, J. & Zündorf, A. (1999). Using Fujaba for the Development of Production Control Systems. *Lecture Notes in Computer Science*, Vol. 1779, pp. 181–191.

Poseidon®. http://www.gentleware.com [Consulted July 11th, 2011].

Quatrani, T. (2000). *Visual Modeling with Rational Rose 2000 and UML*. Reading: Addison-Wesley.

Robbins, J. E., Hilbert, D. M., & Redmiles, D. F. (1997). *Argo: A Design Environment for Evolving Software Architectures*. Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Boston, USA, pp. 600–601.

Together, Borland Software Corporation. "Borland Together Architect". http://www.borland.com/us/products/together/index.html [Consulted July 11th, 2011].