Automatic generation of fully-executable code from the Domain tier of UML diagrams

Macario Polo, Agustín Mayoral,
Juan Ángel Gómez and Mario Piattini
Alarcos Group - Department of Computer Science
University of Castilla-La Mancha
Ronda de Calatrava, 5
13071-Ciudad Real (Spain)
Tel.: +34-926-295300; mpolo@inf-cr.uclm.es

Abstract.

This position paper presents a method to build fully Java executable applications with three-tier architecture from the design of the Domain tier of an object-oriented system. The database in the persistence tier is built using the "one table, one class" pattern, and Domain objects accede the database through a database broker. To generate the presentation layer, a window is built for each class, as well as some additional windows that allow the easy navigation across classes at this tier. The generated code has very high cohesion and low coupling due to the use of the Reflection API defined in Java.

We also present the first prototype of a tool which does this code generation from UML diagrams made with Rational Rose.

1. Introduction.

Several design patterns can be used to build the architecture of a system. One of them is the "one table, one class" pattern, which advices to implement a class per each table in the database to be managed. This pattern is valid if we start the building of the system from the database schema; however, we can start from the design of the Domain layer and, then, to generate both the set of tables in the database and a set of windows in the Presentation Layer.

Through the use of the "one table, one class" pattern, each class at the Domain tier must have the implementation of all the methods related to the persistence of its objects (at least *Insert*, *Delete*, *Update* and a *constructor* that builds objects from the information saved in its respective table of the database). Other patterns that take the "one table, one class" pattern as a basis, advice to assign persistence responsibilities to other classes, as the "Pure fabrication" or the "Template method pattern", in order to reach high cohesion and low coupling. With the first one, each persistent class at the Domain tier should have an associated class which takes the responsibility of executing persistence methods; with the second one, persistent classes in the domain tier inherit from a superclass which implements all common methods and enumerates (as abstract methods) the operations whose implementation depends on the class.

Typically, the four methods afore mentioned are implemented in each Domain class, because its implementation depends very much of the structure of the class (i.e.: the *Insert* statement to add a *Person* to the database is very different than the same statement to add a *Car*). However, the structure of all these methods is quite similar, since all of them are built depending on the fields of

the class and on the values of their fields. For example, the SQL statement to introduce any new record in the database from an existing object is the following one:

Insert into table (column1, column2, …, column) values (fieldValue1, fieldValue2, … , fieldValueN)

Thanks to its Reflection API, Java allows to accede in runtime to a lot of data about an instanced object, as the name of its class and the names, types and values of its fields. Therefore, it is possible to generate at runtime any SQL instruction for an object through the access to their members:

- The table can be recovered from the name of the object's class
- Columns can be recovered from the names of the fields
- Values of the object's fields can be recovered via the *Object Field::get()* method of the Reflection API.
- Values can be assigned to the object's fields (for constructing an object from a row in a table) using the *Field::set(Object, Value)* method, also part of the Reflection API.

If we had a superclass that implements all persistent methods and from which all Domain classes with persistence responsibilities inherit, then none persistence method should be implemented in Domain classes. In this case, we would be using a "Template Method" pattern with all the methods implemented, and none "Pure Fabrication" class would be needed. Therefore, a Domain class would become totally operative (from the persistence point of view) simply writing the definition of its fields (since the methods related to the problem domain should be implemented in the superclass). Such a superclass is CPI [1] which can be freely downloaded from http://www.inf-cr.uclm.es/www/mpolo/easytest, jointly with a practical application of its use (a tool to the automatic generation and correction of test exams).

2. Generating the Domain-tier code.

Starting from the UML class diagram which represents the Domain tier of an object-oriented system, the code of the corresponding Java classes can be easily generated, only writing the enumeration of their attributes and two constructors: one for constructing an empty object and another one for constructing an object from a row saved in a table of the database. The implementation of this latest constructor simply consists of a call to the parent's constructor, passing it the parameters that are used to build the *Where* clause (usually the values of the primary keys of the object to be constructed) of the *Select* statement. The first constructor does not take parameters and also consists in a call to the parent's constructor with no parameters: this one takes each field of the Domain object and assigns them a default value which depends on its type (for example: *new String()* if the field is an *String*, *0* if it is a *long*, etc.). The following is an example of how to build the *Person* object with 13203881 as SSN:

public static void main(String args[]) { *String SSN[]={"13203881"};* *Person p=new Person(SSN);* *…..* *}*	*class Person extends Yet {* *String mSSN, mName;* *long mAge;* *Person() { super(); }* *Person(String arguments[]) {* *super(arguments);* *}*

Table 1. Example of the use of the Java code generated.

The right side in Table 1 shows all the code that must be generated to build any persistent class: only the list of attributes and two constructors. The Yet superclass is in charge of doing the connection to the database via a Broker: in this manner, we avoid any possible direct coupling of the Domain classes with the database.

3. Generating Presentation-tier code.

The user interacts with Domain objects through classes at the Presentation tier. We always generate the set of screens shown in Figure 1:

- A main window with two functionalities (element "1" in Figure 1): to create a new object of any of the Domain classes and to see all the objects of a class saved in its corresponding table of the database.
- A window to create, show and manipulate (update, modify and printing) Domain objects (elements "2" in Figure 1).
- A window to show all the rows in a table of the database, which are objects of its corresponding class (element "3" in Figure 1).

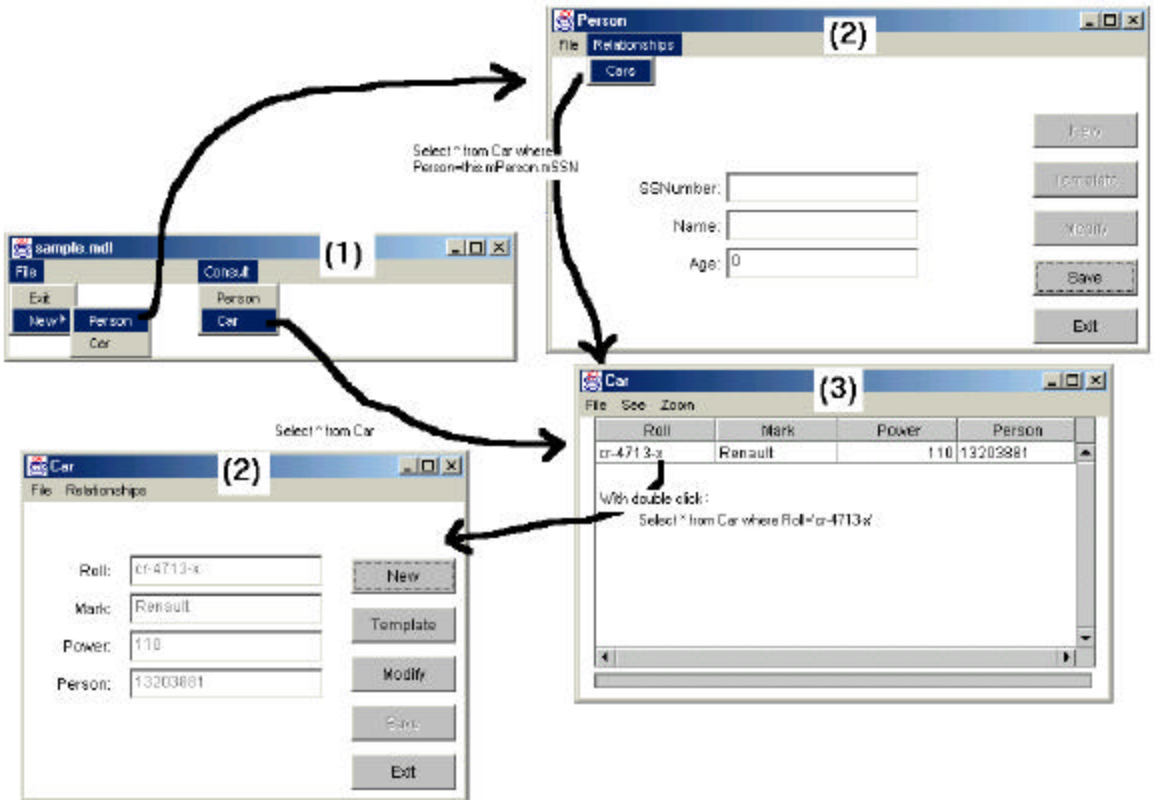


Figure 1. Navigation across windows.

4. A tool to generate the code.

Figure 2 shows the main (and only at this moment) window of RR2Java, a tool which generates all the code explained in the previous section, as well as the database broker and a file with some constants needed to get the adequate performance of the system. It takes as input a Rational Rose model. The user selects the classes for which he/she desires to generate code, and the tool produces the Presentation and the Domain tiers.

Currently, we have a single prototype of RR2Java developed in Microsoft Visual J++, because with this language there exists direct access to the Rational Rose COM component. Future versions of RR2Java will be developed in pure Java and will access the Rational Rose component using a Corba interface.

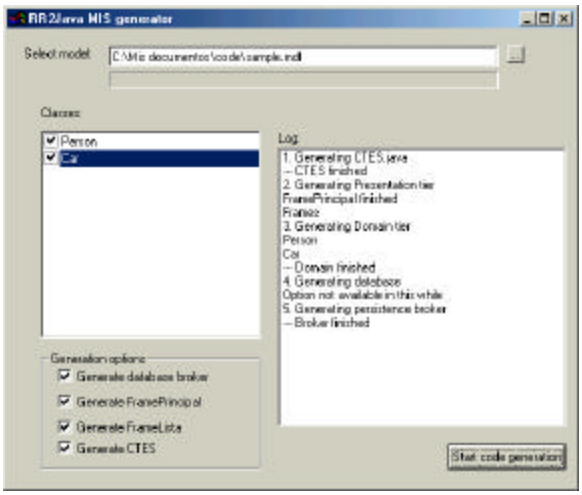
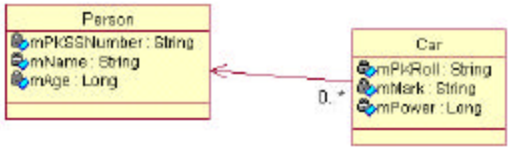
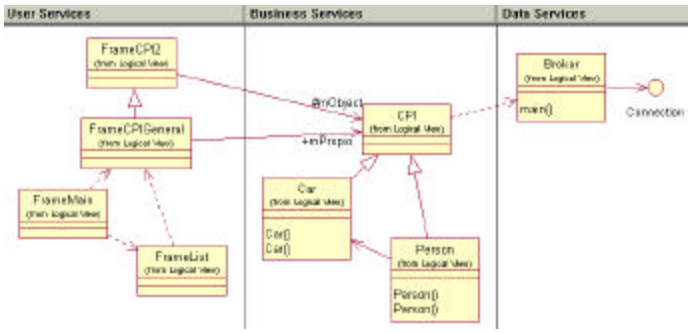


Figure 2. Main screen of the tool for generating code.

Figure 3 shows the original system and the generated one. FrameCPI2 is an abstract window which knows an abstract CPI object, which is mapped to a concrete Domain object in runtime. FrameCPI2 defines and implements many of the common methods to windows that correspond to objects. These are built using metadata at runtime with the FrameCPIGeneral windows, which is a frame class whose widgets are loaded at runtime depending on the fields of the object that it must show. All the persistence operations are implemented using metadata in the CPI class, and Domain classes (Car and Person) have only the two constructors mentioned in Section 2.



original design



generated system (obtained through reverse-engineering of the generated code)

Figure 3. Original and generated system.

Really, given any on of the three tiers of a three-tier design, the other two can be automatically generated. We are developing also another tool to generate Java code from the schema of a relational database.

5. References.

[1] Polo, M., Piattini, M. and Ruiz, F. (2001). *Yet: Yet another pattern for mapping tables and objects*. Submitted to EuroPLOP 2001.