An Application Generator Based on UML Specification

KREŠIMIR FERTALJ, MARIO BRČIĆ
Department of Applied Computing
University of Zagreb, Faculty of Electrical Engineering and Computing
Unska 3, Zagreb 10000
CROATIA

Abstract: - This paper presents a proprietary application generator based on UML specification. The tool is designed for generating the source code in various programming languages from the same specification. The main characteristics of the existent tools are explained in brief. Main generator capabilities and merits are presented as well as an example of usage based on a relatively simple scenario.

Key-Words: - UML, CASE, application generator, source code templates

1 Introduction

At present and in the future, the technology development is accompanied by an increase in applications' complexity. Code generators are used to increase code quality and decrease development time, since their goal is to generate repetitive source code while maintaining a high consistency level of the generated program code.

Code generation assumes the mission of writing repetitive code parts, leaving to programmers more time to concentrate on specific code. The generators provide more productivity; generate great volumes of code, which would take much longer if coded manually. Consistent code quality is preserved throughout the entire generated part of a project. Required coding conventions are consistently applied, unlike handwritten code, where the quality is subject to variation. In case of finding errors in generated code, the errors can be corrected in short time through revising of templates and re-running the process of code generation [1].

Code generators are delivered with limited set of solutions for common problems in a target domain and allow only limited possibility for extension.

Some tools generate only parts of applications while the others generate whole applications. Code generators are especially suited for database-founded applications where large number of forms with similar functionality are needed.

The source code generator presented in this paper is based on UML specifications and on templates written in XML/XSL. UML specifications are greatly enriched with calls to parameterized snippets whose implementation is delegated to the templates while they are carrying semantic description of the model's requisites. The generator is relying on an existing UML tool for delivery of UML capabilities and on its extendible architecture [2]. The most important characteristic of the generator is the preserved flexibility towards the target programming language, accomplished by code generation through two transformations; first into an intermediate code and then into the code of a selected target language. Since the complexity of UML model can vary from simple to highly complex, the tool provides wizards for creation of the most common complex model parts based on input settings.

2 Existing commercial tools

Many commercial products of different applications and approaches to generation are available on the market. In this paper, the categorization based on inputs and outputs [1] is used.

In the first category, *code mungers*, there are many tools. Graphic languages, such as UML, can be used as input language. Most of UML based generators do not have their own UML development environment. Instead, they use UML specifications made in other tools as input in the form of XMI or some other interchangeable format. Such working mode, although exceptionally flexible, can face the problem in extraction of all the data from specification due to different UML tools' particularities and varieties. On the other hand, the UML development environment can be a better option because it offers improved control over the whole process and avoids compatibility issues between the specification and the code generator. Again, the code generating functionality, configurability, flexibility and extendibility are generally less extensive than those of the aforementioned {code mungers without UML IDE}. *Sybase PowerDesigner* is an example for such a tool. Tools with specifications in non-graphic languages also belong to this category. Their disadvantage, besides the use of non-graphic language for specification definition, is the reduced control over specification deriving from smaller manageability and intuitivity. *MyGeneration* is an example for such tool.

Inline code expanders have proven to be rather efficient in web applications with expanded code written in a server-side script language. While these tools can be efficient in their limited application area, they are less extendible than the first category, since the mere choice of an expandable language reduces our possibilities. An example of inline code expander is *Iron Speed Designer* which expands HTML code with ASP tags and code-behind files.

Tool categorization as a partial class generator or a tier generator depends on its templates. Tools in this category are template –based and flexible. They rarely provide a graphical language for specification definition and instead they rely on database metadata and tabular metadata inputs, making them non-intuitive and awkward. *MyGeneration* and *CSLA.NET* demonstrate these characteristics.

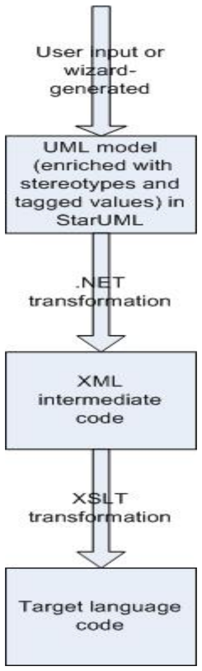

User input or wizard-generated
UML model (enriched with stereotypes and tagged values) in StarUML
.NET transformation
XML intermediate code
XSLT transformation
Target language code

Fig.1 Main working principle

3 The main principle

The main idea of the tool presented in this paper is code generation based on UML specifications, where specifications are expected to be as rich as possible and elastic with regard to the target language. Model descriptions can be target language dependant or target language independent. Target language independent descriptions are stored in attributes defined in shared profiles while target language dependant ones are saved in attributes defined in profiles specific for the target language.

Code generating is conducted through two chained transformations. The first transformation is similar to UML model XMI serializer [3] with the difference that the intermediate code file is generated for each model element. This model element is then defined as a separate-file entity. All data stored in the UML model are rewritten in the form of an XML file of predefined format, called platform independent code (PIC). Templates for the first transformation are independent and invariant in respect to the target language.

A second transformation follows. It is accomplished by using the XSLT processor and modularly written XSLT templates for each target language. The input in this transformation is the PIC file and the templates applied to it. The output of XSLT processor is the target language source code file as the result of template's specifications. The task of *code munging* is performed in the final phase as the PIC is being transformed into the target language via XSLT templates.

Considering that the two transformations are concatenated, where the result of the first transformation is the input to the second, it can be formulated that a transformation pipeline has been established.

4 UML specification

The tool in this paper is based on UML system description. Model complexity can vary from simple to highly complex with rich descriptions by means of *stereotypes* and *tagged values*. The generator relies on existing *StarUML* tool [2] for manipulation over UML specifications through its open Application Programming Interface (*API*).

4.1 Expressing actions

Actions can be expressed in the target language code, as in the case of *PowerDesigner*, but the preferred way is through a platform independent language in the form of snippet calls. An element's actions are specified by hand-coding in tagged value ***BodyPICFragment,*** which expects the intermediate code in XML format.

Snippets participate as model parameters, semantically required to realize action, while the details of realization are delegated to the snippet's realization in the target language. This approach has shown to be most effective as it is a high-level description of an action, leaving enough freedom for the optimal implementation on the target platform. If it were using a lower level to describe an action, such a description would be too closely bound to a specific platform and it would reduce the specification portability.

The next fragment presents ***InsertIntoSelectedTableForm*** snippet call within a method:

```
<cdgn:InsertIntoSelectedTableForm>
  <!--Calling insert form for selected table-->
<cdgn:ParamIndexes>
    <cdgn:ParamIndex Value="0" Ordinal="0" />
</cdgn:ParamIndexes>
</cdgn:InsertIntoSelectedTableForm>
```

From the example, it is evident that the object parameter's indices are the parameters to snippets. Indices relate to parameters bound to a model's element (in this case operation), and they are assigned to it through the ***SnippsParams*** tagged value in the UML specification. The tagged values make a collection of the model's elements required for all snippet calls from that object. Now the parameter indices in the collection are the parameters supplied to calls of snippets. The intermediate code where the object's parameters have been specified is given in the following fragment:

```
<cdgn:SnippsParams>
  <cdgn:Element Name="SelectForm.tableCB"
Path="::Design
Model::proj2::Controls::SelectForm.tableCB"
Stereotype="ControlInstance" Ordinal="0" />
  </cdgn:SnippsParams>
```

In the case of an illustrative snippet generation into C#, we get the following code:

```
  Type form = Type.GetType("proj2.Promjenaproj2_"
+ tableCB.SelectedValue.ToString());
  object forma= Activator.CreateInstance(form,
bind.DataSource);
  MethodInfo method = form.GetMethod("Show", new
Type[0],null);
  method.Invoke(forma, null);
```

From the presented code, it becomes obvious that the snippet implementation in C# relies on .NET platform specific features. If it were for some other platform, the solution could turn quite different. If a lower level specification of actions were used, it would be detrimental to the platform independency, because the formulations of solution to the same problem, can differ in basic concepts due to different platforms.

If the action code were intended to be written in a specific target language, the code should be placed inside XML tags specifying that language.

The C# example is given in the following fragment:

```
<cdgn:TargetCode Language="Cs">
…
C# code
…
</cdgn:TargetCode>
```

5 Intermediate code

The first step of generation process is the PIC generation. PIC is the code notation comparable to pseudo-code. It is a set of XML directions for transformations to generate the final target code.

PIC is a hybrid of:

- XMI-like form, giving the description of pertinent UML specification in XML.
- Intermediate code of the programming language, because UML component descriptions contain coded snippets' calls and the target language code fragments.

The code level is variable. In some occasions, it can be low, resembling to the target language due to general characteristics of the object-oriented languages. However, the level can be high when implementation details are delegated to a target language prone to optimization. In all the cases all metadata have to be supplied.

6 Templates

The templates are written in XSLT/XML. Their task is the transformation of intermediate code into the target language code. Currently, only the templates for C# and MSSQL have been produced. The templates for other languages can be written easily. The requirement on a template is to be stored inside its own subfolder of the generator's folder. The subfolder name must match the pattern: ***<LanguageName>Templates*** (e.g. the existing subfolders are: *CsTemplates, MSSQLTemplates*). The starting point for the second transformation is **basic.xslt**, unless stated otherwise via specified tagged values.

Template folders also contain XML files with data type mapping and configuration data.

6.1 Metadata

The user can define her/his own metadata for each transformation with the only constraint that it must be in XML form. Metadata for a single template must be enlisted in parameter file, which contains all the inputs to the template and also a list of locations for other pertinent metadata files. Location of the parameter file is supplied to the relevant element through the **ParameterFile** tagged value. The location of a special starting point template can be supplied to the element, if it is not the standard **basic.xslt**.

7 Preservation of user added code

Preserving of the code added by a user is achieved by using special, for that purpose intended regions in the code. Existence of the regions is specified already in UML specification by tagged values tied to elements of the model. Regions can be the following:

- ***BeforeNamespace***
- ***BeforeCode***
- ***NoStartCode***
- ***EndCode***
- ***AfterCode***

A region is defined by its beginning and end, and the code within it becomes secured from possible future erasure by the generator since the code had been saved. Boundaries are marked by specially formatted comments for the beginning and for the end, formulated as follows:

startGenComment-elementGUID-regionName-{B** for region start* **|** **E** *for region end* ***}-endGenComment

The elements written in italics are changeable.
The semantics of the changeable elements:

- **startGenComment** – denotes the beginning of a comment in selected language. Preferably the comment should span a row, but if the target language does not support it then it serves as a simple designation for the comment begin.
- **elementGUID_**– every element in StarUML has its own GUID, a unique identifier that univocally ties each element to its regions.
- **regionName** – is one of the following: *BeforeNamespace, BeforeCode, StartCode, EndCode, AfterCode*
- **endGenComment –** signs the end of the comment. This element is optional since many languages provide comments that span the row.

Example of a region in C#:

```
  //ZIRgen-kesIeuSudU+DKDixbzkXkQAA-
StartCode-B
  //ZIRgen-kesIeuSudU+DKDixbzkXkQAA-
StartCode-E
```

When a code is generated anew in the same language, the generator extracts those regions and inserts them in the newly generated code. Currently, user added code preservation has been achieved only for C#, and only for the elements of the model that cause the generation of just a single file in the target language.

8 Wizards

UML models can become very complex when it comes to describing details of the parts of the system, demanding a lot of metadata in the form of marked values. These metadata usually have to adhere to certain rules, therefore demanding that the user knows the elements of the profile. Due to this complexity, the tool includes wizards that use the input settings to generate complex models for often-used concepts, such as forms of user interface and business objects.

The following are the wizards offered:

- *DBReverse* – wizard for reverse engineering of database
- *DBAccess* – wizard for systems used for data manipulation; input, alternation and erasing data in the database
- *UIDesigner* – wizard for user interface. Definitions of the user interface are generated based on definition in the interface designer
- *DBConceptTransform* – wizard for transformation of conceptual model into the physical model.

9 Example of usage

In this section, an example of using the tools is shown. The example of usage shows the construction of a complex application with minimal effort due to usage of the wizard, although the same could have been done by manual designing. Resulting specification is available for manual changes. With minimal additional adjusting, the project can be generated in languages for which patterns had been written.

For UML specifications, StarUML tool is used and the generator is connected via an open API. The generated specification has to be located within the model ”Design model“ of the UML specification.

9.1 Creation of the database model

The database model can be created from scratch, or it can be created by reverse engineering of the existing database using the ***DBReverse*** wizard. In our case, we start with conceptual description of a completely new database.

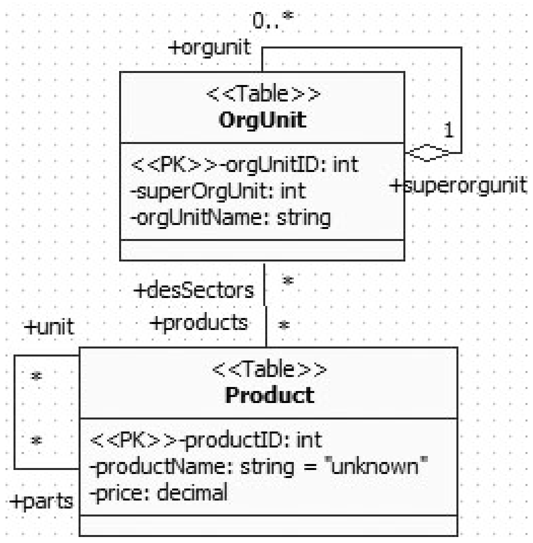


Fig.1 Conceptual database description

After having been conceptually described, the database should be transformed into the relational scheme, suitable for generating. It does not include N:N relationships or associative classes, because they are transformed into simpler forms. Conversion from the conceptual to relational form is done using the ***DBConceptTransform*** wizard.

Tables of the UML class are stereotypified by „Table". In this phase the profile ***ZIRgenDB,*** that contains all the platform-independent elements of the description of the database, is the mostly used. It is possible to use the platform-dependent profiles with additional descriptions of the database.

At the end of this phase, the layer of data storage is ready to be generated in some of the languages of the database management systems for which patterns are avalable.

9.2 Constructing the database founded application model

When constructing a model of multi-layer application, the goal is to create an application of similar functionality as offered by Iron Speed Designer and MyGeneration with basic patterns. UML model of such an application is extremely complex with a lot of data and extensive usage of different profiles provided. In order to manually construct the model, the abilities of the generator should be well known, as should be the profiles that contain instruments for expression of the necessary concepts.

In this case, the complexity and great demands on the programmer are bridged by DBAcess wizard that creates entire aforementioned architecture, starting with the layer for accessing data through stored procedures and business objects to user interface.

When starting the wizard, one of the databases from current specification is selected, and then the tables whose data are to be manipulated are chosen from it via interface. Elements of the layers for data access and business logic for all tables are then created as specifications and user interface is created only for the selected tables.

The new user interface can be accessed with ***UIDesigner*** that enables graphical editing of the form; adding of new controls, defining of their features.

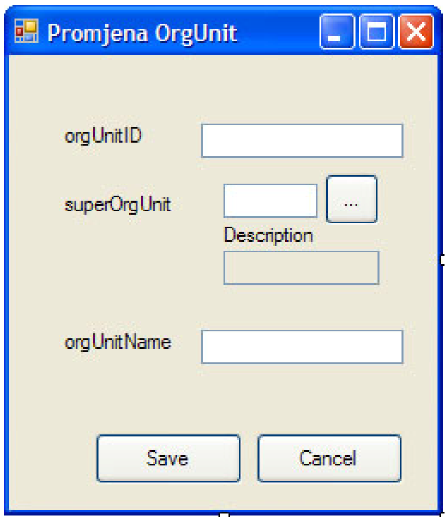
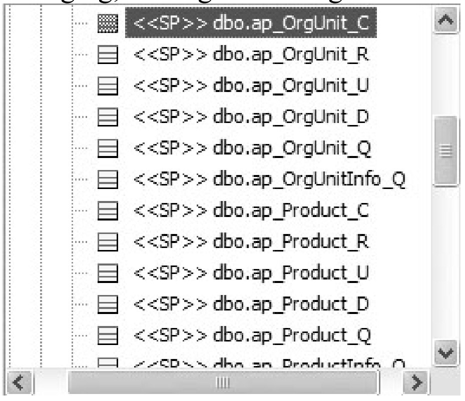


Fig.2 Generated UI for OrgUnit table

ZIRgen and ***ZIRgenUI*** are the most used profiles, both with platform independent features. ***ZIRgen*** profile features basic characteristics of the generator, while ***ZIRgenUI*** features the characteristics needed to describe the user interface. It is possible to use platform dependant profiles for more precise specification in wanted platforms.

The next layer to be generated from this part of the model should access the data within CRUDQ stored procedures that handle direct work with tables: insert, reading, changing, erasing and listing.



```
<<SP>> dbo.ap_OrgUnit_C
<<SP>> dbo.ap_OrgUnit_R
<<SP>> dbo.ap_OrgUnit_U
<<SP>> dbo.ap_OrgUnit_D
<<SP>> dbo.ap_OrgUnit_Q
<<SP>> dbo.ap_OrgUnitInfo_Q
<<SP>> dbo.ap_Product_C
<<SP>> dbo.ap_Product_R
<<SP>> dbo.ap_Product_U
<<SP>> dbo.ap_Product_D
<<SP>> dbo.ap_Product_Q
<<SP>> dbo.ap_ProductInfo_Q
```

Fig.3 CRUDQ stored procedures in UML specification

The next paragraph shows a PIC code fragment aimed for data inserting via stored procedures:

```
  <cdgn:BodyFrag
xmlns:cdgn="http://www.fer.hr/ZIRgen"><cdgn:Creat
eStoredProcedure /></cdgn:BodyFrag>
```

which is further expanded with metadata in the first transformation, i.e. the creation of complete PIC system description.

The next layer is the layer of business objects generated based on some of the existing architectures written for the platform. Business objects contain a part of the layer for accessing data as well as business logic. Often, they also contain a part of functionality tied to user interface. In case of this generator, everything depends on implementation of the patterns. For C# they currently do not contain functionality that would be part of the user interface. When selecting an OO language as target, from the model of the database, basic abstract classes of business objects are generated within the space of the base name. They are created from the elements of the tables and views that hold all business objects necessary metadata. Also, as described in the model, user business object classes are generated in the UserOpen namespace. These classes inherit the base classes, expanding their basic functionality, mainly business logic, with user-added code.

Finally, the last layer of our multi-layer application is the layer of user interface and presentation. These two layers are usually joined in one, in a form such as windows forms. However, for web pages they remain separated, since the presentation is a part of the web browser. For example, in the case of Windows forms, event handling code and business object binding code are generated in whole from the user interface model elements.

When generating all the layers, except the one of business objects, the generator behaves like a layer generator. An entire layer is generated and it can function even without the user code. When generating business objects, the tool acts like a generator of partial classes. Basic classes with basic functionality are generated, while the rest of the functionality and the business logic are left for the programmer to implement within inherited classes.

10 Conclusion

Functionality of the presented generator acting from UML specification, has its advantages over typical patterns-based generation. Among advantages are robustness of the system, configurability via different, elaborate system descriptions and improved manageability. On the other hand, the shortcoming is greater complexity due to increased configurability that causes generator and patterns written for it dealing with great number of cases in order to secure consistent and functional generated code.

According to code generator categorization [1], this generator does not exclusively fit in either of the categories, but it is a hybrid, featuring characteristics of several types. Wizards that use input settings to generate UML models are passive generators. Since during generating translation is performed, in the first phase, from UML to intermediate code, and then from the intermediate code to the target language code, using patterns written in XSLT, the generator obviously features characteristics of the code-translating generator too. There is a similarity with generators of mixed code in regard of the regions intended for preserving user code. It also features characteristics of partial class generator due to the way it generates business objects. On the other hand, entire layers of user interface as well as data layer can be generated which qualifies it as layer generator too.

A very robust, powerful generator adaptable to user demands has been created; with an ability to generate in every language for which it has written patterns. However, potential users are facing a long learning process if they want to use all the abilities of the program since extensive possibilities necessarily incur complex specifications.

References:

[1] Herrington, J., *Code Generation in Action*, Manning, 2003.

[2] Lee, M., Kim, H., Kim, J., Lee, J., *StarUML 5.0 Developer Guide (PDF)*, http://staruml.sourceforge.net, 2005.

[3] Object Management Group, *MOF 2.0/XMI Mapping, Version 2.1.1*, http://www.omg.org/docs/formal/07-12-02.pdf, 2007.

[4] Dollard, K., *Code Generation in Microsoft .NET*, Apress, 2004.