

A Source Code Generator Based on UML Specification

Kresimir Fertalj, Mario Brcic

Abstract — This paper presents a application generator based on UML specification. The tool is capable of 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. The tool extensibility is described as a mean of making the tool to suit a wide range of needs.

Keywords — application generator, CASE, source code templates, UML, XML/XSL transformations

I. 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 [5].

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

This work was supported by grants # 036-0361983-3137 and # 036-0361983-2022 by the Croatian Ministry of science, education and sports.

K. Fertalj is with the Department of Applied Computing at the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia (phone: 385-1-6129918; fax: 385-1-6129915; e-mail: kresimir.fertalj@fer.hr).

M. Brcic just graduated from Computer Science at the Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia (e-mail: mario.brcic@fer.hr).

Generators usually have their own Integrated Development Environments (IDE) because, for instance, a professional Web application development today is not possible without a good HTML designer, a code editor, a Web page preview and often a debugger [18].

The developers need tools that will be able to generate code for most common and most repetitive functions. As every developer has his own method of development, it is up to her/him to choose the product he will be able to work with. In an ideal situation the produced code would be the code the developer would himself make manually knowing the best practices in solving particular problems, having a good knowledge and experience in the area, and having a reasonable amount of time for development.

It is nearly impossible to imagine that an existing generator will produce such code, which the developer will never have to modify or customize. Sometimes, the effort of learning how to use the generator and to think in its terms is not justified considering the size/scope of a project. If that is the case, then using a combination of proprietary custom-built framework and third party solutions to specific problems might be a better way to proceed.

A couple application generators have resulted from the authors' research and development [6], [23]. The experience gathered was used as a foundation when modeling a new tool.

The old tool [7], [8] was used in several projects, with the generated code rate ranging from 10%-90%. Higher percentage was achieved in projects with a large number of straightforward data processing forms. The usage of the generator would be more frequent if the generator had been capable of generating custom code fragments in an easy way.

Later, another tool was built and has been used in building Web-based applications [24]. The generated systems consisted of many different applications, Web pages, and forms which were rather unique in nature. Despite their uniqueness, fractions of code were isolated and converted into XML/XSL templates. Generally, templates could contain code in any programming language. The tool included the support for code generation based on database structure, and generation of documentation. It was expandable with user defined parameter types and dependencies. The generator overcame some problems found in various commercial tools [23], [25], [26]. It was suitable for programmers who wanted complete control of the code generation process, but was still missing modeling capabilities.

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 described in this article 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.

II. COMMERCIAL SOLUTIONS

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.

Finally, one should mention the web applications separately [23]. Because of their platform independency, widespread availability, easy maintenance and other advantages [21] they are taking a large part of market once held exclusively by stand-alone applications. That process has several consequences. Web applications are becoming increasingly similar in their features and usage to stand-alone applications. Modern development tools try to unite the development of stand-alone and Web applications by introducing the use of same languages, offering the same or similar controls, thus enabling more and more developers already familiar with stand-alone application development to become Web developers. Object-oriented development which is dominant in stand-alone application development, but until recently rarely used in Web applications is now becoming increasingly popular in Web application development.

Still, web application development requires coding in different languages (e.g. a combination of HTML, CSS and an object-oriented code-behind language) that is trickier than just supporting different databases because of the difference between languages, which are far beyond syntax variations and include different approach to problem solving. Therefore, new techniques emerge such as tag-based generation of rich components [22].

III. BASIC 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.

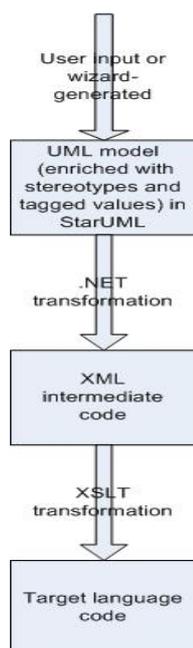


Figure 1 Main working principle

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.

IV. UML SPECIFICATION

The tool described in this paper is based on specifications modeled in UML. 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).

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

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>
```

Although the user can write action code in the specific target language, it should be done so only in particular situations because such action code reduces specification platform portability. Most of information should be stored in the platform independent part of metadata. The distinction between platform dependent and independent parts of a model is accomplished by separation into different profiles, which can be included in specification if necessary.

V. 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.

VI. CODE GENERATING

The specification must be properly designed in order for the generator to be able to transform it to the target code.

The specification is properly designed if:

- All mandatory data, such as attribute types and association role names, is supplied
- Valid input data, obeying the existing rules.

The tool provides a feature *generating in roles* used for generating more than one source code entity from the same model element. The feature was created as a response to the needs in generating business object layer, but can be used as needed in generating any part of specification. The settings for *generating in roles* can be specified in **GenParameters.xml** target language specific settings file. At the moment the preservation of manually added code is not enabled for model elements that use the pertinent feature.

VII. SECOND TRANSFORMATION 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.

A. 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**.

VIII. 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, alteration 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.

IX. EXAMPLE OF USE

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.

A. Database model creation

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.

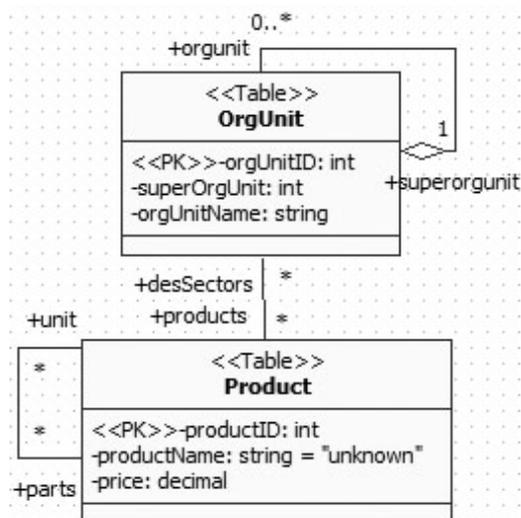


Figure 2 Conceptual database description

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 [17], but they are transformed into simpler forms. It is assumed that the data model should be brought into the third normal form [9], [10]. Conversion from the conceptual to relational form is done using the proprietary *DBConceptTransform*.

Tables of the UML class are stereotyped 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 available.

B. Database founded application model construction

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 *DBAccess* 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.

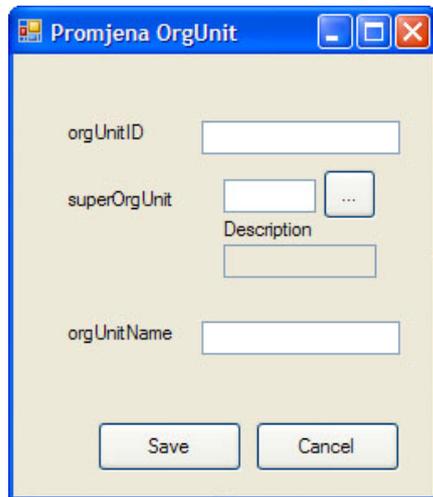


Figure 3 Generated UI for OrgUnit table

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.

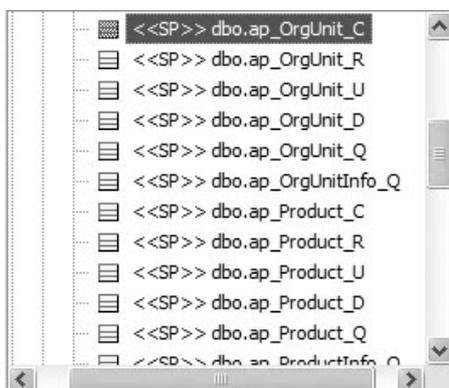


Figure 4 CRUDQ stored procedures in UML specification

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:CreateStoredProcedure />
</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.

X. PRESERVATION OF CODE MANUALLY ADDED BY USER

Programmers could tag the manually written code with some special labels or some special descriptors placed into program comments. At the time of repeated generation, a source code analysis should be performed in order to isolate the manually written code. The isolated portions of code must automatically be transferred into the newly generated code, to their proper positions. Two significant shortcomings can be noticed at the first glance. Manual coding of special comments, along with the code that should be written, lay additional stress on the

programmer, who is forced to focus on two concurrent activities – program coding and tagging. Hence, this technique is expected to be time consuming and error prone. An intelligent source code analyzer needs to be developed, which would isolate manually written code and which would insert it at the proper position in re-generated program. In fact, an additional analyzer should be developed for each of the programming languages supported by the tool.

An alternative can be to compare the manually changed files to the files produced in subsequent steps of generation. The originally devised procedure described in [8] relies on basic file comparison algorithms [11][12], [13], [14].

A good compromise to aforementioned alternatives can be preserving of the manually written code 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*
- *StartCode*
- *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

```
private OrgUnitU()
{
    //ZIRgen-uKQrNSRdk0Cy1K2jT8qXjQAA-StartCode-B
    Console.WriteLine("Manually added code");
    //ZIRgen-uKQrNSRdk0Cy1K2jT8qXjQAA-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

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# inside a method and manually added code inside it:

single file in the target language that is, not using the *generating in roles*.

After regenerating the file from the example the manually added code is extracted and stored in the PIC file as given in the next method definition fragment:

```
<cdgn:Method Name="OrgUnitU" Constructor="true" AccessModifier="private">
  <cdgn:Parameters />
  <cdgn:Body>
    <cdgn:Codezone Name="StartCode" GUID="uKQrNSRdk0Cy1K2jT8qXjQAA">
      <cdgn:TargetCode Language="Cs">
        Console.WriteLine("Manually added code");
      </cdgn:TargetCode>
    </cdgn:Codezone>
  </cdgn:Body>
</cdgn:Method>
```

Assuming no changes to the given method, the generated target language code would be identical to the one above.

XI. THE TOOL EXTENSIBILITY

The tool in this paper can be extended in a number of ways to suit the user needs. The existing tool capabilities are a good starting point for creating complex systems, but the user will always encounter situations where he will need to extend the tool in order to fulfill his vision. The first transformation is hard-coded in C#, it can be modified only in the source code.

User can extend several areas:

- Create and extend UML profiles with new prototypes and tagged values or modifying the existing ones
- Add or alter the second transformation XSLT templates for supported language

- Add new target language and create second transformation templates.

A. Creating and extending the UML profile

Creation of UML profile involves creating the XML profile definition file. In the definition file we declare new stereotypes and tagged values. With profile creation the user increases expressiveness in their specifications. New stereotypes depict entities from a problem domain while new tagged values add to quantity of specification information.

Below is given the fragment of UML profile definition defined for user interface purposes:

```
<?xml version="1.0" encoding="UTF-8"?> <PROFILE version="1.0">
...
<BODY>
  <STEREOTYPELIST>
    <STEREOTYPE>
      <NAME>Form</NAME>
      <DESCRIPTION>Indicates UI form</DESCRIPTION>
      <BASECLASSES>
        <BASECLASS>UMLClass</BASECLASS>
      </BASECLASSES>
    </STEREOTYPE>
    ...
  </STEREOTYPELIST>
  <TAGDEFINITIONSETLIST>
    <TAGDEFINITIONSET>
      <NAME>DataBinding</NAME>
      <BASECLASSES>
        <BASECLASS>UMLAttribute</BASECLASS>
      </BASECLASSES>
      <TAGDEFINITIONLIST>
        ...
        <TAGDEFINITION lock="False">
          <NAME>DescText</NAME>
          <TAGTYPE>String</TAGTYPE>
          <DEFAULTDATAVALUE>
            </DEFAULTDATAVALUE>
        </TAGDEFINITION>
        ...
      </TAGDEFINITIONLIST>
    </TAGDEFINITIONSET>
  </TAGDEFINITIONSETLIST>
</BODY>
</PROFILE>
```

In the exemplary profile the stereotype **Form** has been defined for description of user interface forms.

Also, the tagged value **DescText** is defined for definition of label values for data bound control fields.

BASECLASS value defines the metaclass for which the tagged value set or stereotype is defined while **Lock** value determines element alterability through the StarUML user interface.

B. Creating second transformation templates

The modular written templates separate functionalities in different files. Every module must be included in the core template module. In the future, there can be more than one core files which could be speed-optimized for certain usages

by including only a subset of modules. Altering existing templates is easier and could be done as minor extensions or bug-fixing modifications.

C. Adding a new target language

As the tool provides the capability of generating in any added target language the user is encouraged to add new, yet unsupported, languages.

There are several steps in creating the new target language:

1. create language folder in the tool folder, matching the name pattern **<LanguageName>Templates**
2. create a start point transformation file **basic.xslt** where the second transformation starts by default. The

override exists in the specification tagged value for specific start-point transformations.

3. plug in the language to the tool by adding it to the list in the generator settings file *Settings.xml*
4. create the rest of templates
5. create type mapping between platform independent types used in specification to target language types
6. create configuration data that holds language specific generator settings

XII. 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.

Some future developments of the tool described in this paper may be enhancing of the capabilities for different platforms [15] and support for some common development methodologies [16] and approaches [19].

REFERENCES

- [1] J. Herrington, *Code Generation in Action*, Manning, 2003.
- [2] M. Lee, H. Kim, J., Lee, *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] K. Dollard, *Code Generation in Microsoft .NET*, Apress, 2004.
- [5] A. van Deursen, P. Klint and J. Visser, "Domain-Specific Languages: An Annotated Bibliography", *ACM Sigplan Notices*, 35(6), 2000, pp. 26 -36.
- [6] K. Fertalj, D. Kalpic and V. Mornar, "A Software Development Method Based on Iterative Prototyping", In Proc. *The World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, FL, USA, 1999, 83-90.
- [7] K. Fertalj, D. Kalpić and V. Mornar, "Source Code Generator Based on a Proprietary Specification Language", In Proc. *The 35th Annual Hawaii International Conference on System Sciences 2002*, Big Island, Hawaii, USA, Jan 7-10, 2002.
- [8] K. Fertalj, D. Kalpić: "Preservation of Manually Written Source Code in Case of Repeated Code Generation", In Sahni, S. ed.: *Proceedings of the LASTED International Conference on Computer Science and Technology*, May 19-22, 2003, Cancun, Mexico, ACTA Press, Anaheim, CA, ISBN 0-88986-349-0.
- [9] E.F. Codd, "Further normalization of the data base relational model". In R. Rustin (Ed.), *Data Base Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972, pp. 33-64.
- [10] C.J. Date, *An Introduction to Database Systems, Sixth Edition*, Addison Wesley, Reading, MA, 1994.
- [11] P. Heckel, "A Technique for isolating differences between files", *Comm. ACM*, 21(4), April 1978, pp. 264-268.
- [12] D. Hirschberg, "A Linear Space Algorithm for Computing Maximal Common Subsequences", *Comm. ACM*, 18(6), June 1975, pp. 341-343.
- [13] W. Miller and E. Myers, "A File Comparison Program", *Software Practice and Experience*, 15(11), 1985, pp. 1025.
- [14] E. Myers, "An O(ND) Difference Algorithm and its Variations", *Algorithmica*, 1(2), 1986, pp. 251-256.
- [15] K. Fertalj, M. Horvat: "Comparing Architectures of Mobile Applications", *WSEAS Transactions of Communications*, Vol. 4, No 3, pp. 946-951.
- [16] K. Fertalj, N. Hlupić, D. Kalpić: "RUP and XP - A Modern Perspective", *WSEAS Transactions on Information Science & Applications*, No 8, Vol 3, Aug 2006, pp. 1573-1581.
- [17] P.P. Chen, "The Entity Relationship Model Toward a Unified View of Data", *ACM Transaction on Database Systems*, Vol. 1, No. 1, 1976, pp. 9-36.
- [18] P. Fraternali, "Tools and Approaches for Developing Data-Intensive Web Applications: A Survey", *ACM Computing Surveys* 1999; 31(3), pp. 227-263.
- [19] S. McConnel S, *Rapid Development*. Redmond: Microsoft Press; 1996.
- [20] J. D. Reilly, *Designing Microsoft ASP.NET Applications*. Redmond: Microsoft Press; 2002.
- [21] I. Reinhartz-Berger, D. Dori, S., Katz, "OPM/Web - Object-Process Methodology for Developing Web Applications", *Annals of Software Engineering* 2002; 13(1-4): pp. 141-161.

- [22] Q.L. Chen, T. Shimomura, N.S. Lang, K. Ikeda, "Uniform Tag-based Rich Component Generation for Web Application Development", *WSEAS Transactions On Information Science & Applications*, Issue 12, Volume 4, December 2007, pp. 1431-1438.
- [23] T. Helman, K. Fertalj: "A Critique of Web Application Generators", In *Proceedings of the 25th International Conference on Information Technology Interfaces*, June 16-19, 2003, Cavtat, Croatia, pp. 639-644.
- [24] T. Helman, K. Fertalj: "Application Generator Based on Parametrized Templates", in L. Budin, V. Lužar-Stiffler, Z. Bekić, V. Hljuz Dobrić ed.: *Proceedings of the 26th International Conference on Information Technology Interfaces*, June 07-10, 2004, Cavtat, Croatia, SRCE, Zagreb, pp. 151-157.
- [25] J.R. Roger, "Jostraca: A Template Engine for Generative Programming", *ECOOP Workshop on Generative Programming*; 2002 June 10-14; Malaga, Spain.
- [26] J.E. Smith, *CodeSmith Template-based Code Generator*, <http://www.ericjsmith.net/codesmith/> [4/25/2004]

Kresimir Fertalj is an associate professor at the Department of Applied Computing at the Faculty of Electrical Engineering and Computing, University of Zagreb. Currently he lectures a couple of undergraduate and postgraduate courses in Computing. His professional and scientific interest is in computer-aided software engineering, complex information systems and in project management.

He has written over 90 scientific and professional publications and participated in conferences locally and abroad. He participated in a number of information system designs, implementations and evaluations.

Fertalj is member of ACM, IEEE and PMI.

Mario Brcic is a prospective assistant at the Faculty of Electrical Engineering and Computing, University of Zagreb. He graduated from Computer Science at the Faculty of Electrical Engineering and Computing, University of Zagreb in 2008. His scientific interest is in computer-aided software engineering.