



**KTH Information and  
Communication Technology**

# **Evaluation of Code Generation Tools**

**Farzad Sedghi Farooji**

Degree Project in  
Software Engineering of Distributed Systems  
KTH Information and Communication Systems  
Stockholm 2014



# Abstract

Code generation is an important part of today's software development. Using code generation can increase code quality, ease maintenance and shorten development time. It can be used for development of different parts of software systems like database access layers, communication protocols and their proxies/stubs, user interface and many others. Code generators may be ready to use products or developed in-house for project's specific requirements. There are different tools and environments for the development of code generators.

As there are so many different possibilities for code generation solutions, it becomes hard for a developer or team to choose the best solution for their purpose, especially when there are few academic or industrial resources for comparing such solutions or providing the criteria for their comparison. Most of the academic works related to code generation are about specific software areas like parsers, signal processing and embedded systems, rather than general software development.

This report defines a framework for comparison of code generation solutions, which provides a categorized list of relevant criteria for such comparison. The list of criteria is gathered by reviewing a set of available code generation solutions and categorized based on software quality attributes, since the code generation solution is software itself. Finally some of the tools are chosen based on the requirements and applications of the company and they are compared side-by-side using the comparison framework.



# Abbreviations

4GL	Fourth generation languages
ABSE	Atom-Based Software Engineering
DSL	Domain Specific Languages
DSM	Domain Specific Models
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Environment
FIX	Financial Information eXchange protocol
FTL	FreeMarker Template Language
M2M	model-to-model
M2T	model-to-text
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MERL	MetaEdit Reporting Language
MOF	Meta Object Facility
MTL	Model to Text Language
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
QVT	Query/View/Transformation
RMI	Remote Method Invocation
TTM	Time to Market
UML	Unified Modeling Language
W3C	World Wide Web Consortium
WSDL	Web Service Definition Language
VTL	Velocity Template Language
XMI	XML Metadata Interchange

# Table of Contents

<b>Abstract .....</b>	<b>2</b>
<b>Abbreviations .....</b>	<b>4</b>
<b>Table of Contents .....</b>	<b>5</b>
<b>List of Figures .....</b>	<b>8</b>
<b>List of Tables.....</b>	<b>9</b>
<b>1. Introduction .....</b>	<b>11</b>
<b>1.1. Code generation .....</b>	<b>11</b>
1.1.1. What is code generation?.....	12
1.1.2. Basic concepts .....	12
1.1.3. General process .....	14
1.1.4. Why code generation is useful.....	14
1.1.4.1. Separation of high level abstractions and system code .....	14
1.1.4.2. Code generation makes the process of implementation faster .....	16
1.1.4.3. Code generation increases software quality.....	16
1.1.5. Drawbacks and risks of using code generation.....	17
<b>1.2. Programming languages evolution.....</b>	<b>18</b>
1.2.1. Increasing the level of abstraction.....	20
1.2.2. Code generators and Compilers .....	22
1.2.2.1. Semantic changes .....	22
1.2.2.2. Intermediate presentation and optimization .....	23
<b>2. Goals and Approach .....</b>	<b>26</b>
<b>2.1. Goal of this thesis.....</b>	<b>26</b>
<b>2.2. Method .....</b>	<b>27</b>
2.2.1. Software comparison approaches .....	27
2.2.2. Approach in this work .....	29
2.2.3. Methodology .....	32
<b>3. Comparison Framework.....</b>	<b>32</b>
<b>3.1. Categorization of comparison criteria .....</b>	<b>32</b>
3.1.1. Software requirements.....	34
3.1.2. Function and Performance Requirements .....	35
<b>3.2. Comparison criteria framework.....</b>	<b>37</b>
3.2.1. Which software? Who is the user?.....	37
3.2.2. Function requirements for a code generator .....	39
3.2.3. Software quality attributes for a code generator .....	40
3.2.3.1. ....	40

3.2.4. Finalizing the categorization of tools' criteria .....	53
3.2.5. Code generation comparison criteria.....	55
3.2.5.1. Code generator usage.....	55
3.2.5.2. Code generator development .....	57
<b>4. Reviewing code generation tools .....</b>	<b>59</b>
<b>4.1. Related concepts.....</b>	<b>59</b>
4.1.1. Meta Object Facility (MOF).....	59
4.1.1.1. Related standards .....	61
4.1.1.2. Model Driven Architecture (MDA) .....	62
4.1.2. Eclipse modeling framework (EMF) .....	63
4.1.2.1. Core framework.....	63
4.1.2.2. EMF.edit.....	64
<b>4.2. Code generation tools review .....</b>	<b>66</b>
4.2.1. Atom Weaver .....	66
4.2.1.1. Concept .....	66
4.2.1.2. Code Generator usage features .....	68
4.2.1.3. Code Generator development features.....	69
4.2.2. Java Emitter Template (JET) .....	71
4.2.2.1. Concept .....	71
4.2.2.2. Rational Software Architect .....	73
4.2.2.3. Code generator usage features .....	74
4.2.2.4. Code generator development features .....	77
4.2.3. Xpand.....	81
4.2.3.1. Concept .....	81
4.2.3.2. Code generator usage features .....	85
4.2.3.3. Code generator development features: .....	89
4.2.4. Acceleo .....	91
4.2.4.1. Concept .....	91
4.2.4.2. Code generator usage features: .....	92
4.2.4.3. Code generator development features .....	95
4.2.5. AndroMDA.....	98
4.2.5.1. Concept .....	98
4.2.5.2. Code generator usage features .....	100
4.2.5.3. Code generator development features .....	103
4.2.6. MetaEdit+ .....	104
4.2.6.1. Concept .....	104
4.2.6.2. Code generator usage features .....	108
4.2.6.3. Code generator development features: .....	112
4.2.7. FreeMarker .....	116
4.2.7.1. Concept .....	116
4.2.7.2. Code generator usage features .....	117
4.2.7.3. Code generator development features .....	119
4.2.8. Actifsource .....	121
4.2.8.1. Concept .....	121
4.2.8.2. Code generator usage features .....	125
4.2.8.3. Code generator development features .....	127

4.2.9. XSLT .....	130
4.2.9.1. Concept .....	130
4.2.9.2. Code generator usage features .....	132
4.2.9.3. Code generator development features .....	133
<b>5. Code generation tools comparison .....</b>	<b>136</b>
5.1. Current code generation solutions in the company .....	136
5.2. Problems and Improvements .....	137
5.2.1. Code generator's usage complexity .....	137
5.2.2. Code generator's performance .....	138
5.2.3. Complexity of the code generator's implementation .....	138
5.2.4. Need for a user friendly development environment .....	140
5.2.5. Fulfilling the current features .....	140
5.2.6. Possible further improvements .....	140
5.3. Prioritizing the features .....	141
5.3.1. Features table for code generator usage .....	141
5.3.2. Table for code generator development .....	143
5.4. Tool selection and comparison .....	144
5.4.1. Filtering out some of the tools .....	145
5.4.1.1. Atom Weaver .....	145
5.4.1.2. JET .....	145
5.4.1.3. AndroMDA .....	146
5.4.1.4. MetaEdit+ .....	146
5.4.1.5. FreeMarker.....	147
5.4.1.6. XSLT.....	147
5.4.2. Side by side comparison .....	147
5.4.2.1. Comparison of the code generators: usage .....	148
5.4.2.2. Comparison of the code generators: development .....	151
5.4.3. Which tool to choose .....	156
<b>6. Results and Conclusion .....</b>	<b>159</b>
6.1. Background study on code generation.....	159
6.2. Tools comparison process .....	159
6.3. Tools comparison result .....	160
6.3.1. Code generation usage .....	160
6.3.2. Code generation development.....	161
6.3.3. Code generator's essential requirements.....	161
6.3.4. Importance of modeling in code generation .....	162
<b>7. Evaluation and further works .....</b>	<b>163</b>
<b>Bibliography .....</b>	<b>164</b>



# List of Figures

Figure 1-1 direct conversions from source to destination languages .....	24
Figure 1-2 conversions from source to destination languages using an intermediate format .....	25
Figure 3-1 General hierarchy for software quality attributes.....	36
Figure 3-2 Different environments and actors involved with code generator.....	38
Figure 4-1 3 layered MOF architecture, modeling with UML and DSL .....	61
Figure 4-2 All the meta-models which are MMA, MMt and MMb conform MMM, which represents MOF meta meta-model.....	63
Figure 4-3 <i>cpp_class</i> atom in the tree (left window) and the atom editor with seven categories (right window) .....	67
Figure 4-4 replacing JET template static text with a reference to meta-model element, using right-click menu.....	74
Figure 4-5 Rational Software Architect wizard for defining the meta model and mapping exemplar's elements to meta model elements. Basic versions of JET templates are created automatically, visible in Project Explorer tab .....	80
Figure 4-6 definition of meta-model, adding a new type to UML activity diagram as an example .....	106
Figure 4-7 A template for C++ function written in MERL is open in Code generation editor window...	107
Figure 4-8 code generator running in debug mode, output generated and shown below while moving on template lines .....	111
Figure 4-9 meta-model created in graphical designer.....	122
Figure 4-10 model created by help of content assist. Errors are detected based on meta-model.....	123
Figure 4-11 The template body assigned to "Entity", specified in "Selector" field and marked by vertical orange bar on the left .....	124
Figure 4-12 Inside the template partially assigned to "Attribute", specified in "Selector" field and marked by vertical orange bar on the left .....	124
Figure 4-13 built-in method "ToUppderFirst" used in template for attribute name .....	129

# List of Tables

Table 1 - Prioritizing features for code generator usage .....	142
Table 2 - Prioritizing features for code generator development .....	144
Table 3 - Comparison of the code generators' usage, Functional Requirements .....	148
Table 4 - Comparison of the code generators' usage, Reliability .....	149
Table 5 - Comparison of the code generators' usage, Extendability .....	149
Table 6 - Comparison of the code generators' usage, Maintainability .....	149
Table 7 - Comparison of the code generators' usage, Portability .....	150
Table 8 - Comparison of the code generators' usage, Usability .....	150
Table 9 - Comparison of the code generators' usage, Resource saving .....	151
Table 10 - Comparison of the code generators' development, Functional Requirements .....	151
Table 11 - Comparison of the code generators' development, Reliability .....	152
Table 12 - Comparison of the code generators' development, Maintainability .....	153
Table 13 - Comparison of the code generators' development, Improvability and Connectability .....	154
Table 14 - Comparison of the code generators' development, Portability .....	154
Table 15 - Usability .....	155
Table 16 - Financial saving .....	156



# 1. Introduction

## 1.1. Code generation

Code generator is a program that is usually used to produce the source code for another piece of software. In many cases the programmers find themselves more comfortable to spend some time to write a piece of program which generates the desired code, instead of writing that code manually. The main goal is usually to reduce the time needed to produce the code, since programmers can find repetitive and similar tasks in the process of writing the code that persuades them to use a program to do that instead. Another reason to use a code generator can be the ease of changing and extending the software. In fact there are many benefits in code generation which makes developers use it to generate source code and other artifacts like documentation and test cases.

Today it is nearly impossible to find software that is developed without using code generation. This use can be restricted and used only for some special parts of the system like generation of database access layer by using specific code generation tools like Hibernate tools or using Apache Axis for generation of java proxy and stub for using SOAP<sup>1</sup> Web services in the system. IDEs<sup>2</sup> always provide a lot of features that generate code for the developers like wizards to create new classes, getters and setters for class attributes or generating Javadoc for Java code.

On the other hand, instead of using in a restricted area, it is possible to use code generation for development of most parts of the software parts, as it is done in Model Driven Software Development (MDSD). In such systems, the most parts of the software are generated based on high level models or languages. There are many successful software products [1], developed using Model Driven Architecture (MDA) in different fields like financial and banking, railways and transportation management and Space programs.

---

<sup>1</sup> Simple Object Access Protocol, an XML-based protocol used for implementation of web services

<sup>2</sup> Integrated Development Environment

### **1.1.1. What is code generation?**

Code generation is about separation of a system's design and high level concepts from the actual software code. Using code generation, a developer wants to define the system specifications in a separate place than the final code, using a language of higher level compared to the final code, and then use a code generator to construct the actual software. This code generator can either be developed by the developer him/herself or it can be an already existing one. A code generator can be thought as a tool that has a limited and simple set of tasks, which can be used repeatedly and in a proper order to form a bigger task that is the generation of a specific piece of software.

A classic example of a code generator is a tool for creating database access layer. The code generator software has the knowledge and ability to construct the needed code to access some specific database type. It knows how to create libraries to read, write or update tables in a database. It doesn't matter how many tables a database has or how many columns each table has. Also the names of tables and their columns are not important. The code generator works based on a set of high level knowledge on how a database system should be accessed, rather than the metadata about a specific database like table names and column types. Based on such knowledge, it can create the code to access this database. The user provides the code generator with the specifications of the database, say database schema, and leaves the tedious and repetitive task of building the access layer code in a programming language to the code generator. So no matter how many tables there are, as soon as the definition of the tables is provided, creating the access layer is done by running the code generator.

The higher speed of development is not the only benefit of using a code generator. There are other benefits in using code generators during software development which will be listed and explained, after having a general overview on how code generation is done.

### **1.1.2. Basic concepts**

A simple "Hello world!" method can be very useful to demonstrate the fundamentals of a typical code generator. Imagine a simple method which takes a person's name as a string and says hello to the person. So if "Joe" is given as the input, it prints out "Hello Joe!" and giving "World" results in "Hello World!" There are three main concepts in this example that are the

fundamentals of code generation. First there is a method, or piece of program that writes the whole sentence instead of the user. So the user doesn't have to write the whole sentence for each person. It can be a complete letter starting by "Hello Joe!" The next point is that the program takes an input, to build up a sentence based on it. So there is a way, known to the program, to take the input from a user and act based on that. In other words, the tool knows where and how the input is going to be introduced to it. The third point is that the program knows what to write. It has the knowledge on how to write a hello world sentence or a letter, starting by saying hello to the receiver of the letter. It saves this knowledge as a string inside it that can be later combined by the input name to form the final text.

How is this example related to a code generator? The method can be a code generator, which generates the source code of a complicated method or component, based on the input it takes. In all the code generations, the code generator takes a series of parameters as input, which specifies the output code, exactly as the input parameter of the above method. This input can be very complicated, like specification of a software component. So it is usually structured in a more efficient way like an XML file, which is a very usual way of giving input to a code generator. This XML file or input set of data is always in a format that is familiar for the code generator. For example if an XML is provided to a code generator, the code generator can be implemented based on the XML file's schema, so it knows what each tag means and what to do with them. Or if the input is a UML diagram for a component, the code generator should have the knowledge about UML stereotypes and formats.

Now imagine the above method takes two parameters. The first one is the person's name as before and the second one specifies the text of the letter, among a list of texts that the method has inside. So for example calling the method by Joe and number 1 will result in a welcome letter for Joe and calling it by Joe and 2 will end up in a letter of eviction for him. The same can happen in a code generator. For instance a code generator for a class has different templates, among which there is one template for generating the getter and another for generating the setter of an attribute. So it can apply each template for each attribute inside the class, to generate a getter and a setter for that attribute. Now if an attribute is specified as read-only on the input, the code generator "knows" that it should only call getter template for that attribute.

### 1.1.3. General process

A code generator is a piece of program that knows the meaning of the input and applies the proper template to the input to get the desired output.

The input for code generation can be in any format, e.g. XML, text file, UML model or Java code. The code generator then should process this input file to find the proper templates to use. The more structured the input file the easier the input processing. For example if a code generator knows the definition of an XML file, then it knows all its elements and where to look to find the needed data for generation. But if the input for a code generation is unstructured like a text file, then the code generator may need powerful or even intelligent text parsing abilities to search through the text and find the required data.

In order to generate the output, the code generator keeps the format of the output text in the form of templates. Templates vary from a fixed string or a sentence with one variable word to a complicated combination of strings, logical expressions, method calls and references to other templates. Template's its purpose is to provide the output text based on the input parameters, with a specific format and standard.

Knowing the general concepts and steps in code generation, it's now easier to talk about benefits of using code generation in software development.

### 1.1.4. Why code generation is useful<sup>3</sup>

There are a couple of important benefits that can be achieved as the result of using code generation. In this section these benefits are stated in three groups.

#### 1.1.4.1. *Separation of high level abstractions and system code*

There is always some kind of high level definition in code generation that is defined somewhere other than the final generated code. This high level definition can be anything like database schema, API definition or definition of a component in a complicated system. In all these cases, this definition is at a high level, e.g. business logic, which is usually not a part of the final

---

<sup>3</sup> [26] [27] [28]

working software and has a higher level of abstraction. This separation leads to the benefits below:

- Since the design decisions and high level concepts of the system are represented in a readable and short format, instead of being embedded within the software code, it is easier to review the functionality of the system by looking at this definition. In the traditional systems, looking at the software code, it is not easy to see the functional design of the system through the lines of a general purpose language like Java.
- Since the definition and logic of the system is kept in an implementation independent source like some XML files, changing the output format is also easier compared to traditional software systems. For example it's enough to change the templates and create new ones to switch the implementation of the same system from Java to C++, being sure that the logic is kept the same as before. Also for the same reason, it is easier to generate other deliverable outputs like documentation and test cases.

This is why code generation makes the system less dependent on the changes in technology. It is possible to change the templates according to the latest technology or version of a language, run the code generator and have the same system logic, but now compatible with the new technology.

- The definition files keep inside the knowledge of the domain. The definitions or models are a kind of documentation. They can be readable for a system designer or business expert if well designed, and also can be used in code generation to generate the final documentation. This makes the system less dependent on knowledge of people, since the knowledge is kept in a readable format. So change in the members of the development team will have less impact on the project schedule, compared to traditionally developed systems.
- System logic can be defined in a more readable format for a business expert. This can bridge the gap between Business and IT and let both sides talk the same language. Using modeling methods like Domain Specific Languages (DSLs), it is even possible to let the business persons create the model of system directly, which is one step apart from the



final product; running the code generator. That can highly reduce the time of development and number of functional errors, caused by misunderstanding of the business logic by the system developer.

#### ***1.1.4.2. Code generation makes the process of implementation faster***

Many parts of the final product can be generated from input models. These input models contain elements, each of which can be translated to multiple lines of code in a fraction of a second. This makes the implementation of the software faster, compared to when it is going to be hand written. There are some benefits as consequence:

- The development of the project is more cost effective since the code generation can reduce Time to Market (TTM) and costs of the project like less people needed in the team and also results in a higher level of code quality which makes it easier to maintain and improve.
- Shorter iterations in production, makes it possible for the team and domain experts to quickly find the design and requirements problems, fix them and go to the next iteration. So as it's easier and faster to update and change the software, software can survive better in the markets with changing requirements.
- Since a lot of implementation is done automatically, engineers have more time to spend on more important and interesting stuff like software architecture, performance qualities of the software or the code generator itself. This is a reason why can be said code generation is not against hand written coding since the important code that should be handwritten will be of a higher quality and good hand written code can be repeated a lot of times by its usage in templates.

#### ***1.1.4.3. Code generation increases software quality***

- Since the engineers and high skilled people can spend their time to develop the code generator, they can use their best practices in the templates and the structure of the generated code, which means these best practices are always used everywhere in the system and it results in a higher quality software.

- The system is less error prone since there is more time to test the system, especially because the focus can be on functionality of the system. This is because the technical errors can be tested during the development of the code generator, and also since the code generator is used many times and in many places, the possible bugs will be found faster. In addition one fix in the code generator or a template may fix a number of errors in the system at once.
- Using the code generator, it's possible to force the members of the team work within the architecture of the system, which results in architectural consistency in the system.
- The knowledge of the system is gathered in the definition files and models, which means there is a single point of knowledge in the system. This prevents replicated knowledge in the system which is the source of errors and anomalies and makes it hard to update and maintain the system. Although this benefit may not be available for the systems using different modeling environments and partial code generations, still having some of the concepts and logics in high level definitions makes the system one step closer to having a single point of knowledge and less data inconsistency. As a result it is easier to evolve such systems to have all of their meta-data in an integrated collection.

### **1.1.5. Drawbacks and risks of using code generation**

There are also drawbacks in using a code generator which should be taken into consideration when deciding to use a code generator. Not all of the tasks are suitable for handling by code generation. In addition not the teams are suitable for working with code generation.

- More time is needed for modeling and architectural decision in the start of project, which means the first cycle may become longer if code generation is going to be used from the start of project.
- Programmers are unwilling to switch to use code generation for some parts of their work. Most of them prefer to continue doing the task using the methods and skills they have and directly, instead of handling it via another piece of code. It is never easy to switch, especially when one thinks the current method is working fine.

- Programmers tend to modify the generated code which either should not be done or it should be done only under control of the code generator.
- If the definition/model is not designed to be flexible, system flexibility will be low and hard to change. Flexibility should be taken into consideration from the first day of using a code generator.
- The code generation should be used for the projects which are suitable for using it. There should be enough repetition in the development to make its use justifiable.
- Version controlling is also needed for generators, templates and frameworks which requires additional effort. In the projects where version controlling is required also for the generated files, it becomes harder to do.
- Code generator itself needs documentation and maintenance as a tool for software development. It is a development tool which requires some extra effort, apart from its development effort, compared to other development tools which are not domain specific like code generators.
- If the project fails before the first version of a system is completed, there is usually less result compared to when the system is developed traditionally.
- It can be hard to test the code generator's transformation, especially if the fault is not repeated in many places of the result code. Then it is hard to see the bug by testing. So code generator itself can be an additional source of bugs in system development.

## ***1.2. Programming languages evolution***

The following section is a short review on different generations of the programming languages and how they are prepared for the machine to be executed. This is relevant to code generation since shows the process of programming languages evaluation with the goal of providing the developers with a higher level of abstraction, which is the same concern in code generation. So

the differences and similarities of such solutions, especially compared to compilers, can be interesting and useful for the process of code generation development.

The first computer programs were written as a sequence of bits to represent numbers and the operations, which were basic operations like sum and assignment. Development of programs with such a method is slow, complex and also error prone. In addition reading, understanding and modification of such pieces of code is hard to do. This was the first generation of programming language.

The first step to improve the programming languages was introduction of assembly language which allowed the programmers to use a set of mnemonic instructions or addresses, instead of their binary form. Later macros were added as well, to be used instead of frequently used pieces of code. Assembly code is translated to machine code by a piece of program called assembler. The task of assembler is basically replacing each mnemonic instruction or variable with its equivalent piece of binary instruction or address and in case of macros, the macro calls are replaced by the definition of the macro, before converting the code to machine code. This was maybe the first code generation by computer, to help programmers with development of their code. Assembly language is the second generation of programming languages.

FORTRAN, COBOL and Lisp were the first from the third generation languages. They provide the developers with high level notations and instructions which helps them develop their code easier and in a more natural and robust way. The code written in such languages needs more than replacement with equivalent machine code to make them executable for the processor, since they are in a higher level of abstraction compared to assembly code. A compiler is used to create the machine executable version of the program.

Code compilation is done in a couple of steps, grouped into two main phases which are code analysis and synthesis. Analysis is generally the preparation of the input for the final code generation step, in an intermediate presentation of the original program that is an easy to read structure for the code generator. The synthesis phase is then basically the code generation part which produces the executable code as output. Optimization of the program for improvement of

speed or resource usage is also done on the intermediate representation, before the final code generation.

The high level concepts are captured and taken care of in the analysis phase in the compiler and in other words, the analysis phase is “aware of” the semantics of the language. So when it’s time for code generation, the code generator only cares about transforming the same sequence of operations into machine code. The conversion is more like a word to word translation, rather than interpretation of concept in low level terms.

Third generation programming languages provided the programmers with higher level general concepts which were valid and useful in any field of usage like more complex mathematical and logical operations, flow control instructions or I/O methods. Although the instructions are conceptually at a higher level compared to machine language, they are still general to all the domains and don’t capture the logic and concepts of any specific target business. That’s why they are called general purpose languages and the context related concepts should be captured in the programs by the developers and in the body of the program code.

There are fourth and fifth generation languages have tried to increase the level of abstraction even higher to provide an easier and less error prone development process. Fourth generation languages (4GL) are specific to their usage domain and are categorized to groups like database query languages, data manipulation and analysis, report generators or web development languages [2]. Fifth generation languages are also focusing on declarative programming and defining constraints to define the problem and let the computer solve the problem.

### **1.2.1. Increasing the level of abstraction**

When a language is intended to be even higher in the level of abstraction and closer to the domain concepts, it cannot be used in the other domains since now it is customized for that specific domain. So each domain needs its own language at such a level. But there is an important point and that is there are a huge number of usage domains, if not infinite, and providing one language per domain is impossible. Looking at the huge number of third generation languages, while they are meant to be general purpose, shows the fact that the higher the abstraction level gets, the more is the number of corresponding languages.

The third generation languages are general purpose from business domain point of view, since the high level concerns they are fulfilling is common and valid in all the business domains. For example one may use Java for its portability, C++ for performance, a functional language like Erlang<sup>4</sup> for easily implementing complex logic in a small piece of code or SQL is usually needed where a database exists. All of these concerns are high level concerns compared to assembly language, while they are cross domain concerns. So it is meaningful to develop languages to fulfill such level of abstraction and be sure that the language will be used many times and not only inside one company!

4GLs are the examples of the languages which help the programmer develop their code, using the terms and concepts of a target domain and this is why they are sometimes referred to as a subset of Domain Specific Languages [3]. Most of the available and ready to use 4GLs belong to the domains which are used like common aspects in all the business and problem domains like database queries, data analysis and reporting. But development of a language per problem domain is not necessarily the easiest or best approach for increasing the level of abstraction since, as mentioned earlier, then there is a need for development of nearly an infinite number of languages. They will be so much in number that it seems more logical to develop them on demand.

But this does not mean there is no way to develop programs in a higher level. It's always possible to express different concepts of the system in a high level format like XML files, text files or UML models. Code generation techniques and tools are used to convert these high level descriptions of the systems into source code which usually belongs to third and fourth generation languages. Model Driven software development is based on using modeling tools and languages to model a system and then transform these models to source code for different parts of the final product. The source code itself maybe 3GL like Java or 4GL like SQL.

So these modeling tools and languages, together with their code generators play the role of a high level language for development of a system. But the solution is not a standard language since the models and the code generator are both created and customized for a specific domain. In fact each development team can use specific development environments to develop their own

---

<sup>4</sup> <http://www.erlang.org/>

modeling language or tool in order to use it for generation of source code, so they can have as much as their domain specific details they want in their high level development language or tool.

### **1.2.2. Code generators and Compilers**

There are differences and similarities between the process of development of programming languages and using their compiler and development of software by using models and code generators. Looking at these differences and similarities may help in development of a good code generator.

#### **1.2.2.1. *Semantic changes***

A major difference between code generation in compiler and code generation for software development is the semantic difference between input and output of each. A compiler receives a program written in a specific language and converts it to machine language, without changing its semantics. So the result is exactly the same program, but in another language and lower level of abstraction.

This is not what happens in high level code generation, since the generated code may only use parts of the input model for generation of a specific piece of code and it also will add parts of the output code from its templates which include semantics that are not included in the input model. For example the input model maybe the data about some books and the code generator may generate SQL commands to generate their database tables. Another code generator may create its data layer access in Java language and also documentation may be generated. So this means the input and output of the code generators are not the same semantically, while this is the case with compilers.

One reason for the above difference is the higher level of models as input to the code generators, compared to compilers. In case the models are used to model the whole solution, they cannot usually be implemented in only one language. So each aspect of the solution needs its own specific code generator, if it's going to be generated. So each code generator tries to extract only the parts in the input model which are related to its task and no matter if the generation covers

the whole development or only a part of it, the implementation semantics are added to the result, which are stored in the templates.

This difference in changes of semantics introduces some risks that should be considered during the design of a code generator. Since the code generator is adding some new semantics to the output and code generation should be a flexible and maintainable solution, these semantics should be presented in a readable format. Compilers do not change the programs semantics and they are not meant to be changed rapidly, while code generators may need to be changed based on the requirement changes in the business domain. That's why if there is any complicated concept in code generation, it should be presented in the templates and in a readable and understandable format.

For example if Java code is responsible for preparing the input model to a code generator, like parsing it from a text document, it is better not to change the data semantically in java code and instead handle such changes in code generator. The reason is that the languages used in code generators for transformation are more readable and understandable, because they are designed for such tasks compared to Java as a general purpose language. This means all the semantic additions to the output can be kept in a set of readable templates, instead of being embedded in the middle of Java code.

#### ***1.2.2.2. Intermediate presentation and optimization***

One advantage of the intermediate code generated by compiler is that when there are multiple input languages and multiple target machine languages, there is no need to create a compiler for each pair of input language and output target. All the input languages are converted to the intermediate code and then different generators are created for each target environment. Also a part of the optimizations can be written only for the intermediate format.



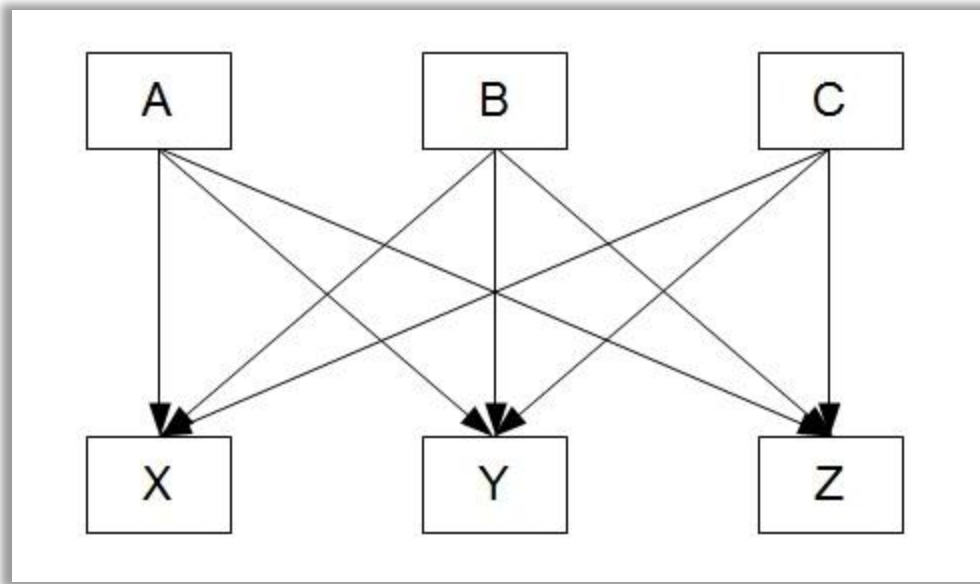


Figure 1-1 direct conversions from source to destination languages

As shown in Figure 1-1 there are nine conversions needed from three source languages to three target languages. Also changes in each of the target or source languages, needs modification of three conversion processes. But by using an intermediate representation, according to Figure 1-2, not only less conversions are needed, but also as the source and final languages are decoupled by an intermediate representation, changes in each of them only needs a modification in one conversion.

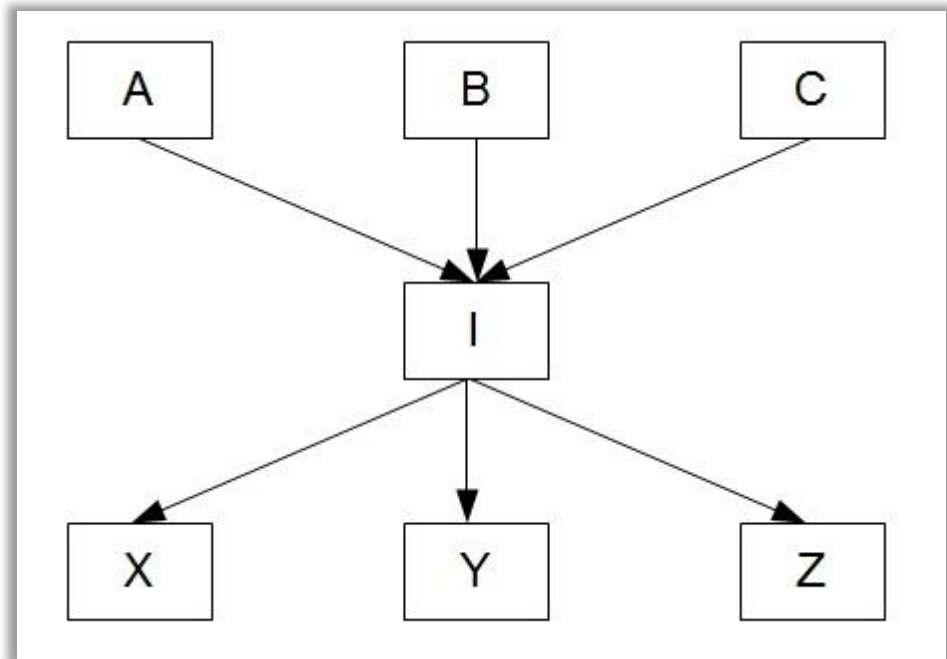


Figure 1-2 conversions from source to destination languages using an intermediate format

## 2. Goals and Approach

### 2.1. Goal of *this thesis*

While code generation is used a lot in industry and a huge amount of produced code is always automatically generated, there is not much academic work focusing on this subject compared to how important and useful it is in the industry, especially in software development field. There are many academic works and papers about code generation in the fields like embedded systems or compilers. In the field of software development, most of the works are about Model Driven Engineering (MDE), which is a methodology based on generation of the software from definition of models. It is easier to find related works in some restricted fields like API code generation, database access layers and web services.

But in reality, there are a lot of cases that code generation can happen in any part of systems and also in development of any kind of software systems. So there are a lot of important factors that should be considered when choosing a tool. Also looking at all the benefits and drawbacks of code generation, it is obvious that it is not an easy decision which code generation solution to use. It is important to know where in the project code generation is useful, how it should be used, what tool should be used and a couple of more criteria that can highly affect the life cycle of a project.

In addition there are a lot of code generation tools and solutions to choose between. They may differ for instance in what they can generate, how they accept the input, whether they are free or open source or what parts of a software architecture can be generated using them. In order to have as much of the benefits of generating code as possible and also avoid the drawbacks, many factors should be considered to find the best tool and setting for code generation in the project.

The goal of this work is to define an evaluation framework which can be applied to different code generation tools in order to make their comparison easier. Having such a framework and set of criteria, one can be encouraged to think about taking advantage of different code generation tools, because it is easier to make sure that all the necessary factors for choosing the tool are taken into consideration and also the process of tool selection is done faster.

Furthermore, since this thesis work is defined by a company, the concerns of the company regarding code generation are important in direction of this work. According to these concerns, it is possible to find the features in the code generator tools that are mostly needed by the company.

Here is the list of the main concerns:

- Complexity which makes maintainability, improvability and also learn ability hard
- The code generation process should be faster
- Since the current tool is an old solution, it is interesting for the company to know about the new tools and technologies and what possibilities are provided in the newer tools that can help them improve their code generation and in general, software development process.
- It is not easy for developers to use current tools for their own everyday use. Improving this can motivate them to use code generation as much as possible.

So there are two main results of this work which can be useful for the company. First is reviewing a number of code generation tools that may be useful for the company and the other is the evaluation framework, which can be applied to any other code generation tools in the future.

## **2.2. Method**

### **2.2.1. Software comparison approaches**

Evaluation of the software systems is an important problem since it's needed in all the domains using software. It's complex because it needs a lot of parameters to be considered. An evaluation model like a framework needs to be developed and applied to the software for evaluation. The development process of such a model starts by identification of relevant quality attributes of the product, as suggested in ISO/IEC 9126 [4] which is an international standard that provides a framework for software evaluation.

ISO/IEC 9126 first defines the set of software quality characteristics which can be used for evaluation of software products and then provides some guidelines for using these characteristics in the process of evaluation. These characteristics are more or less the same set of attributes which are mentioned as functional and nonfunctional requirements of software, like reliability, maintainability and usability. It later suggests an evaluation process model to show the required steps for software evaluation.

The first suggested step is to define the quality requirements based on the domain and environment in which the software is going to be used. The next step is to prepare the basis for the evaluation by defining metrics for quantifying the quality attributes and making them measurable. Finally the assessment criteria should be defined, meaning providing the means to summarize the result of evaluation, e.g. decision tables and weighted average of the rates for different quality attributes. Having such an evaluation model prepared, it can be applied to different software.

These steps are general guidelines provided by this standard. The different works done in the field of software evaluation and comparison have their own specific set of criteria. They take the same steps as stated in this standard, but have their own specific set of software characteristics. Such characteristics may be similar in different cases, but they are tailored for each specific context and requirements.

Some evaluation methodologies suggest their own fixed set of quality characteristics, as they are not general solutions and are applicable for their specific domains. There are other approaches mentioned as “constructive quality model approaches” [4] [5] [6] that suggest taking a general quality model and construct a customized set of quality attributes, based on the usage domain. Using such a flexible method needs more effort and skill, compared to the approaches which suggest a prepared and fixed set of attributes.

Other required steps of the evaluation are also customized and modified based on the specific domain and methodology. For instance, one approach may pay more attention to user’s perspectives and requirements while the other may suggest more detailed steps for preparation of

the evaluation model. The following is a suggestion about the required steps for preparing an evaluation model [7]:

1. Identifying the relevant actors and their roles, purpose of the evaluation, the available resources and the objects of evaluation
2. specifying the type of evaluation which can be in form of ranking them as highest to lowest preferred or formal description of the software
3. defining a non-redundant hierarchy of evaluation attributes
4. associating a measure, a criterion scale and a function to transform the measure scale into the criterion scale for each attribute
5. An aggregation technique to help aggregate the values and make a recommendation

### **2.2.2. Approach in this work**

The steps mentioned in the previous section are the guidelines chosen in this thesis work, as the steps to take for creating an evaluation framework. There is no ready to use framework for software evaluation. Such a framework depends not only on type of the software, but also is affected by the requirements of its target environment.

In addition, in the field of code generation solutions, there is lack of similar works, which makes it impossible to find a prepared comparison framework to adapt. The existing works are not in the field of general software development and development of code generators. Also the comparisons between the tools that can be found over the internet are not detailed and informative enough to help with finding a proper solution. They usually list their general features like their supported operating systems, input and output.

One important fact about this thesis work is that as it is defined by a company for practical purposes, one of its major goals is to provide an easily and practically usable evaluation approach. That's why it is kept simple and the complexities needed in the process of evaluation are avoided, so hopefully the method can be used easily for later evaluation of other tools as

well. So here is how each step is taken in the following thesis work, based on the five steps mentioned above:

1. There are two main actors influenced by the code generation solution. The code generator developer and the developer who uses the code generator. So the requirements are set based on the perspectives of both of them. The evaluation is done based on the current problems in the code generation solutions of the company and the goal is to solve the problems and have a look at possible improvements. Different existing code generation tools and development environments are reviewed during this work.
2. Although the task is defined to be used practically, still it does not have a very narrow and specific scope. There are a couple of problems mentioned that belong to different existing code generation solutions in the company and are explained in more details later in “Current code generation solutions in the company”, page 136. While the company needs to address these problems, it is also open to see the further possible improvements.

When it comes to compare the tools practically, the problem needs to be restricted. So out of all the company’s code generation tasks, the main task is chosen as a code generation implementation use case. The task is believed to have all the crucial requirements that all the other tasks include.

3. The hierarchy of the quality attributes is a modified version of the quality attributes mentioned in [8], customized based on the context. The context not only includes the problems and requirements of the company, but also the fact that the software under evaluation is a code generation solution which has its own characteristics and requirements, independent of the usage domain. One reason to choose this software quality attribute hierarchy is that this book is one of the main references used in the company for definition of software quality standards.
4. This step is highly affected by the purpose of definition of this thesis work, as well as its scope. There should be a set of measures for each attributes and also a function that maps the measured values to an understandable scale, which makes it useful for decision making. The approach taken in this work to measure the quality attributes is based on the

features of the tools which affect each of them. The list of features affecting each attribute is found by reviewing the tools and their approaches for doing each step of code generation. This is possible to do since the basic concepts of all the tools, as template based code generation solutions, are the same and their provided features are comparable with each other. Also some experiences are done with each tool for finding their powers and weaknesses.

So for each quality attribute, all the affecting features are found after reviewing different tools. Then based on the company requirements, a priority is assigned to each feature, which determines if this is a mandatory, trivial or improvement feature. This provides a checklist of all features that makes it easy to review each quality attributes in a feature-based approach. The final result is one table per each quality attribute that shows which features are available for each tool, while the features are prioritized.

This output is understandable and easily extendable for more tools since:

- The output is readable because of its simplicity
  - The features are categorized based on the quality attributes and summarized based on the company's priorities
  - Using it for a new tools is easy and only need filling out a new binary column in each table
  - The features are already prioritized based on the company's requirements to help the process of decision
5. The solution provides a side by side comparison of categorized and prioritized features to help finding the final tool or set of tools. The prioritized list of features helps filtering out all the tools that does not provide the mandatory features. Then the final best tools are used to implementing a real world task, in order to find the best match to company's needs.



### **2.2.3. Methodology<sup>5</sup>**

This work is a qualitative research trying to develop a framework which can be used to evaluate and compare different code generation and code generator development tools. It has a realistic philosophical assumption, as it tries to understand the collected data in order to build knowledge from it, which is an evaluation framework. As the goal of the work is to find all the important features of some specific tools to build up a practical framework that can be used for comparing and choosing a proper tool, the research method is an applied method.

The research strategy and data collection method are based on a use case, which is implementation of a sample code generation task by different tools. In order to analyze the collected data, it is tried to quantify the different qualitative aspects of these tools by finding a list of all important features for each quality aspect and give each of them a high, medium or low priority, based on requirements of the use case.

Ethical issues are not relevant to this work. So apart from general issues, no ethical issues are considered.

## **3. Comparison Framework**

### ***3.1. Categorization of comparison criteria***

There are a lot of features and capabilities that a code generation tool may have and depending on the application, they can be more or less important for a user or project. Some developers need a fast code generator and some need a flexible and extendible, while some may need a very easy to use tool. So one should know what to care about more and what is less important when choosing a tool. Each existing tool has its own capabilities which may or may not be useful for different tasks. So in order for a developer to find a good tool matching the task, there should be a way to find the most important criteria based on their application. An input to the process of tool selection is the requirements of the development process. Based on factors such as

---

<sup>5</sup> Methodology based on Håkansson, A. (2013) [35]

development methodology, architecture of the system, modeling framework, project build tools and even skill level of the team members, the development team should decide which tool is more useful and suitable for their task. After considering such factors, it is time to check different characteristics of the existing tools and find the tool that satisfies their requirements.

In this section the features and capabilities of code generators will be categorized to make it easier for one to evaluate a tool based on them. So once the development needs are specified, it is easier to find which tool is more suitable for the development process, based on the features of the available tools. Since code generators are software, the quality measures of software will be used to categorize their features. So for instance, the features that affect the maintainability of the code generator like debugging capability will be placed under maintainability category, among all the software quality measures that can describe the quality of the code generator as software.

Such a categorization fits the selection process mentioned above. When a development team knows what the requirements of their development process are, they can express the characteristics of their required tool in terms of software function and quality requirements. Then for each of these quality factors, they can use this categorization to find out which features should be available in their desired tool.

As an example, imagine a team wants to generate Java code for their application. They want to generate an API that is an implementation of a specific communication protocol which is defined as an XML file. Also since the protocol is under development and may change quickly over time, it is important for them to be able to extend the code generator easily. From these requirements it is possible to find the characteristics of their desired code generator, as software. They need a tool that can accept the input model as an XML file and also be able to generate Java code as output. These are the function requirements of the tool that they need. Also they need an extendable tool, so it should be easy for them to change the code generator. Extendability is a quality measure of software. By looking at the extendability category in the list, it is possible to find the features that help a tool be extendable. So then it's easier to know which tool is helpful and which is not.

### 3.1.1. Software requirements

A code generator is a piece of software and can be described using software requirements. The following is one of the definitions for software requirements:

*“Requirements give information to the system designers and to a wide range of stakeholders. They state what the stakeholders want the system to achieve” [8]*

The software development team, as the stakeholder, knows why there is a need for using the code generator and how this tool should work. So they can specify their required tool in terms of software requirements. According to the same source, software requirements are grouped into the following categories:

**Vision:** at the highest level, the future direction for a system.

**Function Requirements:** what a system has to ‘do’: the essence of a system, its mission and fundamental functionality.

**Performance Requirements:** the performance levels that the stakeholders want their objectives. How good?

**Resource Requirements:** the levels of resources that stakeholder plan to expend to develop and operate a system. Resources have to be balanced against the stakeholders’ perceived values gained from the system functions and the system performance levels.

**Design Constraints:** any design ideas that must be included in the system design

**Condition Constraints:** these are any additional constraints to those imposed by the other types of requirement

These are the requirement types that can be used to describe a software system’s different aspects like what it should do, how well it should do it or what are the constraints of the system.

The most relevant categories to the goal of this thesis work are Function requirements and Performance requirements. These requirements are chosen to categorize the features of the code generation tool since they cover what the system should do and how well it should do that. Of course all of the other requirement types can be used for specifying a code generation tool but

choosing the most important ones helps to keep the work simpler and also avoid too much complexity in categorization of the code generator's features.

### **3.1.2. Function and Performance Requirements**

Function requirements specify what a system should do. It is a list of features or attributes that the system should have, no matter how well they are provided. A system either has a Function attribute or not.

On the other hand there are Performance requirements which specify how well the system should perform its tasks, which are specified by the Function requirements.

*“Performance describes the system benefits: how good the system is and how it affects the external world. Performance attributes state the actual and/or potential benefits and effects experienced by stakeholders in their environments.” [8]*

There are a lot of performance attributes which can be considered for a system. One categorization divides such attributes into three main groups [8], which are themselves decomposed into smaller groups of attributes, and then suggests a general hierarchy of the performance attributes. The general hierarchy is presented in Figure 3-1.

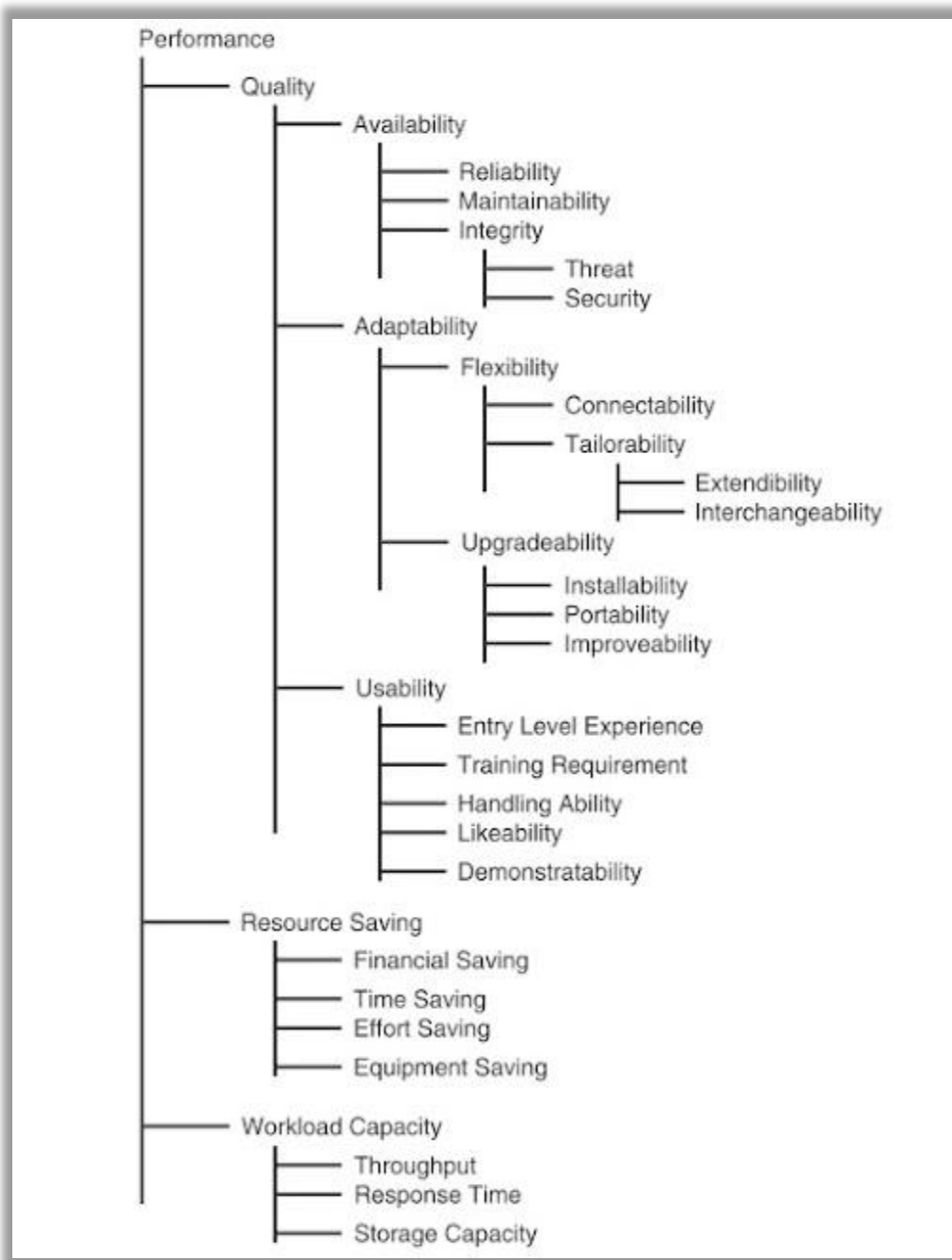


Figure 3-1 General hierarchy for software quality attributes

This hierarchy is used as the base for categorizing the features of code generators in this thesis work. Since it tries to cover all the performance aspects of software, it is possible to label the

different features of the tool with the performance attributes to show which area or areas of software performance they have impact on.

As mentioned before, this hierarchy of the performance attributes is a general hierarchy and as stated in [8], *“You should learn the art of developing your own tailored scales of measure for the performance and resource attributes, which are important to your organization or system. You cannot rely on being ‘given the answer’ about how to quantify”*. This hierarchy is used as a guideline for classifying features of code generators, in order to create a framework for representing the criteria important to choose a tool.

## **3.2. Comparison criteria framework**

The goal of this section is to review the software quality attributes and find which features of a code generator has impact on each of these attributes. Function requirements and Performance requirements are chosen to classify the features in the tools.

### **3.2.1. Which software? Who is the user?**

There is an important point to consider when talking about a code generation tool’s quality as software. There are two types of developers who are involved with the code generators. First type is the code generator developer, who creates the code generator parts like templates and the code generator engine itself. The second type is the software developer, who uses the code generator to develop software. The code generator developer uses a development environment to build the code generator, so the code generator user can later use it, maybe in another development environment. These two types of developers can be the same developer, or two developers of the same company or they can be in different companies.

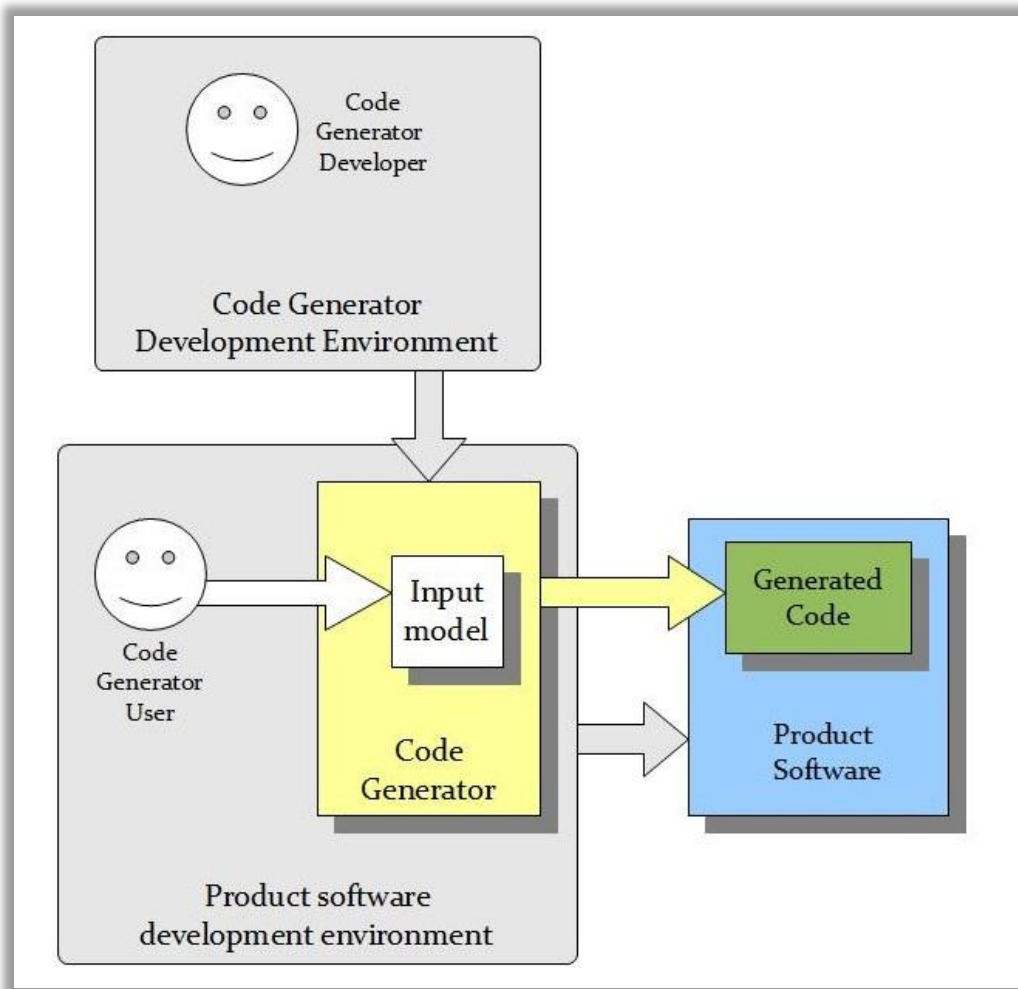


Figure 3-2 Different environments and actors involved with code generator

As suggested in Figure 3-2, ‘Code Generator’ is the outcome of the development by ‘Code Generator Developer’, which is done in ‘Development Environment’ for the Code Generator. Then the developer who uses the code generator, called ‘Code Generator User’, uses it by providing the proper input to generate some code for use in development of another software, called ‘Product Software’ here. So as shown the environment for development of Product Software can be different with Code Generator Development Environment. Similarly the developer of the code generator and the Product software may not be the same developers.

So while trying to look at performance attributes of a code generator, it is important to consider which developer and also which development environment is being reviewed. For example when talking about maintainability, it is possible to talk about maintainability of the code generator

itself and also maintainability of the software, which is produced using the code generator. Both of these maintainability types may be important for a developer when it comes to use a code generator and since they are about different pieces of software, different features are needed to support them. This complexity is caused by the fact that the code generator is a piece of software which outputs another piece of software.

As a result, in the following section the software quality attributes are explained from two points of view. First is the code generator developer who uses a development environment to create a code generator and the second is the product software developer's point of view, who uses the code generator to create another software or Product Software.

### **3.2.2. Function requirements for a code generator**

The function requirements are the necessary capabilities of the system. The functional requirements define a system in terms of what it should do. They specify the features or capabilities of the system that if any of them are removed, the system's nature will be changed. They are essence of a system and what the system should be able to do.

There are some features of software that can be categorized as function requirements. Output format is one of the most important function requirements for a code generator. When a system is being developed in C++, then generating code from input model to Java code is not acceptable under any condition. So generating the code in C++ is a function requirement of the code generator in that system. Maybe another project also requires generating documentation and automatic tests. Then they are also the function requirements of the system, defining the output of the system.

Format of the input model may also be a function requirement in some systems. Maybe the code generator should use a set of existing resources as input. So in order to integrate the code generator with the system, it is mandatory to use that type of input model. In contrast, maybe the developer of the code generator is allowed to use any kind of input model like XML file or a plain text file. In this case, the choice of the input model is up to the developer of that generator and is not part of the function requirements.



### 3.2.3. Software quality attributes for a code generator

In this part the different software quality attributes will be explained and discussed in the context of code generation, to see how much important each of them is and how they are related to code generators. This will help to show which features of a code generator is related to each of these quality attributes that represent a software quality aspect. This can help finding a quality hierarchy, specific for code generators, which can be simpler than the original hierarchy. Each attribute is explained by a short description from [8] in the title.

#### 3.2.3.1.

##### *1. Availability*

Availability of software is how much the software is ready to do what it should. So if it's not serving as it should for any reason like the system being down or malfunctioning, it is not available. Whenever software is not performing as intended, it needs maintenance. Also during the maintenance, the software is not available. So there are two important factors that affect the availability of the system. The first one is how often the system needs maintenance, which means it is not available, and the second is how hard it is to maintain the system when needed to make it available again.

##### *1.1. Reliability: The system performs as intended*

###### *Code generator usage*

It is very important that the code generation tool itself works correctly, since the output of this tool is going to be used in the production software. For example when a transformation from the input model to Java code is done wrong, it has direct effect on the functionality of the final product. This may happen because of a fault in the template of the code generator, or a bug in the code generator transformation engine.

How to avoid an unreliable code generator? It depends on whether the tool is developed in house or is an off the shelf or an open source tool. If it's a commercial or open source ready-to-use tool, then it is possible to read the reviews in the user's communities to find if there are any reported problems with the tools. So a good factor is the popularity of the tool and an active user

community. But if the tool is developed in house, then one needs a good development environment for the code generation tool which helps the developer avoid making mistakes while developing the tool. A good IDE can provide the code generator developer with supports like code completion and real time error detection based on system's meta-models, refactoring or pre-defined code generation patterns which can help the developer to work more accurately and with less risk.

### *Code generator development*

Reliability is also important for the environment in which the code generator is developed. It is important since if the code generator, as the output of this environment, is a faulty tool then it will generate the wrong code for the final product. But since the output of this environment is not directly used in the final product, the importance of reliability for this environment is not as much as the reliability of the tool itself. The example of such a fault in code generator development environment is when the graphical tool used in the environment for defining the meta-models in the system generate the wrong output, based on which the code generator transforms the model into code. But such a problem can be discovered during the development of the code generator, which is before the code generator usage, during development of the product software.

In order to find such unreliable behaviors in a development environment, again one should read the reviews and comments of the users in the forums and online communities. Usually commercial tools have better support and higher level of reliability, so if this factor is very important for code generator developers, they can give the priority to the commercial tools.

## ***1.2. Maintainability: Resource required for repairing an unreliable system***

### *Code generator usage*

When there is a fault in the final generated code and the product software is acting unreliable, then the product developer should maintain the software. It is very important for the developer to quickly find the source of the problem and be able to fix it easily. So imagine the product developer has debugged the software and has found that the problem is in one part of the generated code. The next step is to find the source of this part of code in the input model of the

code generator. This fault can be a result of a wrong input to code generator or because of a bug in the code generator itself. In any case the developer should easily find the place in the input model, which is the source of this part of the code, to see what the problem is. The code generator can help the developer to do so by providing Traceability. It is a very important feature in the code generator that helps with the maintainability of the generated code by letting the developer find the correspondence between the source model and the generated code.

After finding the source of the fault, the developer should fix it and then make it ready for test. It should also be easy for the developers to see the result of their change fast enough to be able to change it again if the correction is not suitable for this problem. If every time it takes long to have the generated code with the new correction, then the maintenance process takes a long time from the developer, especially if she/he does not know what is the exact problem and wants to do some tests to find the best solution. So the faster the code generator, the easier is the maintenance of the code.

Now imagine the process of a code generation task is so huge that no matter how fast the tool, the developer must wait for a few minutes each time after the fix, to see the result of their fix. The ability of the code generator to generate code from a part of the model instead of generating the complete output again is a significant feature to reduce the time of generation. There are times when the developer or the code generator itself knows that change in some parts of the model has only impact on some specific parts of the code, without any dependency on other parts of the model, so there is no need to run the whole generation process based on all parts of the source model. Then this feature comes very handy since the task becomes lighter and the change takes place faster. Some tools provide such a feature by keeping the track of changes in the input model and only generating the code for changed parts of it, to avoid regeneration of the same content each time. Also in some of the code generation tools the user can manually specify the parts of the model for which he/she wants to generate new code.

### *Code generator development*

There are times when a developer finds that there is a fault in the generated code, which is caused by a bug in the code generator itself and not the source model. Again it is traceability which helps the code generator developer to find the place of the fault e.g. a faulty template in

code generator. So traceability is an important feature that is very helpful in the maintenance of the code generator itself as well as the generated code.

There are some other features that have effect on the maintainability of a code generator. The ability to debug the generation process is one of these important features. Generally the usability of the development environment has a direct effect on the ability of the code generator developer to fix the faults in it.

In some solutions there may be a need to generate an intermediate model which is the result of all the input models, and then convert that model to the final code. In such cases, it is very useful to provide the developer with this generated intermediate model, to help better understanding the logic of the code generation. If the intermediate model is generated in the memory and removed after the generation is finished, it is harder for the developer to find the source of problem, especially when there is no traceability available from output to the input models and the process of constructing the model is complicated or it is sourced from many different input models.

Furthermore, another important factor is again an active user community and also development and support team which can help in solving the similar problems, reporting bugs and learning about the features of the development environment. It is always better to face an already reported bug, rather than being its reporter!

## ***2. Adaptability: the efficiency with which a system can be changed***

There are times in the life of a system that the development team has to change the software and how it works in different ways. This change can be on how the system is configured or a more fundamental change like changing a subsystem or component. Software may need a change when there is a change in the environment with which the software interacts, when there is a change in the requirements of the system or when there is a need for refactoring and improvement of code quality.

## ***2.1. Flexibility: the ‘in-built’ ability of the system to adapt or to be adapted by its users to suit conditions***

All of the required changes in a system are not always fundamental changes. These kinds of changes are the ones that don’t need a development effort to take place. They can be connecting the system to work with different environments or adding a new entity to the system like a new user or node. Flexibility concerns how easy these types of changes can be handled in a system.

### ***2.1.1. Connectability: ‘the cost to interconnect the system to its environment’***

#### *Code generator usage*

The code generator tool should be integrable with system in which the developer uses it. Looking at connectability from functional point of view, the input format and output format and content are part of the fundamental requirements of a code generator, which are part of its design and development.

But connectability may also be important about the integrability of the code generator with other tools in the project like project build tools. If code generation should be a part of the build process of a project, then it should be integrable with it, e.g. it should be runnable using Ant. This will be covered also later in portability of the code generator.

#### *Code generator development*

What are the systems that the code generator development environment should interact with? A development environment can be as simple as a text editor for writing Ruby code or can be an environment integrable with an existing IDE, like an Eclipse plug-in. It may also support one or a specific set of meta-model formats to define the input model with. This is actually another important aspect of such an environment. If a company is already using a special framework for modeling, then the ability of a code generation framework to interact with that modeling environment can be very useful. Not only the development process can be faster because of the existing system models, but also the system’s knowledge is kept in a single place which keeps its consistency high.

### **2.1.2. Extendibility**

#### *Code generator usage*

Extendibility of the software being developed by help of a code generator mostly depends on the design and architecture of this software. Of course using a code generator can help a lot with extendibility of the software under development since it's one of the main goals of code generation, but the code generator is not what determines the extendibility of that software.

One feature that can help with extendibility of the software is the ability to put manual code or custom code in the middle of the generated code. In this case, all parts of the generated code are not created based on templates and input model, since there is a part of code that can be hand written in the middle of the generated code. The code generator should recognize this part in some way and preserve it from being overwritten by generated code, while generating code later.

Another feature which may be helpful on extending the product software by code generation and changing the input model is the automatic update of the generated code. In case the input model is created in the same environment as the code generator, the integrity of the environment may allow the generator to update the code instantly, based on the changes in the input model. This helps avoiding the developer to generate code after changing the input model.

#### *Code generator development*

How can a code generator be extended? What kinds of changes are available to be added to a code generator, with the help of its existing features and without a need of a fundamental change and development? One of the very common extensions to a code generator is to add a new output format like a new language. This is one of the main purposes of using a code generator. The logic of the program is kept in the code generator and it can be converted to any language, if correct templates are provided for it. So it is important to know how easy it is to add a new output format to a code generator.

Another important change in the code generator can be change in the format of its input model. The input of a code generator should have a known structure. For an XML file as input, an XML schema can be used as a meta-model to define its expected format. The implementation of a code

generator can be such that in order to change the input format, it is enough to change the schema of the input XML, or more generally, the meta-model of the input model.

For example the input model can be an XML file, which complies with a schema file that is used by code generator to parse the XML inputs. In order to change the input model, the schema file should be changed and if this change is as simple as adding a field to a message or adding a new message, similar to other existing messages, there is no need to do any development in the code generator.

Another available feature in some tools is extending the existing templates of the code generator by wrapping it with some additional commands, without changing the inside of the template. In this case the template is used as a black box and is used whenever suitable with a little bit of extension. This feature or the similar features are referred to aspect oriented programming support, in the context of code generation.

## ***2.2 Upgradeability: The cost to modify the system fundamentally***

In contrast to flexibility, upgradability of a system is about the changes which are fundamental to the system. It can be changing of a system component which needs a complete development cycle in the system or can be installing a system from scratch in a new environment.

### ***2.2.1 Portability: The cost to move from location to location***

#### *Code generator usage*

Developers of the same system may develop their code in different environments. So if there is a code generator, which is purchased or developed based on the product requirements, it is important that it is usable in these different environments. It should be runnable on different operating systems. If it's a standalone application, should be checked whether it is runnable on all the required operating systems. For instance when the code generator is a runnable Jar file, then there is no worry about its portability.

Another concern about a tool's portability is whether it is integrable with build tools in a development environment. It may be necessary to call a code generator during a build process. It

can be a task like generating API from an XML definition, which should be included in the build before the system can run. So it is important to make sure it is possible to run the code generator tool using the existing build tools, when choosing to use the tool in the project.

### *Code generator development*

If there should be a code generator developed in house, then the portability of its environment matters. But it may not be as important as portability of the code generator itself, since everybody may need to use the code generator, but not everyone develops it. So usually it is not a big deal if an environment for code generator development is not very portable, since one or a team develops it in a special environment, then a lot of people use the tool in their environment. So it is portability of the tool itself that matters more.

By the way if the tool is developed by many different people, e.g. it is the strategy of the company to encourage people to use code generation in their development as much as possible, then it is important to have a portable environment to develop, extend or customize the tool. Of course every one can develop their own code generators in any environment they want, but if there is a specific framework or technology that fulfills everyone's needs for creating a suitable code generator, then it is better for everyone to use the same framework. It can have a lot of advantages like compatible model formats, reuse of the parts of code developed by others, possible to have a more active and populated community of people using the same framework which can collaborate and also as a result, holding the licenses for the company for that framework is then more profitable and useful for more people.

So in this case again it is important to know which operating system or IDE's are compatible with this environment.

### ***2.2.2. Improvability: The cost to enhance the system***

Improvability includes changing software fundamentally, such that there are design and development efforts needed to make these changes. It's different from extendibility since extending software is adding some capabilities to the software in a predicted and already provided way that does not need any special development.



### *Code generator usage*

Extendibility of the code generator can impact the improvability of the product software. For instance adding a new component may need extending the code generation models and templates. Modularity of the models is one factor which can ease extending the code generator. The development environment should provide proper means for managing and maintenance of the input models and meta-model. Ease of editing and refactoring the templates is also another important factor which depends on environment usability. Type based template languages, which provide the meta-models as static types in the process of development, are good examples of usable tools for refactoring the code generator, according to the changes in model definitions.

### *Code generator development*

When a code generator needs improvement in its input model, then there is a need to change the model definition in the development environment, which means changing the meta-model for the generator's input model. This meta-model may itself have a definition language, named meta-meta-model, based on which the modeling environment used in the code generator development environment can interact with the meta-model.

For instance when XML schema is used in the development environment of a code generator to define the models, then the modeling framework needs to know the definition of this schema, in order to provide the developer with graphical representation of the input meta-model. The environment needs to know the language in which the meta-model is defined. For instance the meta-meta-model in eclipse modeling environment is Ecore, which is used to define meta-models like UML or domain specific languages (DSL).

Such a meta meta-model is needed, when changing the input meta-models of a code generator. Let's say there is a need to switch from using UML input models to an in-house type of modeling or a DSL. Then the code generator should be able to read this new format of model definition or meta-model. If the framework provides the means to introduce such a new format, it means it's possible to change the input meta-model to probably any other format. But if there is no means to define a custom meta-model, like a code generation tool which only accepts one or a fixed set of input format, the possibility to change the input meta-model is restricted to a

predefined set. Not all the code generators and their corresponding modeling frameworks provide the means to define the system models using a custom meta-model or a DSL.

### ***3. Usability: How easy a system is to use***

An important characteristic of any software is how easy it is for its user to interact with it. This quality of the software is very important for both the code generator tool and also its development environment. A major reason why usability is so important when it comes to code generators is that there is usually a resistance from developers, who are used to develop everything manually, to use code generation during their development. They know how to do things without using a code generator and many of them who are skilled and experienced developers, know very good ways for solving different problems. In order to persuade a developer to use code generation in their development process, it is not always enough to tell them about the advantages of using it. It is very important that they can start using it easily, and communicate with the tool and progress with it fast enough to feel that it is a useful change. Otherwise they will give up using the tool and start to do the jobs in the same way as before.

#### ***3.1. Entry level experience***

##### *Code generator usage*

How much skill should the developer have to start using the code generator? What kind of knowledge and skills are prerequisite before starting to use this tool? These are the important questions that should be answered before choosing a code generator as a tool for development. If a lot of people in a project fail to start using the tool or don't have the knowledge to use it, then the tool is rejected by the team and the cost of creating or buying the tool is wasted.

##### *Code generator development*

It is also important to know the required level of skill needed for a developer in order to start the development of a code generator in a specific environment. But this quality measure is less importance for the development environment of a code generator tool, compared to the tool itself. The reason is that all the users of the code generator are not necessarily the developers of

it. So there is less cost to start working in this environment, if their skills are not enough, since there are less people who need to gain the required skills.

If the developer and the user of a code generator are the same, then only entry level for code generator development environment is important, since the developer can use their own developed software for sure.

What matters for a developer who starts development in a new environment is the amount of educational support for the environment. Tutorials, active communities and forums, good documentation, seminars and workshops are important in this case.

### ***3.2. Training requirement***

How much training and learning effort is needed for a user to get enough proficiency in working with the software? The more complex or extensive the software, the more training effort is needed.

#### *Code generator usage*

The code generator user should provide the tool with input model and then run the generation process to get the desired output. So if the format of the input model is complicated, then the user needs more time to learn how to create a model in a correct way. The input model can be an XML file, defined according to a schema file, which can have a very simple structure or very complicated format. The input model can be as simple as java code itself when generating Javadoc, which means there is no special thing a user should do to generate the code, rather than running it. A DSL or domain specific language is another example of an input for a code generator, which can make creating the input model easy since DSLs are designed for the use of the people in a specific domain who are familiar with terms and concepts of that domain.

The required amount of training for a code generator is an important factor which affects the chance of accepting the tool by the developers. Also if it takes long for the developers before knowing how to use it correct, the process of development may become very slow which can raise the risk of project's failure.

### *Code generator development*

Again like entry level experience, this factor is not as important for the development environment, compared to the code generator itself. It's because the number of the developers of a code generator is usually less than the number of its users.

### **3.3. Likeability**

Likeability of a system is a result of all the other attributes of the system like other usability attributes, performance and maintainability. Also it mainly depends on the user and may be caused by personal reasons, rather than quality of the software. For instance if a user is familiar with a development environment and the new code generation tool is integrated to this environment, then the user may like the tool, although the tool may not be a really user friendly tool.

A good way to measure the likeability of a tool is to collect user's opinions about it. Using a questionnaire is a good way of doing this.

## **4. Resource saving**

### **4.1 Time saving**

#### *Code generator usage*

Using a code generator in the development process can reduce the time to market because a lot of code is produced automatically and faster by the computer instead of a programmer. But before being able to use a code generator, time and effort is needed for development of the code generator itself and also for learning how to use it properly. All of these depend on the type of the task defined for the generator and available resources in a project. So this quality aspect is more project dependent and it is not easy to find some code generator features which always have the same effect on resource saving. Generally the higher the quality of the code generator, the more resources are saved. For instance when it is easier to work with a code generator, less time is needed to learn the tool and accomplish the task with it and also less bugs are produced.

In the same way, higher level of skill and knowledge in team will reduce the time for the project to start using a new tool. Also a faster code generator makes the development of a project easier.

An important issue when using a code generator, regarding time and effort saving, is regeneration of the same code and hence rewriting the file with the same content as a result of a change in a part of model, irrelevant to the newly generated piece of code. Specially in large projects, this can take a lot of time from the developer waiting for the code to be generated and also for version controlling of the generated files, since there can be a lot of generated files each time among which only a few are really changed and need to be committed. The ability to avoid regeneration or rewriting of the unnecessary files is a feature of code generation tools which can influence time and effort saving.

#### *Code generator development*

The speed of code generation is also important for saving time in development or maintenance process. If the developer has to wait a lot in order to see the result of each change he/she makes, the process of its development can become slow. As a result, partial code generation is important for time saving during the process of development as well.

#### **4.2 Financial saving**

Although using code generation in its right place can reduce development costs e.g. less developers are needed, but it can be a commercial tool which costs more than developers for the company. So the fact that if it is commercial or free and also its price if it's a commercial tool are other important criteria for choosing a good tool.

#### **5. Workload Capacity: The raw ability of the system to perform work**

The most important measure for a code generator in workload capacity is its response time. A slow code generator will lower the interest of a developer in using the tool. The faster the tool, the more comfortable is the developer to change the model and generate the new code, which has direct effect on the software's maintainability and improvability.

### 3.2.4. Finalizing the categorization of tools' criteria

During the previous section, the software requirements were explained in the context of code generation. The connection between each software quality measure and the features of code generation tools and environments was demonstrated. By looking at the result of such a review, it is possible to see that there are some groups of measures which are affected by the same set of features of a code generator. This can help to customize the hierarchy of quality measures for the code generation software, by packing the related measures together and having a more summarized and easy to use hierarchy of features for code generation tools.

So the related measures will be discussed and grouped in this section. Again this will be done once for the code generation tools and then for code generator's development environment. Then the final categorization, including the list of related features for each quality category will be proposed.

#### *Code generator usage*

The first quality measure of a code generator that can be merged with another is its connectability. Connectability of a tool is about its capability to connect and interact with its environment. In the case of a code generator, the format of the input and output is what affects its interaction with its environment. But the format of the input and output in a code generator tool is a fundamental requirement of the system, which makes it a part of its functional requirement. As a result, it is not needed to put this tool's characteristic under connectability. Low level connectability like interaction with IDE or OS is also included in portability of the tool, so it is possible to remove connectability from the hierarchy, when talking about a code generator.

Improvability of the product software, which is using the generated code, can be also removed since fundamental changes in the product software are not a task that should be done using the code generator. A code generator is a tool to provide extendibility for software and this is achieved by changing the system in a high level and let the tool take care of the rest. But improvement, which means fundamental change, needs new development and design. This development can also include changes and extensions to the code generator, which are addressed in next section.

The resulting categories of features which are important for code generation tools are:

- Functional requirements like input and output format
- Reliability of the code generator
- Extendibility of the final software, developed by help of the code generator
- Maintainability of the final software, developed by help of the code generator
- Portability of the code generator
- Usability of the code generator
- Response time of the code generator

### *Code generator development*

Extendibility and maintainability of the code generator are closely related with each other. An easily maintainable system is also easily extendable. IDE supports like debugging, refactoring, highlight and code completion are the examples of what has effect on these two quality measures for a code generator development environment. An extension to a code generator can be adding a new meta-model element. This is the same as trying to change an existing meta-model for maintenance. So that's why these two measures can be grouped together as one.

What if there is a need to change the language of defining a new meta-model? Meta meta-models are used to define the meta-models in a system, which are themselves the meta-models for code generator's meta-models. This meta-model can conform to XML schema format, or can be defined by any other language like Ecore, which as a meta meta-model in Eclipse Modeling Environment (EMF) is a language to describe meta-models. So to change the format of the meta-model, the environment should support the language to define them. In that case the modeling part of the environment is improvable to a great extent, which is one of the most important parts of developing a code generator. When an environment supports definition of meta-models in it,

like supporting of Ecore in EMF based tools, it means the environment has the ability to interact with a lot of different meta-models, since it is possible to add the new meta-models to the environment, using a meta meta-model, to be able to work with them. This is in contrast with the environments that support only one meta-model format, like xml-schema. Those environments are not able to connect to modeling frameworks that use another set of meta-model definition language. So the improvability of the code generator is closely related to its connectability to other systems and environments.

Now it is possible to have the final items in the list of software requirements, used to categorize the features of a code generator development environment:

- Functional requirements like working with a certain modeling environment
- Reliability of the code generator's development environment
- Maintainability and extendibility of the code generator
- Improvability and connectability of the code generator
- Portability of the code generator's development environment
- Usability of the code generator development environment

### **3.2.5. Code generation comparison criteria**

The following is the categorization of the possible features of code generator and its development environment, based on the relevant software requirement types for each. This categorization is the result of this chapter's discussion about software quality attributes and can be used as a reference for choosing a code generation tool based on expectations of the development team from a code generation tool as software.

#### **3.2.5.1. *Code generator usage***

##### **Function requirements**

- Input/output format



### **Reliability of the code generator**

- Validations for input model e.g. against meta-model
- Active user community and up-to-date bug reporting system (Open source code generator)
- Commercial tools

And if the tool is developed in house:

- High usability of the development environment
- Error detection and warning in the development environment
- Testing framework in the development environment
- Possible to reuse the meta-model parts during modeling, in order to keep semantic consistency e.g. avoiding different definitions for the same concept like time

### **Extendibility of the product software, developed by using the tool**

- Use of manual code between the generated code
- Auto update of the generated code after changing the model

### **Maintainability of the product software, developed by using the tool**

- Traceability of generated code back to template and meta-data definition.
- Code generation speed
- Input model readability

### **Portability of the code generator**

- Operating system compatibility
- Build tools compatibility e.g. Ant, Maven
- Stand alone or IDE dependant

### **Usability of the code generator**

- Documentation, auto generation of documentation for input model

- Easy running e.g. integrated with IDE
- Complexity of the model definition and Modeling framework support e.g. Graphical interface for creating the input model

### **Resource saving**

- Partial generation, based on the model changes
- Possibility to generate partially from models, based on user selection
- Optimized performance e.g. caching of the queries to input models

And if the tool is developed in house:

- Performance optimization support in development environment

### **3.2.5.2. Code generator development**

#### **Functional requirements**

- Specific to the usage and context e.g. Working with specific modeling framework

#### **Reliability of the code generator's development environment**

- Active user community and up-to-date bug reporting system
- Commercial development environment

#### **Maintainability and extendibility of the code generator**

- Traceability of generated code back to template and metadata definition.
- Readability of the input model and templates
- IDE for generating templates, debugging capability
- Editor support e.g. Code highlight, Code completion
- Refactoring support
- Aspect-oriented schema for extending the templates as black-box

- Possible to reuse the meta-model parts during modeling, in order to keep semantic consistency and faster development

### **Improvability and connectability of the of the code generator**

- How many different meta meta-models are supported by the development environment e.g. XML Schema and UML
- Does the environment provide the means to define custom meta-models or DSLs, using a meta meta-model?
- Refactoring support
- model abstraction not very close to generated code e.g. using language specific concepts in modeling

### **Portability of the code generator's development environment**

- Operating system compatibility
- Stand alone or IDE dependant

### **Usability of the code generator's development environment**

- Documentation and training material
- Language/format for metadata: ease of using this language to define new meta-models
- Graphical user interface for modeling
- IDE support: generating templates, debugging capability
- Editor support e.g. Code highlight, Code completion, refactoring, Error detection, warning, outline, open declaration
- Handling the formatting of the output code from templates: no need to format the output text separately after generation e.g. handling of whitespaces and indentation in the template

### **Financial saving**

- Commercial or free
- Price

## 4. Reviewing code generation tools

In this chapter some code generation tools are introduced and reviewed. For each tool, first its basics and fundamental concepts are explained and then in ‘features’ sections, their features are stated and categorized, based on code generation tools criteria framework. Features are first stated for code generation usage environment and then for their development environment.

### ***4.1. Related concepts***

Before starting to introduce the tools, two topics are explained shortly. Meta Object Facility (MOF) which is a meta-modeling standard and Eclipse Modeling Framework (EMF). These two topics will be referred to many times when reviewing the tools, because their standards and related technologies are used in many code generation tools.

#### **4.1.1. Meta Object Facility (MOF)**

MOF [9] is a well known standard for model-driven engineering by Object Management Group (OMG) [10]. It is the result of formalization of different meta-modeling technologies by OMG. MOF is referred to as the foundation of OMG’s industry-standard environment where a model can be:

- exported from applications
- imported to applications
- transported across a network
- stored and retrieved in a repository
- rendered to different model formats
- transformed
- used to generate application code

In order to support such functions, the models should not be necessarily implemented in UML and can be defined in any language that is defined based on the MOF specification.

MOF suggests a four-layered meta-modeling architecture, providing a meta meta-model at the highest level, called M3 layer. Meta meta-model is used to describe the meta-model in layer 2, or M2 layer, which itself should be used for defining the format of the models, in M1 layer. An example of a M2 model is UML language - not UML models - and domain specific modeling languages. The models used for structural or behavioral modeling of a system are in layer M1 and examples of those models are UML models. The lowest layer, M0 or data layer is used to describe the real world objects.

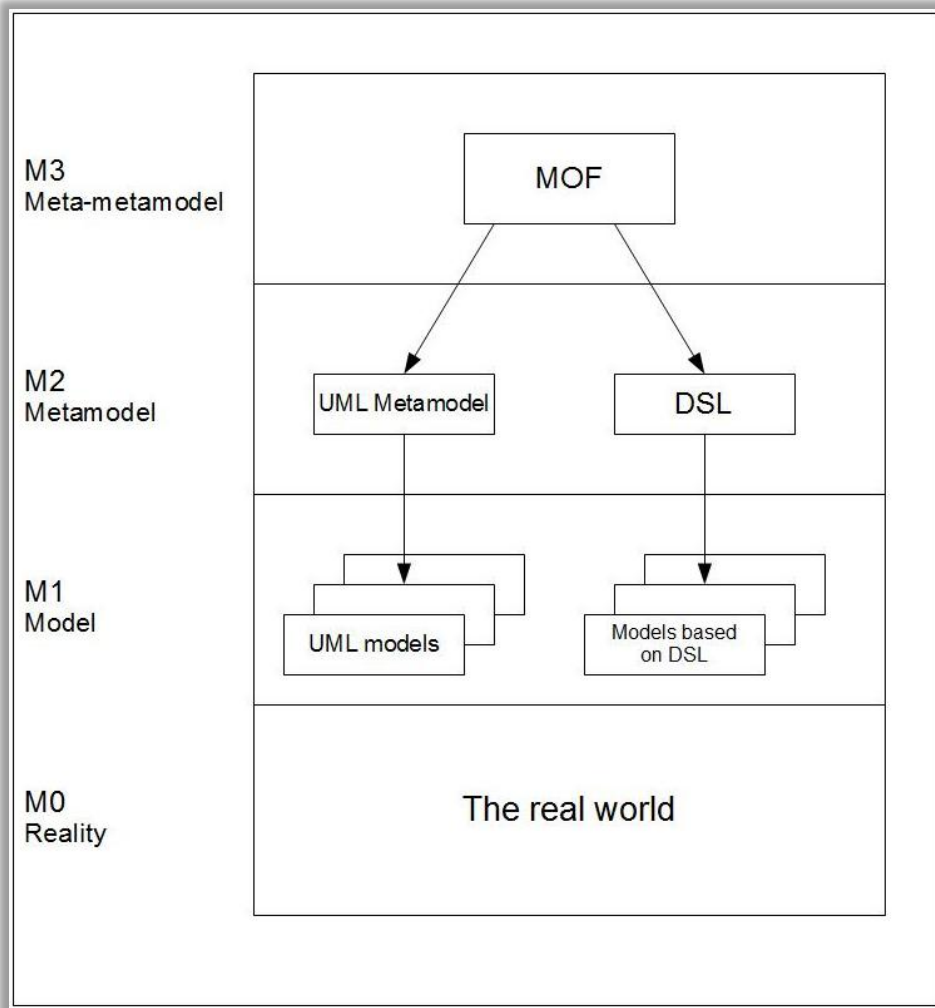


Figure 4-1 3 layered MOF architecture, modeling with UML and DSL

MOF is a closed meta-modeling architecture and the M3 model conforms to itself. It does not provide any implementation for the architecture and only defines the abstract syntax to define the meta-models. MOF in meta-modeling is similar to Extended Backus-Naur Form (EBNF) [11] in definition of programming languages. Similar to EBNF, MOF can be defined using MOF.

#### 4.1.1.1. *Related standards*

There are a couple of supporting standards for MOF. XML Metadata Interchange (XMI) [12] is an XML-based exchange format which is used for serialization of models and meta-models in a physical format. XMI is used for providing interoperability between MOF-based software and tools.

Object Constraint Language (OCL) [13] is another related standard which is a declarative language that is used to define constraints and query expressions on MOF-based models and meta-models. It was originally developed by IBM for description of the rules applied to UML models.

#### **4.1.1.2. *Model Driven Architecture (MDA)***

MDA is OMG's industry-standard architecture for model-driven software development and it is defined based on MOF. MDA suggests the complete separation between a system's conceptual design and the platform in which it is going to be implemented. The system is first described in a Platform Independent Model (PIM) which covers the concepts and design of the system. This model is created using a meta-model - M2 model - like a DSL.

Then this model is transformed into one or more Platform Specific Models (PSM), in one or more steps. Such transformations from PIM to PIM and also PIM to PSM are called model-to-model (M2M) transformations. PSM is created based on the specifications of the target platform, while it is not still a runnable code. OMG defines a standard set of languages, called Query/View/Transformation (QVT) for supporting M2M transformations.

The next step is to generate runnable code like Java or SQL, based on PSM. This transformation is called model-to-text (M2T). Like QVT for M2M transformations, a model transformation language is specified by OMG in a standard called MOFM2.

As suggested in MDA, the meta-models for input model and output model of transformations should conform to MOF meta meta-model or MMM in Figure 4-2. Also the transformation model, Mt in the same figure, which is used to transform the input model to output should conform to a model, defined by MMM.

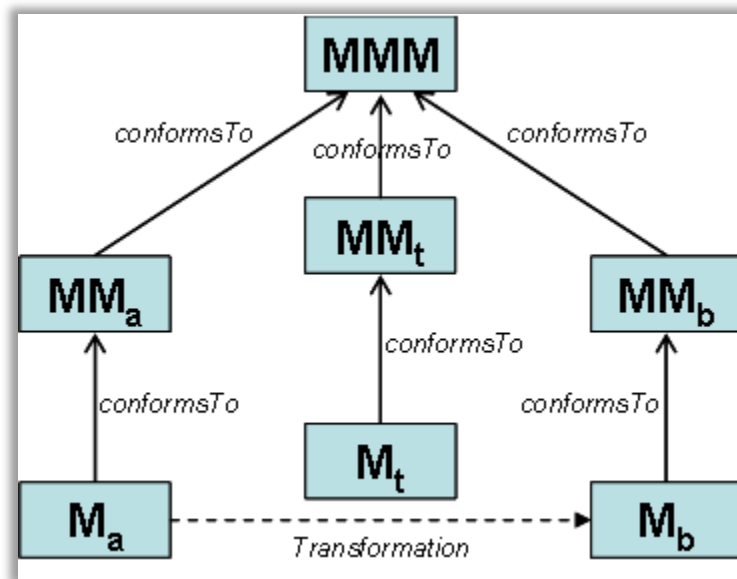


Figure 4-2 All the meta-models which are  $MM_a$ ,  $MM_t$  and  $MM_b$  conform  $MMM$ , which represents MOF meta meta-model<sup>6</sup>

### 4.1.2. Eclipse modeling framework (EMF)

EMF is a Java framework and code generation facility for building applications and tools, based on structured models. EMF started as an Implementation of MOF specification, but now it can be thought of as a highly efficient java implementation of a core subset of the MOF API. The meta-model definition or meta meta-model in EMF is called Ecore.

EMF consists of two frameworks. Core framework is used to create the Java implementation classes for a model by providing basic generation and runtime support. The other framework is EMF.edit which is built on top of core framework and adds support for generation of adapter classes for viewing and command-based editing of the models, as well as a basic model editor.

#### 4.1.2.1. Core framework

Models are defined as XMI in EMF, which is the format in which the UML models are persisted. So one way to get the models into EMF is using UML tools to define the models and then export XMI from them. But there are other ways to do it as well. Another way is to define the model by

<sup>6</sup> Figure from <http://wiki.eclipse.org/ATL/Concepts>



using XML schema, which is the description of model serialization. The other way is to annotate Java interfaces, including model properties.

As soon as the model is imported to EMF using one of these methods, EMF creates their corresponding Java implementation classes. These EMF models have lots of benefits and applications as below:

- an efficient reflective API for manipulating EMF objects
- model change notification
- model persistence support including XMI and schema-based XML serialization
- a framework for model validation
- increasing productivity

The reflective API for manipulating EMF Java objects can be used from code generator templates to access the model elements. Also another very important benefit of using EMF is the interoperability with other EMF-based tools and applications. There are lots of software like modeling tools, code generators and DSL development tools available, which are based on EMF [14]. So development of software based on EMF makes the software interoperable with all of these tools.

#### **4.1.2.2. *EMF.edit***

EMF.edit is a framework that includes generic reusable classes that can be used to build editors for EMF models. Its features are:

- Ability to display EMF models in standard desktop (JFace) viewers and property sheets<sup>7</sup>
- A common framework for building model editors, supporting fully automatic undo and redo
- Generating everything needed to build a complete editor plug-in for the EMF model

---

<sup>7</sup> [http://msdn.microsoft.com/en-us/library/windows/desktop/bb774538\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb774538(v=vs.85).aspx)

So as a result, a customized model editor is generated which can be easily updated based on meta-model changes and can be used for viewing and editing the input model, in case of code generation.

## **4.2. Code generation tools review**

### **4.2.1. Atom Weaver**

#### **4.2.1.1. Concept**

As described on its website, Atom Weaver is a commercial code generation and model-driven IDE, implementing Atom-Based Software Engineering (ABSE) as its own model driven software development methodology. With ABSE it is possible to create reusable assets which can later be used to develop specific parts of other software. These assets, called Atoms, are used to break-up and capture the problem into concepts, ideas and features, as opposed to traditional methods which break the problem into objects, functions and data.

Atom templates are knowledge units that contain the ideas, concepts and features in the form of a specific language called Lua<sup>8</sup>. This code is divided into seven categories which are ‘Admin’, ‘Form’, ‘Create’, ‘Pre’, ‘Exec’, ‘Post’ and ‘Functions’, each of which have their own specific tab in Atom’s editor window in the IDE, as shown in Figure 4-3. ‘Exec’ is where the transformation code is defined. ‘Create’ is like a constructor, ‘Pre’ and ‘post’ are executed before and after ‘Exec’ and ‘Functions’ includes the set of internal methods, defined for this Atom. ‘Admin’ is the code section for management issues of the Atom like labeling and adding description to make its reuse easy and fast. Finally ‘Form’ is used to define the input form based on templates input parameters, used to instantiate the template.

---

<sup>8</sup> <http://www.lua.org/>

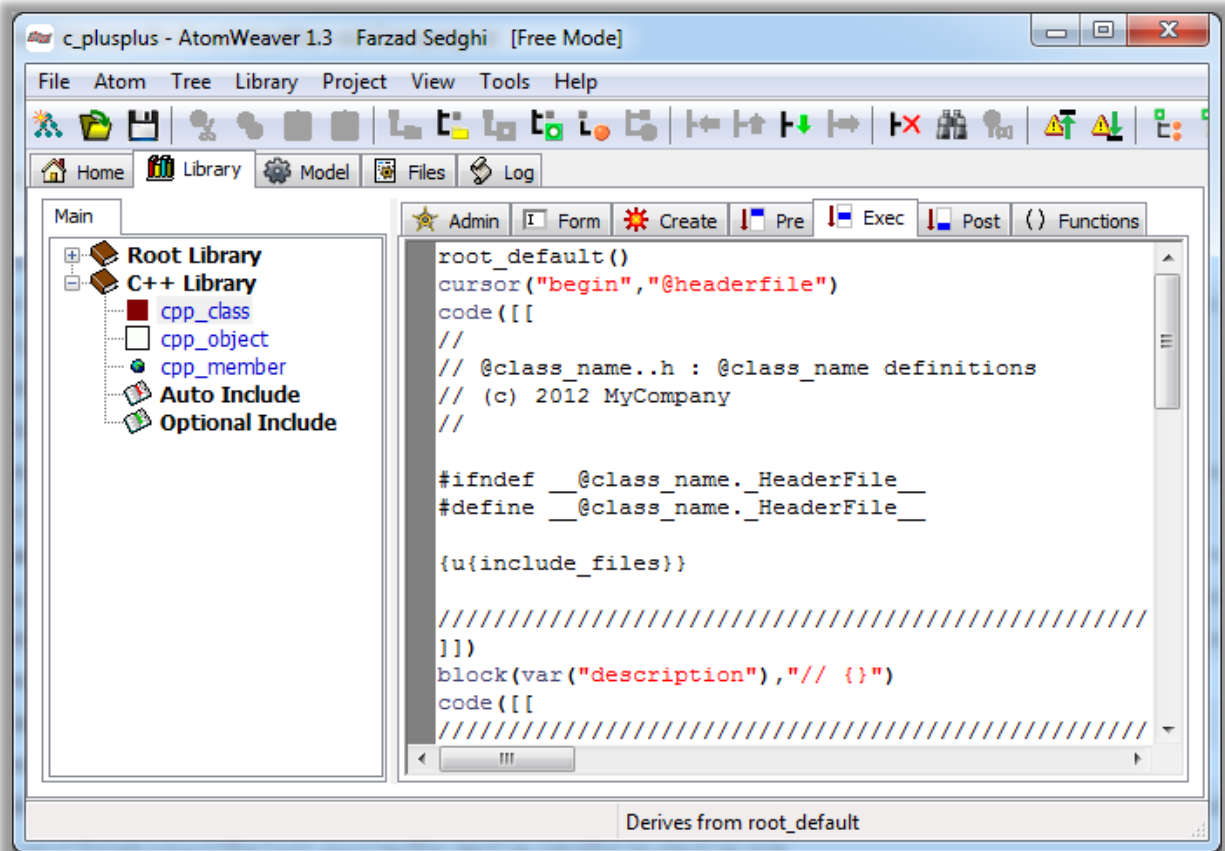


Figure 4-3 *cpp\_class* atom in the tree (left window) and the atom editor with seven categories (right window)

In ABSE, Atoms are organized in trees for any purpose they are going to be used. Creating the model of the solution logic, library of the reusable Atoms, functional requirements, issue tracking or help documentation are all handled by organizing the relevant Atoms in trees. It is also possible to define constraint to specify Atoms under which it is possible to place the current Atom. For instance an Atom for definition of a ‘Method’ for java language should be placed under a ‘Class’ Atom. Constraints not only make finding and using the reusable Atoms faster during development, but also make the development less bug prone.

Atom templates are organized into trees to form Atom libraries, which can be used to find the required atoms in them, based on their label, description and constraint that are defined in their ‘Admin’ section. So when an existing Atom, which exists in the library and hence is used and tested before, provides the same concept as needed in a project, it will be customized using new input parameters and put into the new project.

Atom templates can also be reused to make up another template either by composition of templates or by inheritance. In composition, two or more existing templates are composed to form a larger template and in inheritance. Just like in Java and C++ inheritance, the child inherits the attributes and code from the parent template.

#### **4.2.1.2. Code Generator usage features**

##### **Function requirements**

*Output:* Since it is possible to generate free-form textual output, any kind of code can be generated using Atom Weaver.

*Input:* Atom weaver has its own format for input models which are atoms, so does not support different meta-meta-models for definition of meta-models.

##### **Reliability of the code generator**

Reliability of the tool includes reliability of the templates and models, which are developed in Atom Weaver IDE, and also reliability of the transformation engine which is Atom Weaver's code generator. But usually the task of a transformation engine, excluding the model and templates, is not a complicated part of code generation process. When the templates and models are developed correctly, then the easy part remains to be done by the transformation engine.

While developing the Atoms , the list of warnings and errors is provided by Atom Weaver to help the developer produce less bug prone Atoms. Encouraging the developers to reuse Atoms as much as possible is also an effective way of developing reliable software, which is the main goal of ABSE framework.

Since Atom Weaver is a commercial product, it is expected to have a reliable code generation engine. By the way it should be mentioned that the tool does not have a big or active user community which means there are not much reviews and discussions that can be found about it.

##### **Extendibility of the product software, developed by using the tool**

It is possible to put custom code in between generated code. This feature makes it easy to extend the software developed by this tool. But the custom code should be inserted into the output code,

as input parameters of the model which means no direct change should be done to the generated code.

### **Maintainability of the product software, developed by using the tool**

Debugging capabilities are available in the IDE. Warnings and errors are marked in the list of atoms, which can help the user find the problems before running the generator. Also after generation of code, it is possible to follow the run trace using log tab. traceability is available from log lines to their source atom and vice versa.

In addition, traceability is provided from generated code to its source models and vice versa. Since the environment of code generator development and code generation is separate from development environment, where the code is going to be used later, it is not possible to use traceability in the target environment in order to navigate directly from the generated code line to its source model. Since these two different environments are not integrable, the developer has to switch the environment in order to see the source of generation.

The models are handled in the IDE and it is easy to browse and view different models of a project in different views and tabs and this means high readability of the project and helps with maintenance of the code generator and the product software.

### **Portability of the code generator**

It is not possible to run the code generator in a standalone mode or from command line. This means it is not possible to integrate the code generator with a project's build tools like Ant and Maven.

### **Usability of the code generator**

Since the code generator developed in Atom weaver is not a standalone application and should be run from Atom Weaver IDE, the usability of the code generator is described together with usability of its development environment.

#### ***4.2.1.3. Code Generator development features***

##### **Function requirements**

The modeling in Atom Weaver is based on ABSE which defines meta meta-model for all kind of atoms. So the modeling framework of Atom Weaver is specific to itself and not a general framework or standard.

### **Reliability of the code generator's development environment**

As Atom Weaver does not have a big user community, it is not easy to find reviews and reported bugs. Also looking at the forum in its website, it can be seen the forum is not very active. But as a commercial tool, it is expected to have an acceptable level of reliability. Measuring and testing is needed to investigate reliability of such a tool with a low number of users.

### **Maintainability and extendibility of the code generator**

All the features mentioned for maintainability and improvability of the output software are valid for this section. Also the management of Atoms which helps the developers to discover the correct Atoms for their purposes will encourage reusing the already developed code when developing the system. This not only keeps consistency high in the semantics of the system, but also improves the speed of software development and change.

### **Improvability and connectability of the of the code generator**

Since Atom Weaver has its own special modeling framework and standard, it is not possible to integrate it with other modeling frameworks and/or code generation tools. This also means it is not possible to improve the modeling in meta meta-model level, e.g. adding the possibility to accept meta-models, defined as XML schema files.

### **Portability of the code generator's development environment**

Atom Weaver's installer is only available for windows and it needs visual studio redistributable runtime, which is included in the installer, so installed automatically.

### **Usability of the code generator's development environment**

All the steps from definition of Atom templates, which are meta-models of the code generator, to making the model and generating code are supported by Atom Weaver IDE. The created Atoms

are managed in the IDE by making different indices based on their different features, adding description and specific icons for each Atom. This makes it easy to discover the Atoms later.

Atom Weaver provides its users with several tutorials of different types and levels. Introduction to concepts of ABSE and Lua language, basic and advanced hands on tutorials and sample codes are accessible directly inside the IDE. Different printable versions of such tutorials on different topics can be downloaded from their website<sup>9</sup>. An in context help is available in the IDE which helps the users according to what they do at the time.

Generally it is not hard for a person to start working with the tool, even if they have no experience in code generation or model driven software engineering, since the tutorials and educational materials start by explaining the basic concepts. But since many of the concepts in ABSE are specific to itself, it takes some time for a new user to get familiar to these concepts. For example Lua is a completely new language which may not be an extensive language, still needs to be learnt by user from scratch.

## **4.2.2. Java Emitter Template (JET)<sup>10</sup>**

### **4.2.2.1. *Concept***

In JET, the input model can be XML and EMF models. However there are extension possibilities for introduction of custom models. By default JET expects to open the input model as an EMF model and load it as Java classes in the project. Then it's the task for JET templates to define how these models should be transformed into the final code, which can be any kind of text. JET was developed by IBM as an open source technology.

JET templates are written using JET expressions which resemble JSP expressions a lot. After the templates are defined, a Java class per each template is generated which implements the actual transformation from input models to final code based on templates. It is a possible to provide the end user of the code generator with the templates and allow them to modify the code generator's output for having more flexibility.

---

<sup>9</sup> <http://www.atomweaver.com/download/bookshelf.html>

<sup>10</sup> [30] [29]



There are three types of expressions used in JET which are directives, expressions and scriptlets. Directives are used to specify the settings of the template like the package and name of the generated template class, import classes or the folder containing the generation result. For instance the following directive specifies the package and class name of the output, as well as the needed classes to be imported to the Java template class.

```
<%@ JET package="samplePackage" class="SampleTemplateClass" imports="java.io.*
java.util.*" %>
```

Expressions are used for putting Java expressions within the output. They can be simple as a string, like putting keywords as *class* or *private* to form the body of the class or can be expressions which need to be evaluated at invocation time like:

```
<%= (new java.util.Date()).toString() %>
```

Scriptlets are used to put any kind of java code within the template, the same concept as it is in JSP templates.

The actual transformation is done by java classes that are generated based on JET templates. This makes it possible to use IDEs to debug the transformation code. These classes can be instantiated in java code and by calling *generate()* method of the object and the generated text will be returned as a string.

```
SampleTemplateClass tTemplateClass= new SampleTemplateClass();
String result = tTemplateClass.generate(params);
```

Template implementation classes, *sampleTemplateClass* in the above example, are customizable. The skeleton for all of these classes are based on a default template which provides a method called *generate()*. Customization is done by introducing a new skeleton and including its URI in the JET directive of the JET templates that are going to be customized. This makes it possible to modify the behavior of these classes like by adding new methods, changing the generator or implementing an interface.

#### 4.2.2.2. *Rational Software Architect*<sup>11</sup>

There are different tools to develop JET templates like JET-editor<sup>12</sup> and Rational Software Architect which are based on Eclipse. The latter is an IBM tool which helps the user to develop the JET templates from input models in a bottom-up manner. The tool is basically developed and implemented for rapid development of software pattern implementation. So an example of a proven solution to a known problem is used to create a tool for generating the similar solution implementations, tailored for the same type of problems.

User of this tool provides an example output called Exemplar project to the wizard. Then step by step, the wizard helps with specifying the actions needed for transformation from input to output which finally results in a code generation project including the JET templates.

So after importing the sample project, a meta-model should be introduced. This meta-model can be an XML schema, EMF model or UML, while it can also be composed in the editor. Initially a static template is created per output artifact. The wizard helps the user create the transformation project and the template skeletons by providing outline of the sample output, meta-model and properties tab for templates. The user can change the static elements of the template with references to the input meta-model by selecting the element in meta-model. These references are written in XPath language.

---

<sup>11</sup> [31] [32] [33]

<sup>12</sup> <http://JET-editor.sourceforge.net>

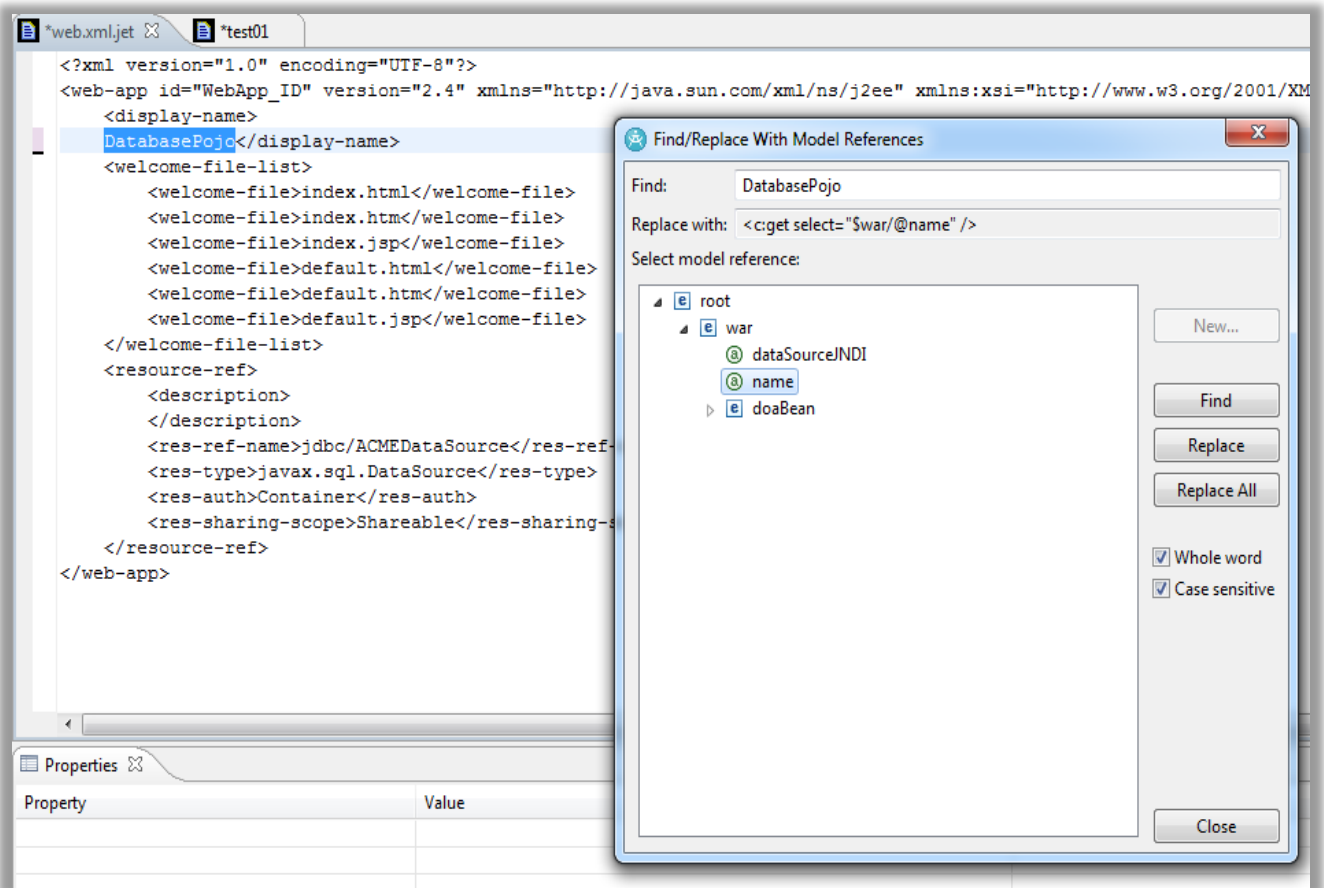


Figure 4-4 replacing JET template static text with a reference to meta-model element, using right-click menu

After associating the templates with meta-model elements, it is possible to further customize the templates by help of the editor. Based on the static values in the templates and the elements of the schema, the editor marks the areas in the template which possibly need changes and also suggests their related replacements. The suggestions will be shown by hovering over these areas and it provides refactoring features like replacing all the occurrences with the suggested meta-model references.

#### 4.2.2.3. Code generator usage features

##### Function requirements

*Output:* Since it's possible to generate free-form textual output, any kind of code can be generated using JET.

*Input:* XML files, EMF-based models like .uml, .uml2 and .emx files, as well as Eclipse workspace are the resources that can be loaded as input to JET and it allows navigation through them by XPath. By default JET attempts to load a resource as an EMF file. It is possible to introduce custom models as well. [15]

### **Reliability of the code generator**

JET code generators should be developed by the user. So its reliability depends on the user's implementation. Still the support in the development environment can increase the quality of the code generator. The most important part of code generator's development is creating the template or JET files. Eclipse version of JET template is more or less like a text simple editor, which helps the user by automatically generating the templates implementation classes. But there are other editors that can be used for editing JET templates and provide more support like code assist and outline. Rational Software Architect by IBM is an example a user friendly and reliable tool for such a purpose.

### **Maintainability of the product software, developed by using the tool**

Traceability from the generated code is not provided out of the box, since the output of the JET generator is free text based on the JET templates. So traceability should be implemented manually for code generation solutions. One possible way to do this is to override the skeleton class of the templates and add the required traceability information to the new skeleton. This way it is possible to leave traces of the related models and templates in the generated code, since traceability information is added to all the templates. Also since there are implementation classes for each template in Java code, it is possible to run the generator in debug mode if needed. Still the development environment will not provide features like navigating to meta-model by choosing its references in templates, or showing all the references to an element of the meta-model.

Another important factor when trying to maintain the final software is the ability for the developer to see the final result as fast as possible, especially in large projects, to be able to test their modifications. Fortunately it is possible to generate only some part of the final code using Jmerge [16] and annotations in the final code. Using Jmerge, the developer does not have to

generate the complete project after small changes in the code generator. Instead they can re-generate only the parts which include the modifications.

### **Portability of the code generator**

Portability depends on whether the templates are also going to be distributed with the code generator or only their Java implementations are intended to be in the final code generator. In the former case, JET engine is needed to be included in the generator project as well to make it possible to compile the templates into Java classes. In this case the generator should run as an Eclipse or Eclipse headless application, since JET is should run as a workspace application in order for the plug-in initialization to take place.

But if the code generator only includes the compiled Java classes, then it has a lot of portability. In both cases above there is no operating system restriction.

JET transformation projects developed in Rational Software Architect, allow accessing the transformation API from source code of other applications.

### **Usability of the code generator**

Basically the input models can be any format, but the default format is EMF. There are a couple of useful tools for creation and modification of EMF models, which can be found among Eclipse projects.

In order to use the code generator inside Eclipse, it is easy to develop the code generator as a plug-in with its own GUI which can be used by the developers to generate code, pretty similar to “New Class” wizard in Eclipse. This is familiar and easy to use for most of the developers.

### **Resource saving**

As mentioned in maintainability, it is possible to find the changes in input model and only generate code for those parts. This is a feature that is provided by EMF and can be used to reduce the response time of the generator.

#### **4.2.2.4. Code generator development features**

##### **Function requirements**

Loading the models in JET is done by *modelLoader* class that can be configured in JET transformation plug-in file. So it is possible to load any kind of data model by introducing a customized data loader. Out-of-the-box JET has data loaders for EMF-based models and XML files.

##### **Reliability of the code generator's development environment**

JET is a well-known code generation tool developed by IBM and is been used for a couple of years in different software development projects. Now it is a part of Eclipse modeling Framework project, as a solution for code generation component needed in Model Driven Development architecture. There are tutorials and articles on using JET that can be found in IBM and Eclipse websites dated back to 2004 which together with number of releases<sup>13</sup> from Eclipse since 2007 shows that it's been used and developed actively through the years. Active user community increases the number of bug reports which makes the tool more reliable.

Rational Software Architect is well-known and high quality software which makes it a reliable solution for creating the transformation project. It helps managing the meta-model and updating the project based on the changes.

##### **Maintainability and extensibility of the code generator**

For extending the templates, Rational Software Architect helps the code generator developer to maintain or extend the templates easily and with fewer errors, since such editors provide the user with code completion. Still it doesn't detect the wrong XPath references from templates to the meta-models before running the transformation.

Possibility to include the templates in the code generator's release enables the users of the code generator to edit the templates in the usage environment. This is a change to extend the code generator that can be used to improve the final software, developed using the generator.

---

<sup>13</sup> <http://www.eclipse.org/modeling/m2t/downloads/?project=JET#archives>

Another extendibility point for a code generator is extending the input model definition, along with their templates. JET does not provide any specific refactoring facility like updating the templates automatically as the meta-models are modified, which makes maintaining the code generator a bit hard. Even in Rational Software Architect, there is no facilities for refactoring e.g. renaming meta-model elements will not update the templates referring to those elements.

### **Improvability and connectability of the of the code generator**

JET can use different kind of models, because it is possible to define customized model loaders for it. This makes it highly connectable with other modeling frameworks. Apart from that, it accepts EMF models out of the box which means JET inherits maintainability and extensibility level of the models from EMF as a rich modeling framework.

There are also some extension points for improving a JET code generator. One extension point is defining custom XPath functions which can be used inside JET templates. These functions which are written as Java classes can help the developer to avoid complexities and long lines of XPath commands. Furthermore it is possible to define custom tags for JET templates. These are also defined using java language and can be included in JET templates.

Rational Software Architect makes it possible to invoke the transformation API from application source code, which makes the transformation solution connectable to other applications, like modeling solutions.

### **Portability of the code generator's development environment**

JET templates can be edited and developed even as simple text files. They can also be developed as JET projects in Eclipse or in Rational Software Architect, both of which are installable in windows and Linux.

But these files should be transformed to Java classes by using JET engine. JET engine should run as an Eclipse application or a headless Eclipse application. So it is not possible to edit the templates and use JET engine as a standalone application to recreate their corresponding Java class.

### **Usability of the code generator's development environment**

Template editor provided in Eclipse JET projects is nothing more than a text editor and does not provide any editing help like code highlight or code completion. But Rational Software Architect provides a more usable environment for developing JET templates. It helps with creating a transformation project, by introducing a sample output project and refactoring it step by step. JET templates are created for each output file, by dragging them to their corresponding meta-model element. The static texts in templates can be replaced by meta-model attributes by marking them and then navigating through the models. But they are not included in code completion, which is usually preferable for developers. Code completion for JET expressions and suggesting replacement of static parts with relevant meta-model references are other usability features of Rational Software Architect.



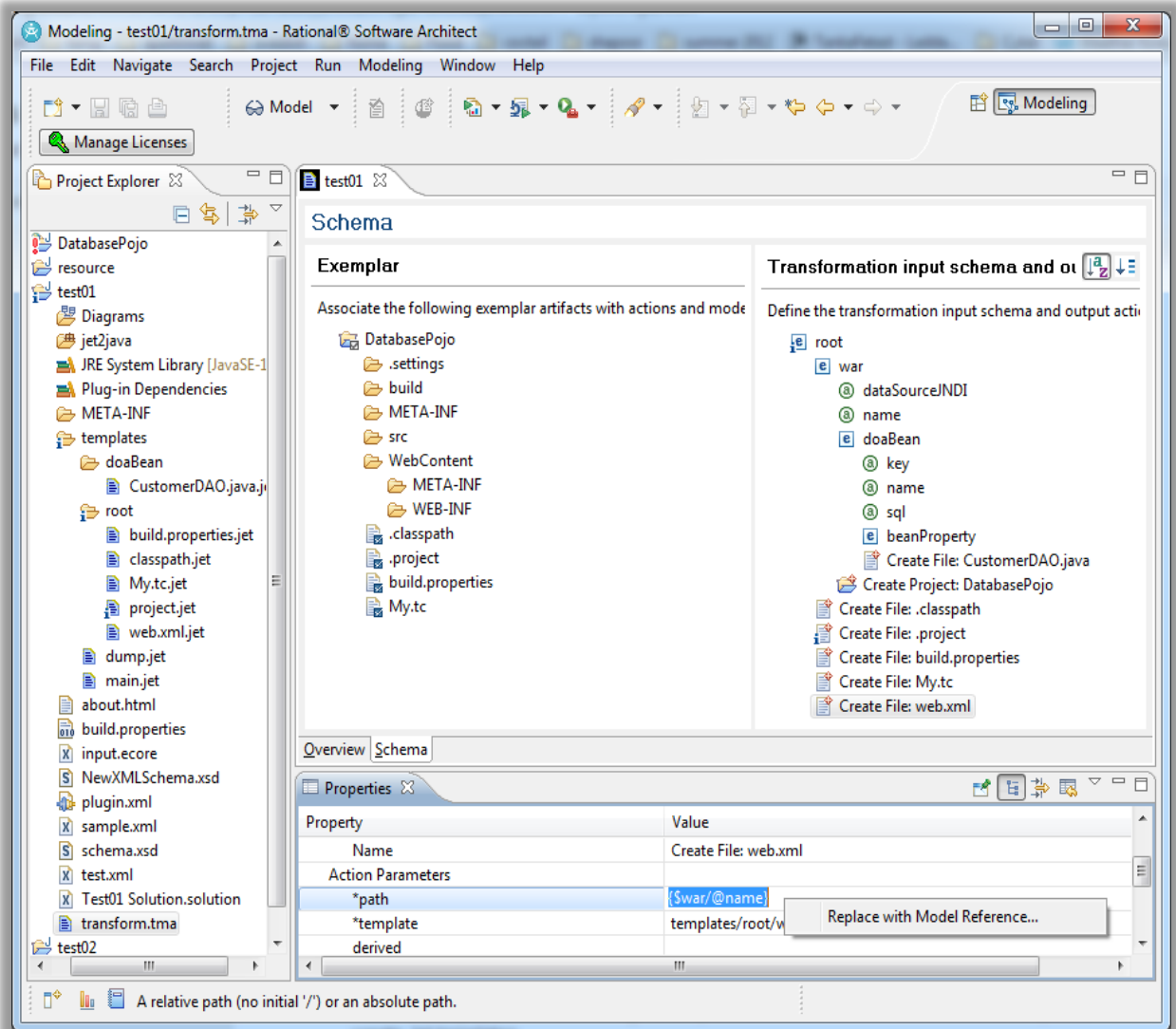


Figure 4-5 Rational Software Architect wizard for defining the meta model and mapping exemplar's elements to meta model elements. Basic versions of JET templates are created automatically, visible in Project Explorer tab

In addition the ability to define custom tags and XPath functions makes it easier for a developer to create JET templates.

Since JET has been around for some years, lots of online reading sources, articles and tutorials can be found on using JET templates for code generation. Most of them are in IBM and Eclipse websites.

## 4.2.3. Xpand

### 4.2.3.1. *Concept*

Xpand is a templates language with some unique features like type safety, polymorphic template invocation, aspect oriented programming, model validation and a lot more. It also comes with a template editor with features like code completion, error highlighting and refactoring. It's a part of Xpand generator framework which provides other languages and features as complementary to Xpand language, in order to cover the entire process of code generation as a very fundamental part of model driven architecture. It was originally a part of openArchitectureWare project<sup>14</sup>, before becoming an Eclipse component.

Different textual languages are provided by Xpand generator framework, which belong to different contexts of model driven software development (MDSD) like validation, meta-model extension, code generation and model transformation. Xtend, Check and Xpand are the languages provided which are used for different purposes but are all based on the same expressions language and type system. This makes them easy to use since there is no need to learn different languages for different purposes, as they are all built around the same expression language.

### **Type System**

As mentioned above, the framework provides a uniform abstraction layer over different meta-meta-models called Type System. Type system makes it easy to refer to different meta-models in code generator, as they are all treated uniformly using this API-based layer. Single or multiple-inheritance is also supported, based on the underlying meta-meta-model.

It is possible to register more than one meta-meta-model to the code generator and use all of them at the same time. Here is where polymorphic model invocation is useful, which matches the elements in models to their templates, based on their most concrete type.

Using this layer it is possible to access built-in meta-models and also different registered meta-model implementations. Built-in types are basic types like object and void, simple types like

---

<sup>14</sup> <http://www.openarchitectureware.org/>

String and Integer and also collection types like Collection and List. In addition, by registering meta-model implementations, it is possible to access the types they provide. Out of the box Xpand has many popular built-in meta-models like EMF meta-models, UML2 or XML schema.

In addition to these built-in meta models, it is possible to implement custom meta-models and register them to be able to use them along with these meta-models.

## Expression language

The expression sublanguage is a syntactical mixture of Java and OCL [13]. It is used in all the three languages provided by framework which are Check, Xpand and Xtend. The language provides literal and special operators for built in types like *Integer*, *Boolean*, *String* which are all ordinary and familiar operators. Also like OCL, it has several special collection operators like *Select* and *forAll*. Here are examples of such operators:

*select*: returns a subset of a collection based on boolean expression

```
{1,2,3,4}.select(i | i >= 3) // returns {3,4}
```

*forAll*: returns true if the boolean expression is true for all the elements in the collection

```
{3,4,500}.forAll(i | i < 10) // evaluates to false (500 < 10 is false)
```

## Check language

Check is a simple language which is used to define constraints over input models. So depending on the defined constraint, the code generation flow will throw a warning or stop with an error, if the input model does not satisfy the constraint. The following Check constraint is defined over “Attribute” meta-class in model “data”, using expression language operator *.length*:

```
import data;
context Attribute ERROR
  "Names have to be more than one character long." :
    name.length > 1;
```

## Xtend language<sup>15</sup>

The other language is Xtend which is a statically-typed programming language that compiles to comprehensible Java source code. It is designed to work great with existing Java APIs and idioms, yet introduces new features to modernize Java applications. Using Xtend it is possible to define rich libraries of independent operations and non-invasive meta-model extensions, based on Xtend expressions or Java methods. Such libraries can be referenced from other languages based on expressions framework, like Xpand templates.

When it's not possible or easy to handle a piece of logic using Expression language, it can be implemented by Xtend as a powerful language and imported as extension into the template. Not only the syntax of Xtend is improved to be more powerful and efficient compared to Java, but also there are some features which make it very useful for code generation use. Template Expressions are one of the very useful features that are available in Xtend. As it can be seen in the following example<sup>16</sup>, the templates are surrounded by triple single quotes and it is possible to use Xpand expression in the middle, which will be replaced by string equivalent of their result:

```
def someHTML(List<Paragraph> paragraphs) '''
    <html>
    <body>
        «FOR p : paragraphs BEFORE '<div>' SEPARATOR '</div><div>' AFTER '</div>»
            «IF p.headline != null»
                <h1>«p.headline»</h1>
            «ENDIF»
            <p>
                «p.text»
            </p>
        «ENDFOR»
    </body>
</html>
'''
```

They can be used to write readable methods for producing parts of code generator templates. A great feature of Template Expressions is smart handling of white spaces which can distinguish between the white spaces belonging to template text and the rest like indentations caused by control structures. For instance in the example above, only the white spaces marked by pale blue are parts of the method's output. This feature can be very useful when writing templates, since

<sup>15</sup> <http://www.eclipse.org/xtend/>

<sup>16</sup> Code from <http://www.eclipse.org/xtend/documentation.htm>

there is no need to worry about the exact output format, since the output will have the same format as the template code which also makes the template easily readable.

## Xpand language

Xpand is a template language to control the output generation. A template is marked up with tags which are executed at interpretation time of the template, during code generation.

```
«IMPORT meta::model»

«EXTENSION my::ExtensionFile»

«DEFINE templateName(formalParameterList) FOR MetaClass»
    a sequence of statements
«ENDDEFINE»
```

As it is shown in this simple example, there are initial tags for importing namespaces or referring to Xtend files for including extensions. Then there is *DEFINE* tag which is used for definition of the template related to a specific meta-model which is *MetaClass* in this example.

*EXPAND* tag is used to call another template inside a template:

```
«DEFINE someOtherDefine FOR SomeMetaClass»
    «EXPAND implClass FOREACH listOfAs»
«ENDDEFINE»

«DEFINE implClass FOR A»
    // this is the code generated for the superclass A
«ENDDEFINE»

«DEFINE implClass FOR B»
    // this is the code generated for the subclass B
«ENDDEFINE»
```

Another interesting point in this example is that if *listOfAs* also has instances of *Bs* and *Cs* which are subclasses of *A*, then their own templates will be invoked for them in the *FOREACH* loop, instead of *A* template which is defined for their super class.

There are other features like support for handwritten code, aspect oriented programming and incremental generation which will be explained in features section.

## Generation workflow Component

The flow of the code generation process is managed from workflow XML. Here the meta-models are assigned to their templates and it is possible to assign more than one meta-models to each Xpand templates. Also the output paths are managed in the workflow component. It is possible to assign names to different output paths and also specify whether the generator should overwrite the existing files. Beautifying the generated code can also be managed from this file, since it is possible to assign “postprocessor” classes to each output and let these classes format - or whatever required processing - the output result, before writing it to file.

#### **4.2.3.2. *Code generator usage features***

##### **Function requirements**

*Output* - Xpand is a free text generator so can be used to generate any kind of code.

One of the important features of Xpand is its ability to generate only the needed parts and avoid generating code, when the result is the same. When one part of the input model is changed, not every output file is necessarily affected by that change. So maybe only a few of them are required to be written and generating and writing the rest is wasting of time.

There are two steps in which this problem can be avoided. First is code generation in which the generator avoids generating the code for a model when there are no changes in that, since its result will be the same. This needs keeping track of the changes in the model by code generator, which works only with EMF meta-models.

Another step is writing to file when the code generator can compare the final result of the code generation, with the content of the target file and avoid writing to it if there is no change in it.

This feature is referred to as Veto strategy in Xpand which has a simple default implementation to avoid rewriting the same content to target file, but still it can be customized if needed.

*Input* - Out of the box, Xpand supports different types of input models like EMF models, XML defined in XSD and also it is possible for the developer to introduce new meta-models and implement them to have their own input format for the code generator.

Below is a list of built-in meta-models in Xpand:

- **EMF meta-models:** EMF registry meta-model and The EMF meta-model
- **UML meta-models:** UML2 and UML2 profile which allows to extend the former
- **XMI [12] reader:** important when working with UML models
- **Java meta-model:** The Java meta-model allows Java classes as meta-types used in meta-models
- **XSD meta-model:** The XSD meta-model provides access to models implemented in the XML Schema Definition language.

### **Reliability of the code generator**

The main features for increasing reliability of the code generation process in Xpand are Check language to define constraints for models and also IDE support for creating models based on their meta-models. Using Check, it is possible to find the errors in input model and inform the user about the problem, instead of generating wrong output. This will avoid creating input models with undesirable data in them. Also when a meta-model is introduced to the Xpand Type system, then the IDE will help the user to create a model based on its definition by providing code completion and showing errors and warnings. So using type safety, creating a model with wrong format is avoided.

### **Extendibility of the product software, developed by using the tool**

By the use of *PROTECTED* tag in Xpand templates, it is possible to let the user of the code generator put handwritten code in the middle of the generated code, without worrying about being overwritten in future code generation runs. These parts of code will be marked as protected areas in the generated files, which have an enable flag. Setting it to false will allow the generator to overwrite the protected area, while setting it to true will prevent the area from being overwritten. The directories that need protection should be configured in workflow file. It is also possible to exclude files from included directories if needed.

### **Maintainability of the product software, developed by using the tool**

Xpand does not leave any trace on the output files back to their source models, out of the box. Still there are lots of ways to add the traceability to generated files. One way could be adding the traceability information to a traceability template for the super class of many models to include

the traceability information to all of them by calling the same template. Meta-model inheritance and template polymorphic invocation together with extensions makes it easy to develop a very useful and powerful traceability mechanism for the code generator.

One important feature that eases maintenance of the target software is speed of the code generation. Veto strategies can help reduce the generation time a lot, especially since they can be customized for each project and result in a shorter generation time. More performance issues are discussed in ‘Resource Saving’ section.

Input model and output readability are also very important for the developer trying to maintain software. Xpand provides the means to improve the readability of both. Input model depends on the input meta-model and a lot of different meta-models are supported by Xpand. For the output code there is the possibility to introduce post processing classes in workflow file for each output, like *JavaBeautifier* which formats the output to be more readable and standard. Also smart handling of white spaces in both Xpand and Xtend makes the output code more readable.

### **Portability of the code generator**

The code generator, which is an Eclipse plug-in project, can be deployed for its user in many different ways. It is possible to run the generator in the IDE e.g. having a plug-in for code generator to run it by right clicking on model or opening a wizard. The code generator is also integrable with ANT and Maven.

### **Usability of the code generator**

It is possible to create an Eclipse plug-in for the code generator project, to run it in the target environment. It can be an option in right click menu on input models, a wizard for code generation or even it can be included in incremental build of the project in Eclipse.

Readability and complexity of the input model are also other usability issues with which the user of a code generator encounters. This highly depends on the modeling framework which is chosen for the code generator. But since Xpand supports a lot of different input meta-models, it is possible to choose the desired modeling framework with which the end users are familiar. Of course this issue also depends on the design of the input meta-model.



## Resource saving

Xpand provides a profiler that can help the developer see the time each Xpand, Xtend and Check method has spent during the code generation process. A report containing call back times and a call graph is provided as the result of the profiling. This can be set through the workflow file and can help the developer find the points with performance problems in the code generator components. There are different features to use to improve the performance in Xpand which are mentioned below.

As mentioned before, incremental generation is a very useful feature of Xpand which is intended to prevent the generator from generating the same stuff over and over. Code generation for the unchanged file can be avoided, as well as writing the same generated result to the file. By using EMF meta-models, it is possible for Xpand to keep track of the changes in each input model and only run their templates if they are changed compared to the previous code generation run. This will avoid generation of the code and subsequently writing the result to file for the entire model, if there is only a small change in one of the input models. This is especially important for large projects since for generation process, each model should be traversed fully and then maybe a post process like beautifier runs on the result and finally it should be written to disk.

Writing back the result can take a lot of time in code generation since reading and writing to file are slow processes. So another possible way of reducing this time is to compare the result with existing file and write it only when they are not the same. This is needed since some classes may stay the same, even though their corresponding models are changed. Xpand has a default implementation for these checks, but it is also possible to implement another if needed. This is especially useful to know since this default simple implementation also has drawbacks related to the cost of loading the existing file in order to process it.

Another performance improvement point in Xpand is use of M2T Backend as the execution engine. Xpand templates and Xtend extensions are not translated to Java byte code which causes slower generator runs. M2T Backend provides a common platform for these languages as well as others like QVT [17] which makes it possible to compile these languages to an intermediate language which is later compiled to byte code.

#### **4.2.3.3. Code generator development features:**

##### **Function requirements**

There are a lot of modeling frameworks and meta meta-models with which Xpand can interact. They are mentioned above in Code generator usage features, under Functional requirements.

##### **Reliability of the code generator's development environment**

Xpand, together with Xtend and Xtext, which is used for creation of DSLs, originally started as a part of openArchitectureWare project by itemis<sup>17</sup> in around<sup>18</sup> 2003 and then moved to Eclipse in 2009. The company keeps releasing the tool about twice a year<sup>19</sup> and has an active user community in Eclipse forums. This shows the tool has a proper level of maturity after around 10 years of development.

##### **Maintainability and extendibility of the code generator**

One important feature that helps the developer to maintain or extend a code generator is the support in IDE for debugging. In Xpand it is possible to put breakpoints inside templates and run the workflow in debug mode. As debug mode for Java, it is possible to see all the variables and their values for the running template.

Refactoring is also supported in Xpand which takes care of propagation of changes from meta-models and extensions to Xtend, Xpand and Check files. As mentioned in Eclipse wiki<sup>20</sup>, some of the refactoring features that are implemented are:

- rename Xpand/Xtend/Check/MWE file (resource)
- rename/extract/move extension for Xtend
- rename/extract DEFINE for Xpand

Syntax coloring, error highlighting, navigation and code completion are some of the other IDE supports that ease maintenance of the code generation.

---

<sup>17</sup> <http://www.itemis.com/itemis-ag/services-and-solutions/eclipse-modeling/language=en/35056/openarchitectureware-oaw>

<sup>18</sup> <http://sourceforge.net/projects/architectureware/>

<sup>19</sup> <http://www.eclipse.org/modeling/m2t/?project=xpand>

<sup>20</sup> [http://wiki.eclipse.org/Refactorings\\_for\\_Xpand/\\_Xtend/\\_Check](http://wiki.eclipse.org/Refactorings_for_Xpand/_Xtend/_Check)

Another useful feature that lets the developer extend the code generator is support of Aspect Oriented programming in Xpand. Using the workflow file, it is possible to package and deliver a written code generator and use it as a black box to generate a different code. This means it is possible to keep an already developed code generator untouched and use it to generate some code, slightly different with the result of that generator. This is done by introducing new templates that wrap around existing templates. The additional template may run before, after or instead of the original template. This is a clean way to extend an existing code generator without changing its code.

### **Improvability and connectability of the of the code generator**

Support of different input meta-model formats or meta meta-models, which is even possible in the same code generator, is an important feature which makes an Xpand code generator improvable. This improvability of Xpand is a result of its high connectability to different modeling frameworks. Imagine there is a code generator in the project which uses a set of input XML files to generate Java code. When later there is a need to generate some more code as well but based on some Java classes. It is possible to add Java meta models to the generator and improve the code generator based on this new meta model and the existing java code. Without such support for different meta meta-models, either another code generator should be written and combined with the current one or the XML models based on java classes should be provided to the code generator, which is an error prone and time consuming act.

It is also possible to define a new meta-modeling language or domain specific language (DSL) for Xpand by using Xtext. Xtext is another Eclipse based tool for development of programming and domain specific language which is highly compatible and integrated with Xpand as a code generation development tool. Using such a tool means it is possible to have a completely new and customized meta-modeling language for creating input models for code generation with Xpand code generator and since this language is developed in-house, extending and changing it which leads to improvement of the code generator is also possible and easy to do.

### **Portability of the code generator's development environment**

Xpand development environment is an Eclipse plug-in.

## Usability of the code generator's development environment

The usability features of Xpand development environment are mentioned in previous sections and will be repeated quickly here as a review. Code coloring, error highlighting, code completion, refactoring and running in debug mode are the features provided by IDE that makes it easier to develop the source code for an Xpand code generator.

A usual complexity in code generation is handling of white spaces in the output code which requires a lot of effort in many code generation tools to spend in order to have a readable output. Both Xpand and Xtend languages provide some features to easily handle this issue which can make the work very easy and also make the templates and extension files readable as well.

Xpand has a complete and easy to read documentation<sup>21</sup> that has examples and sample codes and is available in Eclipse site. There are also useful tutorials and samples on the internet, inside and outside of Eclipse website<sup>22</sup>. Eclipse M2T forum (M2T community forum<sup>23</sup>) is also an active community where many discussions about Xpand can be found there. Since Xpand, Xtend and Check languages have the same base which is the type system and the expression language, it is easy to learn all of them faster and together.

### 4.2.4. Acceleo<sup>24</sup>

#### 4.2.4.1. *Concept*

“Acceleo is a pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard” is the definition for Acceleo in Eclipse website<sup>25</sup>. It's an open source code generator which can generate any kind of text, based on different types of meta-models. Because of usage of standards and open format, it is possible for it to work with a numerous modeling tools and meta-model types as input. Acceleo started in 2005 by French

---

<sup>21</sup>

[http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.m2t/org.eclipse.xpand/doc/org.eclipse.xpand.doc/html/xpand\\_reference.html?root=Modeling\\_Project&view=co](http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.m2t/org.eclipse.xpand/doc/org.eclipse.xpand.doc/html/xpand_reference.html?root=Modeling_Project&view=co)

<sup>22</sup> <http://wiki.eclipse.org/Xpand>

<sup>23</sup> [http://www.eclipse.org/forums/index.php?t=thread&frm\\_id=24](http://www.eclipse.org/forums/index.php?t=thread&frm_id=24)

<sup>24</sup> [34]

<sup>25</sup> <http://www.eclipse.org/acceleo/>

company Obeo<sup>26</sup> and in 2009, as it became an Eclipse project it was recreated from scratch in version 3.0. The latest version is 3.3, which was released in 2012.

In Acceleo all files are modules which consist of static code and also Acceleo language expressions, which are evaluated to output code on code generation. Modules can use other modules inside by importing them and should also specify the meta models based on which they work. The syntax of Acceleo language is an implementation of model to text language (MTL) defined by OMG in MOFM2T standard [18]. Acceleo expressions to access input models are built on top of OCL and use the Eclipse Foundation implementation of OCL language [13]. It is also possible to access Java code from modules by using Java services. Java services are the modules that are created by Acceleo to wrap Java code. So the methods of the java class are accessed in the form of Acceleo queries.

The process of creating a code generator in Acceleo starts by creating an Acceleo project and Acceleo module files using the new project wizard. The meta-model information like meta-model URI and the module's associated result type should be specified. So as a result and as a suggested practice in Acceleo tutorials, there should be one or more modules per output type like class, enumeration or interface.

Acceleo editor provides many features like code completion, syntax highlighting and error detection which makes it easy to develop the modules. Traceability is an important feature in Acceleo which is supported very well by the editor. It is possible to navigate between meta-models, models, modules and the generated code when there is a need to trace from the generated code to its source models and modules and vice versa. Another useful feature is an interpreter view which lets the developer to enter Acceleo expressions and see their result, without a need to create and run a complete code generator.

#### **4.2.4.2. Code generator usage features:**

##### **Function requirements**

*Output:* Acceleo is able to generate all kind of text outputs.

---

<sup>26</sup> <http://obeo.com/>

Acceleo does not have any automatic support of keeping the track of changes in the input model, in order to avoid generation of same content again. One solution suggested by the Acceleo team<sup>27</sup> is to use EMF Compare<sup>28</sup> to find the changed models and run the generators only for them.

*Input:* As mentioned in its documentation, models produced by many different modelers are compatible with Acceleo because of its use of standards and open format. Since it is natively based on EMF, it has better compatibility with tools related to this framework. But since Acceleo supports XMI<sup>29</sup>, all the modeling tools that have the ability to output their models in XMI format are supported by Acceleo.

### **Reliability of the code generator**

Acceleo has an active user group in Eclipse community forum where its possible to find a lot of posts about it. This keeps its bug list updated and also discussed. As an Eclipse project it is easy to find the list of its reported bugs. Also there is an Acceleo community in Obeo network which is a place to find other developers and also get direct response from Acceleo development team. The frequency of its release is quite high which shows its regular improvement and maintenance. For instance in 2011 they have been seven releases for Acceleo 3.

Since Acceleo generators are developed in its own development environment, the features of this environment like code completion, error highlight and traceability will help the developer to create a high quality and more reliable code generator. Another important feature of Acceleo which helps to create a more reliable code generator is the ability to extract out pieces of modules as generation patterns and use them again. It is even possible to contribute to editor's environment and add these patterns to this environment. Since these patterns are used in many places, they can be tested a lot and also because they are applied by the tool, there will be less human error in the code generator.

### **Extendibility of the product software, developed by using the tool**

---

<sup>27</sup> <http://www.eclipse.org/forums/index.php/t/375513/>

<sup>28</sup> <http://www.eclipse.org/emf/compare/>

<sup>29</sup> XMI explained in page 61

There are two ways to support manual code inside the generated code in Acceleo. One of them is to use User-code blocks using *protected* construct to specify the areas in the code which should not be overwritten by the generator. The second way, which is only available for Java code as output, is adding *@generated* annotation to the Javadoc of the elements which should be generated. So any other element that does not have this annotation in its Javadoc won't be overwritten during the generation.

### **Maintainability of the product software, developed by using the tool**

If there are problems in the generated code, traceability can help a lot to find the source of this problem. It can be in the meta-model, input model or templates. Anyway in Acceleo it is easy to find all the sources of generated code in the result view of the development environment.

If the problem in the code is sourced from the input model, after fixing it in the model the code should be generated again. The support for maintenance of the generated code is explained in “Maintainability of the code generator” section.

Acceleo is also compatible with a couple of pessimistic version control systems like Clear Case<sup>30</sup>, meaning that it is possible to allow the code generator to write the read only files after generation, by modifying their lock. This will ease the version control of such projects but it is only available in Eclipse and not in standalone mode.

### **Portability of the code generator**

Acceleo's parser and generator engine can run in standalone mode, as well as in Eclipse. In standalone mode they are Java classes and have no dependency on Eclipse. So they can be called from Ant task and Maven plug-in in different operating systems.

### **Usability of the code generator**

As mentioned in Portability section, the code generator can be used as a standalone application in Eclipse. In this case it is possible to add the code generator as a plug-in to Eclipse and use it without knowing anything about Acceleo. The end user only changes the input model and can then run the code generator to get the code.

---

<sup>30</sup> <http://www-01.ibm.com/software/awdtools/clearcase/>

It is easy to deploy the code generator plug-in as an update site to make it available for the users to download and use it in Eclipse.

### **Resource saving**

There are two important features of Acceleo regarding the speed of code generation. First one is caching the result of queries to use as the result of the same query later. In fact this is a part of OMG definition for M2T that the queries with the same parameters should be saved and the result should be cached. This will increase the response time for the queries.

The other feature is the profiler in Acceleo which helps the developer of the code generator to find the parts of the code generator which are interesting for improving the performance. By setting the run configuration to profile mode, the result of profiling shows the sequence of calls, the number each instruction is called and the time spent for each of them.

#### ***4.2.4.3. Code generator development features***

##### **Function requirements**

It is possible for Acceleo to work with a numerous modeling tools and meta-model types as input. EMF as a general and powerful modeling framework is supported and as a result, any meta-model that can be accessed by EMF is supported by Acceleo.

##### **Reliability of the code generator's development environment**

Active user community and the number of releases since Acceleo started in 2005 shows the tool is now in a high level of maturity after around 7 years.

##### **Maintainability and extendibility of the code generator**

Acceleo provides traceability features in its development environment which are very helpful in finding the sources of errors in the code generator. It is possible to find the templates and models and meta-models which are related to each part of the generated code. The other direction is also available e.g. from a template to its result.



Running in debug mode and using breakpoints to see the local variables is available in the development environment as well. It is good to mention that it is not possible to debug queries and only the templates will run in debug mode.

Acceleo also provides features to preview the result of your changes in the code generator, on the output result. It is possible to compare the current code to the result of changes side-by-side and find what is added, removed or changed in the final result. Partial generation of the result during the development of the code generator, which is referred to as “Incremental Generation” by Acceleo, will let the developer to see the change result without a need to run the complete generation that can take a lot of time in large projects.

In addition there are two ways of overriding the current templates or modules, in order to change some behaviors of the existing code generator. There are two ways to override the behavior of the modules which are static and dynamic override. In static override, one module can be added to project which extends an original module and then it can override the operations inside it. It's obvious that this changes the original code generator in some way. But in dynamic override, the original code generator is kept as a black box and used with changing only some of its original operations. This is a kind of aspect oriented programming in which the new Acceleo project can define dependency to the original code generator and define some new templates which call the original template and override any desired operations.

Refactoring features of Acceleo are also helpful when it comes to extending the code or improving the code quality, which are explained later in the Usability section.

### **Improvability and connectability of the of the code generator**

Acceleo code generation is basically based on EMF. There are different tools based on EMF which allow the developers to define their own meta meta-models or DSLs. This means one can always extend such a DSL, and these changes and extensions to the meta-models can then be used to improve the code generators behavior.

### **Portability of the code generator's development environment**

The development tools are Eclipse plug-in projects and should run in Eclipse.

## Usability of the code generator's development environment

Acceleo provides a lot of features in the development environment which eases the process of code generation. Here is a list of these features, followed by an explanation for some of them:

- Editor features: Code completion, code highlight, error and warning showing, code folding, outline and quick outline, visible white space characters
- Running in debug mode
- Tracing from templates, models and meta-models to code and vice versa
- Refactoring
- Generation patterns
- Prototype based development
- “Interpreter” view
- “Preview” view

Interpreter view is a very useful console where it's possible to enter Acceleo expressions and see their result instantly. Also the “Preview” view lets the developer to see the result of the changes in the modules or models, compared to the current output side-by-side and without a need to run the complete generation and wait for its result.

There are different refactoring features available in Acceleo. Rename is available which makes it possible to rename a module, template, query or variable and all of its occurrences in the workspace. Extract template and query allows the user to create a new template or query from a simple text search. Another refactoring feature is called Pull Up which allows the user to create a module out of some templates or queries and the environment will take care of all the inheritance links.

Generation patterns make it possible for the user to choose among different common code generation design patterns and choose the element on which this pattern should apply. After that, this pattern will be added to code completion list of that element. It is also possible for the users to contribute to the environment and adding their own patterns, so the new patterns can be used like the previous ones during the development.

There are different prototype projects available with Acceleo, e.g. Android application project, which can be used as a start for projects, especially when the developer is new to Acceleo. Refactoring features then become very useful to customize the project and develop and modify the current modules. For instance the static codes can be replaced by model references in the prototype modules and Acceleo replaces all the occurrences automatically.

Prototype development is a good way to start learning about Acceleo. Also learning materials like videos, tutorials and documentation can be found in Acceleo website<sup>31</sup>, Obeo website, Eclipse M2T community forum<sup>32</sup> and Acceleo wiki<sup>33</sup> in Eclipse website. In M2T community forum one can ask different questions on M2T tools like Acceleo from other users as well as Acceleo development team members, although the responses are not always very fast.

## 4.2.5. AndroMDA

### 4.2.5.1. *Concept*

AndroMDA<sup>34</sup> is an open source model driven software development (MDSD) framework. It takes any number of input models, which are UML models stored in XMI format, and generates the output code depending on the plugins set for it. These different plug-ins can be used to generate components for different languages like Java, .Net, HTML and many more, in addition to the customized plugins which makes it possible to create any kind of output. The goal is to take the input models and generate fully deployable applications and components. Currently it's mostly used for generation of J2EE applications.

There are two steps of transformation from input model to the generation code. First step is transforming the UML models, as Platform Independent Models (PIM), into Platform Specific Models (PSM) like Spring<sup>35</sup> service classes, and this will be done using Cartridges. Cartridges are the plug-ins, which are specific to the each target platform. The next step is generating final

---

<sup>31</sup> <http://www.eclipse.org/acceleo/>

<sup>32</sup> [http://www.eclipse.org/forums/index.php?t=thread&frm\\_id=24](http://www.eclipse.org/forums/index.php?t=thread&frm_id=24)

<sup>33</sup> <http://wiki.eclipse.org/Acceleo>

<sup>34</sup> <http://www.andromda.org/>

<sup>35</sup> <http://www.springsource.org/>

code based on the PSM which is done by the templates, associated with each cartridge. Java, Spring, Hibernate and Web Service are some of the cartridges available out of the box.

It is also possible to write custom cartridges to generate any kind of output. A cartridge is a bundle of files including a mapping file called cartridge descriptor. The descriptor is an XML file for mapping the classes of the UML class diagram to the actions which should be performed by AndroMDA based on their stereotype. The actions are also bound to the templates, which are bundled with the cartridge as well.

The prepared cartridges in AndroMDA, like Java and Hibernate, are configured to enable AndroMDA produce a ready to deploy output. This configuration information is asked from the user when setting up the project using command line. So these cartridges are meant to be used to create a complete tier and it's mostly useful when the target product can be generated mostly by using one or more of the prepared AndroMDA cartridges. Using them it is very fast and easy to create the components without much development. But if the final product needs a lot of customized components, using custom cartridges does not seem to be the best solution since there is not much support from AndroMDA for cartridge development. There are documentations and tutorials available which show how to do so, but no special development tools or IDE is provided and this makes the development hard to do. Customized cartridges are provided for compatibility and flexibility, but using them as main components of a system means a lot of development effort.

The templates are written in Velocity Template Language<sup>36</sup> (VTL). Velocity is another java based open source template engine by Apache. VTL is a simple and powerful template language which allows reference to the objects defined in Java language. So basically AndroMDA gets the UML models as input and based on the cartridge description, provides the Java objects for Velocity templates to generate the output code.

---

<sup>36</sup> <http://velocity.apache.org/>

#### ***4.2.5.2. Code generator usage features***

##### **Function requirements**

*Input:* As input model, AndroMDA only accepts UML models which are saved in XMI format.

*Output:* Out of the box it includes cartridges for generation of the following:

- BPM4Struts
- jBPM
- JSF
- EJB
- EJB3
- Hibernate
- Java
- Meta
- Spring
- WebService
- XmlSchema

Any kind of output can be generated by AndroMDA, by writing a new customized cartridge.

##### **Reliability of the code generator**

As claimed in their web site, AndroMDA is always under regression test on a daily basis and it is possible for the users to access daily builds as well as the stable releases. Stable versions are released monthly. Also the development of AndroMDA is under control of a few core developers for providing proper maintenance and support.

In early versions of AndroMDA there was not the possibility to perform any check on input models in order to verify their data or format validity. But now using meta-facades<sup>37</sup>, it is

---

<sup>37</sup> <http://www.andromda.org/docs/andromda-metafacades/index.html>

possible to access the information in the UML models in an object oriented manner. Meta-facades are MOF modules that are aimed to hide the complexity of the underlying meta-model and ease the access and control over input models from templates. Using these facades it is possible to put any kind of check over the input model, which will improve the reliability of the code generator.

But when it comes to use customized cartridges, the reliability of the generator will depend on the quality of its development. AndroMDA does not provide any IDE for development of cartridges. So their development is not in a controlled way and is bug prone. There are no features like debugging or editor supports like syntax highlight and error detection.

### **Extendibility of the product software, developed by using the tool**

It is possible to add custom code to the generated code when needed. AndroMDA provides special extension classes that can be used for adding custom code whenever needed. So in order to modify an application, first the model is changed and new code is generated, which will not override the custom codes and the custom code can be added or modified manually.

### **Maintainability of the product software, developed by using the tool**

All the generated files are commented with their related templates and cartridges as the parts of the code generator, as well as the input model class and stereotype. This helps the developer to know where to look for faults when bugs are found in the generated code.

But since there is no IDE for code generation and development of pieces of software using AndroMDA, the developer has to open the files manually and search through them in order to find the exact point of input model or code generator related to each output point. This is also a problem when a customized cartridge is being developed, since the developer does not have any special support for development like debugging, output preview, code highlight and many more, making it hard to maintain or develop these pieces of code.

## Portability of the code generator

AndroMDA needs a JVM to run, which means that it is operating system independent. It is possible to use it with both Ant task and Maven, although Maven is highly recommended since most of AndroMDA tools come with a Maven plug-in.

## Usability of the code generator

Working with AndroMDA is command based. Installation of AndroMDA, creating a new project and code generation are done in the command line. Here is a command for downloading and installing AndroMDA, using Maven:

```
maven plugin:download -DgroupId=andromda -DartifactId=maven-andromdapp-  
plugin -Dversion=3.3
```

In order to generate a new AndroMDA application, which is done by another command, AndroMDA will ask a couple of questions like the name of the project, persistence cartridge to use, whether it needs a web application or web service, and by answering these questions a project will be initiated. So a UML modeling tool like MagicDraw<sup>38</sup> should be used in order to provide and edit the input model and then code generation and other stuff like handling project dependencies are done via the command line.

AndroMDA provides a couple of tutorials, sample projects and good documentation in its website which helps with understanding the concepts and start to learn and use it. As claimed in their website, they have an active and responsive mailing list and the users can get response for their questions very fast.

## Resource saving

By setting the flag “lastModifiedCheck” to true for the models in AndroMDA configuration XML, it is possible to tell the generator not to generate code for the models that are not changed, compared to the previous generation. This will save a lot of time and is useful during the development and maintenance.

---

<sup>38</sup> <http://www.nomagic.com/products/magicdraw.html>

#### **4.2.5.3. Code generator development features**

##### **Function requirements**

AndroMDA environment only accepts XMI files, which represent UML models, as input model and that means it works with UML meta-models.

##### **Reliability of the code generator's development environment**

There are no special facilities provided for supporting code generation development like a template editor or modeling tool. So reliability for the development environment is not relevant in the case of AndroMDA.

##### **Maintainability and extendibility of the code generator**

AndroMDA does not provide any development environment like an IDE which provides development features. The developer only has access to the input UML models and output generated code and the traces from each output to its input models and the corresponding cartridge. This gets worse when a custom cartridge is used and this cartridge can be the source of the fault as well, but there is no debugger or error viewing to help the developer find it.

A useful feature in time of maintenance and development is the possibility to set “lastModifiedCheck” to true. This causes the generator to generate code only for the modified models and at least reduce the code generation time, so the developer can fix the problem in more iterations.

##### **Improvability and connectability of the of the code generator**

AndroMDA is only capable of working with UML models, so it is not possible to add support of other input meta-models. This will prevent improvement of the code generator system for supporting different kinds of input models. But from another point of view, using UML as input model and then using a two-step generation is a good practice to keep the conceptual design of the system apart from its platform specific design which is very useful when it comes to improve the system. Such a separation lets the developers to improve the conceptual model without any technological concerns and since everything is sourced from UML models, the designers and business experts have a complete and readable view of the system.



## Usability of the code generator's development environment

There are no special features in provided for AndroMDA and in fact, there is no development environment provided to code generation developers other than UML editors and the command line.

What makes working with AndroMDA easy is its simple concepts and that it is well designed. So normally there are not many development features needed to use AndroMDA for developing a new system, since most of the work is done by the system and the developer should focus on design and preparing the input UML models. But it's not the case when it comes to develop or maintain the customized parts, since they are not under any control and can go wrong at any place and the developer will not be notified about them until running the generator.

Documentation, tutorials and sample projects, articles and descriptions are provided on AndroMDA website. These are very useful and well written and easy to follow for learning and starting to use AndroMDA. AndroMDA does not have very complex concepts or design which makes it easy and fast to learn.

### 4.2.6. MetaEdit+

#### 4.2.6.1. *Concept*

MetaEdit+<sup>39</sup> is a Domain Specific Modeling (DSM) environment which makes it possible to define a graphical modeling language and make use of it to generate the artifacts of the product like source code, documentation and test code. It is an integrated environment which provides all the means to start a DSM solution from scratch by defining the modeling language, modeling of the system and generate up to 100% of the code in the end. It is a commercial solution from MetaCase which is a leading provider of DSM environments, started in 1991. According to the reviews and testimonials of different experts ranging from their clients to famous MDSD<sup>40</sup> experts, it is one of the most powerful, easy to use and flexible DSM solutions currently available for software development. It is used in different industry fields like communications, financial services, medical and automotive.

---

<sup>39</sup> <http://www.metacase.com>

<sup>40</sup> MSDN - Model Driven Software Development

## **Process**

The process of developing a new DSM definition, which is the definition of meta-models, starts by defining the concepts of the modeling language like objects, relations and roles. Apart from data modeling objects, behavior can also be captured in meta-model definition by use of roles and relation and define their bindings. So the language can provide the means to describe the events and states of the system as well. Having the capability to capture both data and behavior in the model means the ability to generate the complete ready to run software without any manual implementation. So it all depends on the language definition capabilities in which MetaEdit+ is one of the leading solutions. The next step after defining the concepts of the language is to design the graphical symbols for concepts and rules to be used later for modeling the system.

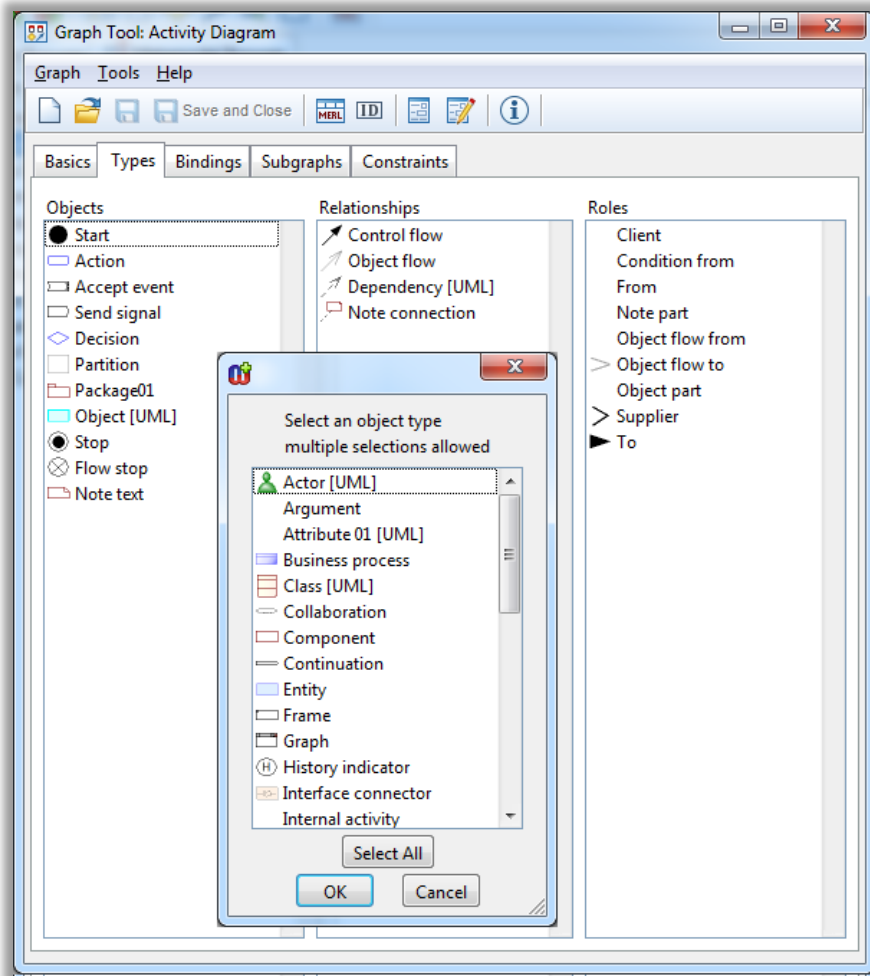


Figure 4-6 definition of meta-model, adding a new type to UML activity diagram as an example

After the definition of the meta-model, the modeling is done using the graphical user interface. When model is ready, it's time to use the code generator for the generation of the artifacts, based on the model. There are a number of prepared generators for generation of source code in different languages like Java and C++, documentation and exporting the model to HTML and etc. which can be used to generate different parts of the output by the use of different parts of input model. So each model which may be composed of different diagrams like class diagrams, state machines and sequence diagram in UML, can use different code generators for generation of system components, APIs, Documentations and test cases. The prepared generators are developed based on the prepared set of different modeling languages that are provided by MetaEdit+. So for instance with UML as input meta-model, generators for C++, Java, Smalltalk, Delphi and CORBA IDL are available out of the box, in order to generate the source codes.

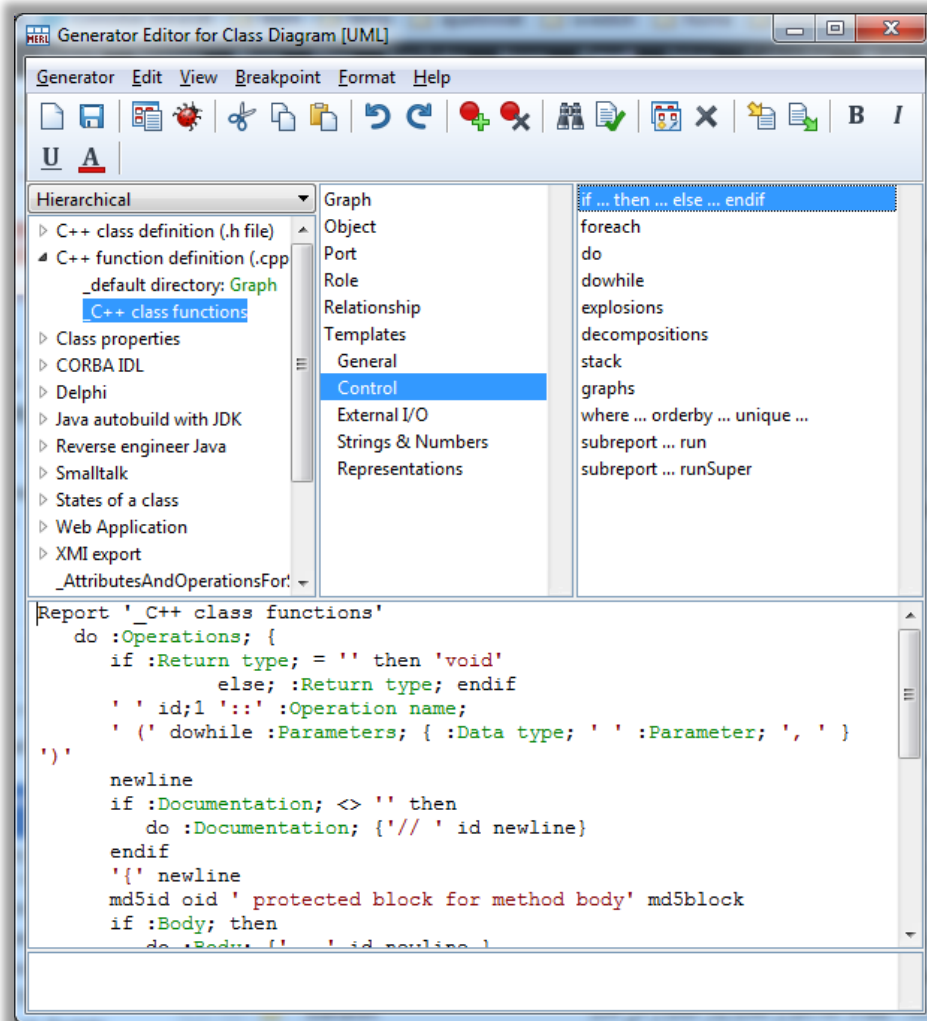


Figure 4-7 A template for C++ function written in MERL is open in Code generation editor window

In addition it is always possible to develop custom generators for any kind of output. MetaEdit+

Reporting Language (MERL) is the template language which is provided for creating code generators. The language, as other template languages, allows navigation through the input models, extracting model data and generating the output text. Development is done in the generator editor which provides features like debugging, traceability and editing features like code highlighting, error viewing and templates of the code generation language. It also provides access to model elements in a well organized element list which can be used to import different parts of input model to the code generator's code, as shown in Figure 4-7.

It is also possible to use other generators instead of MetaEdit+ generator by accessing the model through API or intermediate models like exporting to XML. But using MetaEdit+ generator has the benefit of integration of the meta-model with the code generator editor. So any change of meta-model is reflected in the code generator editor and can be used to extend and modify the current generator. So it helps to develop the language and generator definition in an agile and iterative manner and the models and the generated output are always in sync, which is of course one of the main purposes of MDSD.

#### ***4.2.6.2. Code generator usage features***

##### **Function requirements**

Input: It is possible to define any kind of meta-model for input model in metaEdit+. Out of the box, there are a couple of model definitions that are provided as domain modeling languages, such as:

- **Unified Modeling Language (UML):** An implementation of OMG's Unified Modeling Language 2.0
- **Business Systems Planning:** An implementation of IBM's BSP language.
- **Business Process Modeling Notation (BPMN):** An implementation of BPMN v1.0
- **Structured Analysis and Design:** An implementation of the traditional SA/SD

In addition there are prepared modeling languages for specific domains like telecommunication, phone applications and digital wrist watch applications. Based on the existing meta-models, one can import models to MetaEdit+ as XML files.

Output: It is possible to generate any kind of textual output using the code generation, since MetaEdit+ supports the development of custom generators. Also out of the box there are code generators for some of the existing meta-models, which can still be modified if needed. Some are as follows:

- C++, Java, Smalltalk, Delphi and CORBA IDL for UML

- SQL for Entity-Relationship Diagram

There are also some language independent generators that can be used to study the design of the system, like exporting the graph to HTML or list of objects, which use meta data to generate the output.

### **Reliability of the code generator**

MetaCase is one of the leading companies in DSM development area and having lots of well known customers<sup>41</sup>, is a sign of reliability of their commercial software.

From development point of view, model based development method together with a user friendly graphical interface for generator development is effective at reducing the errors in the software development process. It is also possible to define constraints on input models which can avoid invalid data for input models.

The ability to use different meta-models in the same model or graph is a feature that can increase the reliability of the software, since it encourages reuse of the meta-models. In addition it is possible to call other generators in the repository of MetaEdit+ which are not necessarily the generators for the types of the same meta-model. This means one can keep all the generators of all the types together with the type definitions and reuse them whenever needed by calling them, instead of rewriting a new untested code generator for that type.

### **Extendibility of the product software, developed by using the tool**

The integrated environment of MetaEdit+ provides the possibility for changing the input model in the graph editor and seeing the result immediately by running the code generator. Of course it is the goal of a generator to support such an easy change but a good feature in MetaEdit+ is the ability to run the system and see the graph changes accordingly at the same time. For example running a generated code for a state diagram, it is possible to see the change of states and values in the diagram, in a graphical way, which helps a lot to do the correct changes in the input model.

---

<sup>41</sup> <http://www.metacase.com/cases/testimonials.html>

Protected blocks can be used to preserve the hand-made changes in a file. A unique ID should be assigned to such blocks, since an MD5 checksum is calculated based on UTF-8 presentation of this ID to enable the generator to identify the block in different file types and platforms. So when a block is protected and its ID matches with the ID of the same block on the disk, the generator won't override that block.

### **Maintainability of the product software, developed by using the tool**

In the debug mode of the code generator editor it is possible to see the generation of the final code step by step by using breakpoints. The developer can see what part of the template and which model elements are involved in each generation step and view them as diagram or matrix if needed. Also traceability is provided between generated code and input models and its possible to navigate to the models by double clicking on their name in the output code.

Since MetaEdit is a graphical modeling environment, the input model is always in a readable format.

The usability features of the code generator editor are also important for maintainability, which are explained later.

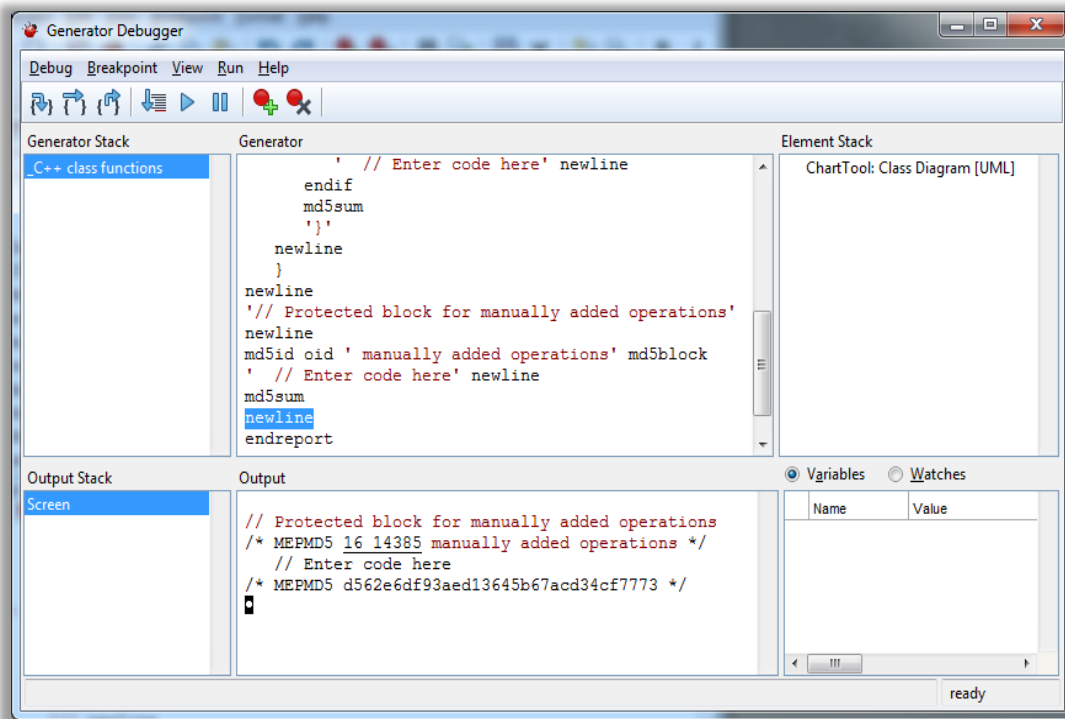


Figure 4-8 code generator running in debug mode, output generated and shown below while moving on template lines

The ability to run the generated code and seeing the changes in the model in a graphical way is a helpful feature for changing the input model in an easy and correct way.

The code generator editor and the meta-model definitions are integrated in a way that the changes in the meta-model are reflected in the generator editor. So the new elements or element changes are visible in the editor and can be used to update the generator accordingly.

Also the documentation generating features in MetaEdit+ makes it possible to always have up to date documentation of the input model, which is useful for the process of changing the software.

### Portability of the code generator

MetaEdit+ is compatible with Windows, Linux and Mac OS X and it is possible to call the generator from the command line for different models, which makes it integrable with build tools. Of course another way to run the generator is to call it inside MetaEdit+ environment.



## Usability of the code generator

Inside MetaEdit+ environment, all the code generators which have application for a specific model are easily accessible. So by choosing the generation option for a folder containing a couple of graphs, all their related code generators are listed while the same action for only one of them, lists only its own related code generators. This is done based on all the types used in the diagrams and the code generators.

MetaEdit+ has generators for documentation generation which are available for all the projects and models, no matter what modeling language they use. Design specifications can be exported to HTML or word format easily by running these generators. These specifications include:

- Diagram representation of the model
- List of graph content like object names and their documentation
- Detailed description of the objects
- Graph information like name, status and users working it. This feature is available for the meta-models which include the corresponding properties.

These documents can help the user of the code generator a lot when it comes to edit the input model to get the desired output.

## Resource saving

In order to generate code, the user should specify the code generator as well as the input graph, before starting the generation. This means it is possible to run exactly what is changed, without processing all the models. This reduces the time for a developer to wait for the result of the generation.

### 4.2.6.3. *Code generator development features:*

#### Function requirements

Importing models in XML format is supported. These models should comply with structure of GOPPRR<sup>42</sup> meta-modeling framework, developed by Metacase for this purpose, which is used

---

<sup>42</sup> Graph-Object-Property-Port-Role-Relationship

inside MetaEdit+ to define the meta-models. One point is that since there is no meta-model information included in the exported files, the user who is importing the data should make sure that the meta-model for the input model exists in the repository; otherwise its parsing will fail.

Exporting and importing the meta-model definitions in XML format is also supported. As this definition is itself a model or type definition, describing a specific language, it should comply with language definition meta-model which is GOPPER.

### **Reliability of the code generator's development environment**

MetaCase is one of the leading companies in DSM development area and having lots of well known customers is a sign of reliability of their commercial software.

### **Maintainability and extendibility of the code generator**

Running the code generator in debug mode is very useful for finding the source of errors in the code generation, since it is possible to move through the templates step by step and see the generation variables as well as the output result which will be generated step by step. Traceability is also available from code generation editor, back to the meta-model elements.

The IDE provides a graphical interface and forms to add new elements to the meta-model. These changes are reflected automatically to the code generator editor environment which can be used to update and extend the generator accordingly. So it's possible to import the meta-model elements from a list of elements to the code generator's source code. This list includes the templates of MERL language commands as well. This is shown in the two rightmost windows at the top, in Figure 4-7.

The other usability features of the development environment affect the maintenance and extending process as well.

### **Improvability and connectability of the of the code generator**

The possibility to define new meta-models and also using more than one modeling language in a project is what provides a lot of possible improvements for the MetaEdit+ projects. So not only are the meta-models extendable for improvement of the system, it is also possible to change the

meta-model format or add one or more meta-models to the same project. This covers the different areas of the business with languages specialized for those fields.

Another connectability feature which has effect on system's improvability is the possibility to access the models of a system via web service API and SOAP protocol. So the application which wants access to design data should have an implementation of the SOAP interface defined by MetaEdit+ in a WSDL [19] file. Using this API it's possible to connect to MetaEdit+ and request different design data.

Some of the powerful object-oriented principles can be used when writing the code generator. When a meta-model inherits another, one can define its own specific code generator for it, instead of using the one defined for the super meta-model. Also if there is a code generator that is useful for more than one meta-model, it can be defined for the superclass of all of them, instead of repeating its definition. This way of organization makes it possible to reuse the same meta-models and pieces of code generators as much as possible, which makes the development process easier and more reliable.

### **Portability of the code generator's development environment**

MetaEdit+ can run in Windows, Linux and Mac OS X and it is possible to connect to it from any platform via a SOAP web service interface, in order to access the design data.

MetaEdit+ is integrable with Eclipse IDE, in order to use it as a programming environment. MetaEdit plug-in for eclipse allows navigation of the input model and interact with MetaEdit like opening the graphs in it.

### **Usability of the code generator's development environment**

When creating and modifying meta-models and models, MetaEdit+ provides different graphical features to ease the process. It is possible to view and edit the graphs models as diagrams, matrices and tables. The graphical interface increases the speed of development, software quality and also readability of the models and meta-models. The graphical icons and shapes for each meta-model can be customized and defined by the user when creating the modeling language. In addition all the Projects, meta-models and models are well organized and accessible very easily.

The code generator editor provides a debugger, a result view, traceability and editor features to help the user with the development process. It has a hierarchical list of all the elements available in the repository, as well as a list of MERL commands that can be imported to the generator code instead of typing by hand. Code highlighting and error listing is also available.

One point about development of models and templates in MetaEdit+ is that the method of development is not similar to what is provided by IDEs like Eclipse for development of text files. Models are developed graphically and templates are developed in a text editor, without features like code completion based on models and commands, which is a familiar feature for many developers. Instead they should navigate through list of commands and meta-model elements in a separated list and import the needed words to the template.

Another point is that MetaEdit+ only works with graphical models, while many developers prefer to have text models, like DSLs instead, at least when creating the model.

The set of predefined meta-models and their code generators make it easy for the users to start a project using those meta-models and generate the desired code in a very short time. This takes a little time to use or maybe some additional time for modifying the existing code generators. Anyway existence of these meta-models and code generators are helpful for a new user to learn the usage faster.

MetaCase provides lots of tutorials and videos about their product, as well as complete and detailed user guides and documentation. In addition it is possible to request MetaCase for free online meeting and demonstration of MetaEdit+ DSM environment. A 30 days full featured trial version of MetaEdit+ is also available to download on MetaCase website.

## 4.2.7. FreeMarker

### 4.2.7.1. Concept

FreeMarker<sup>43</sup> is a template engine which can be used to generate all kind of text output. It is a Java class library that can be accessed from Java code in order to generate code based on Java objects as input model. So inside the Java code for a code generator, first the models are created in the form of Java objects, and then the templates which are written based on the structure of these objects are applied to these objects to generate the output code.

FreeMarker templates are written in FreeMarker Template Language (FTL). Templates consist of text, tags similar to JSP or HTML tags and also instructions called interpolation, which are expressions that will be evaluated and replaced with their result.

Interpolations can be used for fetching data from model objects and doing numeric or string operations on them if needed, in order to find the result value for that place in template.

The data objects should implement *freeMarker.template.TemplateModel* interface, to make them applicable in the templates. Although if standard Java objects and collections are passed to FreeMarker templates, they will be implicitly converted to proper format accepted by templates. This is called ‘object wrapping’ in FreeMarker. In order to use other classes as part of the input model, like *java.sql.ResultSet*, a new class should be implemented to wrap this class and also implement *freeMarker.template.TemplateModel*, which makes it possible for the template to access its content.

A useful capability in FreeMarker is its powerful XML processing. It is possible to use FreeMarker Java libraries to easily load an XML to object models and then access its data from template. The XML will be presented as a DOM tree, so from the templates it can be accessed as a tree of nodes, as described by W3C DOM [20]. FTL provides both imperative and declarative processing of XML. So it is possible to iterate through the nodes and elements of the DOM tree and explore it to generate the output. It is possible to define variables and functions and control the flow of the code generation by using imperative constructs like loops and if-else statements.

---

<sup>43</sup> <http://freemarker.sourceforge.net>

Imperative approach is familiar and easier to adopt by most of the developers, who develop with imperative languages like Java. This can be one reason why a developer prefers using FreeMarker for transforming an XML input to an output, rather than using XSL. Although XSL is the standard for XML transformation and is a very powerful language, its complexity and declarative nature makes it a complicated tool for many developers. But it is still possible to process an XML in a declarative manner in FLT, since there are many cases that declarative approach makes the conversion process easier. An example is when the same element can appear as the child of many different elements.

Some of the main concepts, which help making modular and readable templates are Namespaces, Macros, Functions and templates including other templates. Macros make it possible to define user defined directives that can be called by their own tags. Functions are like Macros but return a value and also are called like expressions in the template. Templates can also include each other which means the output of included template will be placed in the same place that the include tag was called. These features together with the namespace capability, help to create a code generator in a modular manner, especially when it's a large project.

Although FreeMarker is a template engine for generating free format text, but it has some built-in features for supporting web applications. Handling HTML-escaping, integration with “Model 2” [21] Web Application frameworks like Struts and supporting JSP taglibs are the reasons why it's advertised as “Web-ready”.

#### ***4.2.7.2. Code generator usage features***

##### **Function requirements**

*Input:* The input models for the FreeMarker generator are Java objects and it does not come with any modeling environment or meta-model support. So whatever is the main model, it should be first converted to Java objects in a Java program and then used by templates. XML is also accepted as input and FreeMarker can parse it to DOM objects and pass it to its templates and the templates have the ability to process this DOM tree in both imperative and declarative way. It should be mentioned that there is no support for defining the meta-model for FreeMarker to

check the input model or provide features like code completion. So if there is any controls related to the meta-model, it should be handled manually in the Java code that calls FreeMarker.

*Output:* FreeMarker is capable of generating any kind of output. There are some additional built-in features like HTML-escaping that makes it very useful for generating HTML pages.

### **Reliability of the code generator**

Since FreeMarker is only a template engine and does not have its own modeling environment, there is no support in it for checking the correctness of the input model, based on its meta-model. Such types of checking should be handled in the program that calls FreeMarker like the code generator written in Java which uses FreeMarker for generating the code.

There are tools for development of FreeMarker templates which provide support such as syntax highlighting and syntax error indication and code completion based on Java bean properties and names and also macro names in the templates. Using such tools can reduce the errors in the code generator. Eclipse, NetBeans<sup>44</sup> and IntelliJ IDEA<sup>45</sup> have this support as plug-ins or out-of-the-box.

The FreeMarker user community is an jgrowing and increasing number of Servlet frameworks which adapt it as their view layer shows it is a successful and reliable tool.

### **Maintainability of the product software, developed by using the tool**

There is no built in mechanism in FreeMarker for traceability. FreeMarker is just a template engine and it is connected to the models through Java objects. So there is no direct link between original models and the generator and if there is a need for tracing from the code to models, the required data should be included in the object properties and then reflected to the templates manually. Also the IDE support for template development is restricted to editor features like code highlight or syntax errors and in the best case, checks are only done against JavaBeans properties. This means there is no possibility to navigate from FreeMarker tools back to the models during the development process.

---

<sup>44</sup> <http://netbeans.org/>

<sup>45</sup> <http://www.JETbrains.com/idea/>

When running the FreeMarker transformation, it is possible to log the error messages instead of rendering them into Output file, by setting system properties.

FreeMarker provides debugging API, by which it is possible to debug execution of the templates through network (RMI). So the third-party tools can use this API to provide debugging in template development. IntelliJ IDEA editor and also JBoss editor for Eclipse does not provide the possibility to run the transformation in debug mode.

### **Portability of the code generator**

FreeMarker is a jar file that can be included in the project and does not need any special environment to run. Also there is an Ant task included with FreeMarker to use it for applying templates to XML files.

### **Usability of the code generator**

FreeMarker is called from Java code or the command line like Ant tasks. So there is no solution out-of-the-box to support creating a user friendly interface for using the code generator. Also since there is no special support for any modeling framework, it does not provide any features for the user to create or edit the input model.

### **Resource saving**

As claimed in their website, the generation process can easily compete with Java and C#, regarding both time and memory consumption.

## ***4.2.7.3. Code generator development features***

### **Function requirements**

Input: FreeMarker is only a template engine which gets Java objects and merges them with the templates written based on those objects. There is no specific input restriction since the input should be converted manually to java objects. Although it provides XML parsing facilities to automatically convert XML files to input Objects.



Output: any kind of textual output is possible to be generated with FreeMarker, although there are some extra supports for HTML output, which is the original purpose of FreeMarker project.

### **Reliability of the code generator's development environment**

There is support for FreeMarker in different IDEs, which is provided by reliable publishers. For example IntelliJ IDEA has FreeMarker Editor out of the box and JBoss<sup>46</sup> provides the editor plug-in for Eclipse. But the point is that these supports are restricted to code highlight, simple error checks and outline view for writing the templates and as a template engine, FreeMarker is not responsible for complex tasks like providing modeling or meta-modeling tools. The critical development parts are related to input model manipulations and FreeMarker Objects preparations, which are not really related to FreeMarker.

### **Maintainability and extendibility of the code generator**

No traceability features are provided by FreeMarker and if needed, it should be added manually to the input Objects as properties, so they can be later reflected in the template. For debugging purposes, transformation logs provide information like line number of the errors, and there are some third-party editors which provide running transformation in debug mode.

### **Improvability and connectability of the of the code generator**

The code generator can always be improved in any desired way, as long as the objects fed to FreeMarker are in correct format. Since FreeMarker is designed for Model-View-Controller (MVC) pattern, it does not involve changes in the code generator's code and the logic of the code generator is kept separated in the Java code.

FreeMarker also allows working directly with Python objects and using JSP custom tags.

### **Portability of the code generator's development environment**

FreeMarker is available as a JAR file and is exposed as a Java API.

---

<sup>46</sup> <http://www.jboss.org/>

## Usability of the code generator's development environment

There are a couple of editors for different IDEs like Eclipse, NeatBeans and IntelliJ IDEA which provide the developer with code highlight, template outline, syntax error and code completion based on macro names and Java bean property names.

There are a couple of useful features in FTL which makes it an easy to use template language, some of which are stated below:

- Direct access to Java objects and methods
- loop handling like access to control variables of inner and outer loops, breaking out of the loop
- compare and formatting for date/time
- array handling like accessing elements by index and query length of array
- Macros with support for passing them as parameters, definition of local variable and also recursive call
- Namespace support for variables
- whitespace handling by providing trimming directives and also automatically removing the non-outputting white spaces in the templates
- built in operations for manipulation of Java-independent string, list and map
- Migration tools for converting from Velocity<sup>47</sup> solutions e.g. converting Velocity templates to FreeMarker templates.

Because of its simple and easy to use syntax, it is easy to learn and adapt FreeMarker for template based code generation. Also there are good documentation, manuals and tutorials available over the web.

## 4.2.8. Actifsource

### 4.2.8.1. *Concept*

Actifsource<sup>48</sup> is a commercial model driven code generator. Meta-model definition, modeling and code generation development can all be done using Actifsource plug-in for Eclipse. It is

---

<sup>47</sup> <http://velocity.apache.org/>

possible to generate any kind of textual output by developing the required template in its template editor, which is claimed by Actifsource to be a whole new approach for creating templates, without a need to learn any template language.

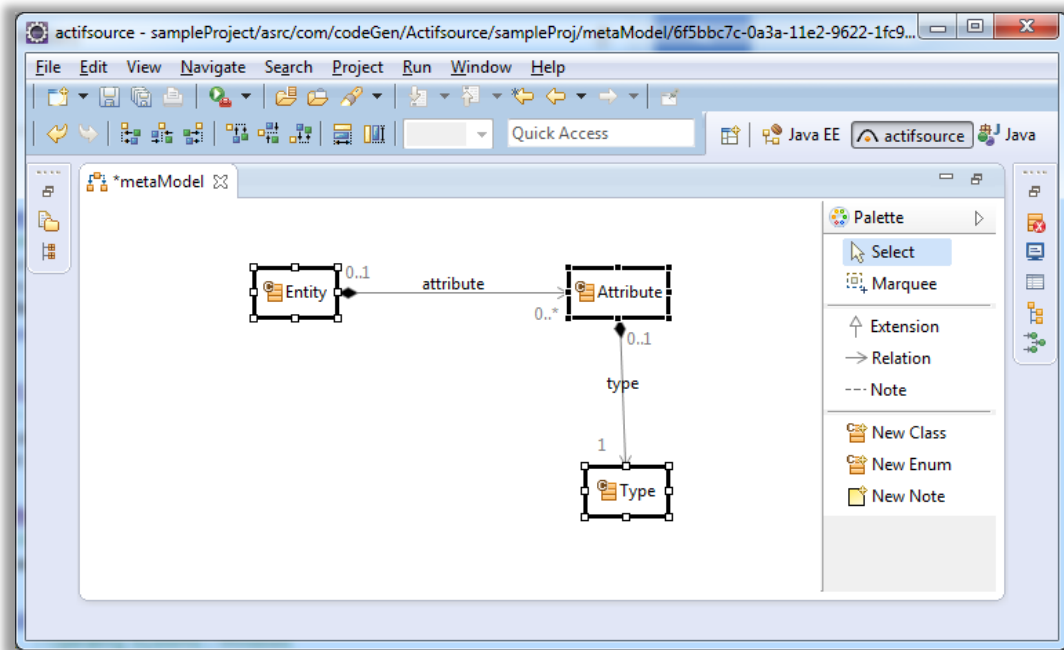


Figure 4-9 meta-model created in graphical designer

Meta-models are created using the graphical designer. A type based approach is used for creating the entities and relationships between them, resulting in strongly typed meta-models. Type safety makes the development of models and templates easier by providing features like content-assistance and error detection. Actifsource makes use of Eclipse builder infrastructure, so changes in model and template are detected and generated code is updated automatically.

<sup>48</sup> <http://www.actifsource.com/>

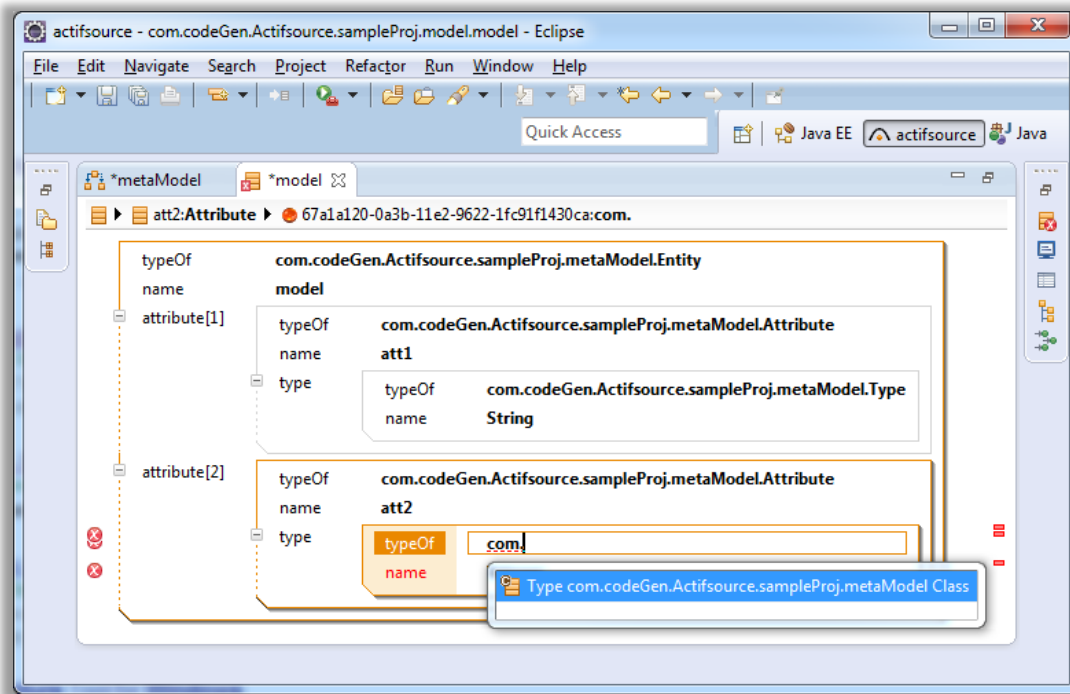


Figure 4-10 model created by help of content assist. Errors are detected based on meta-model

Templates are written in a declarative manner and it is possible to generate each part of the template for all occurrences of a part of meta-model, specified in the GUI as “selector”. Selector is specified by marking a specific part of the template and also specifying its related meta-model element in “selector” field, instead of doing it in template code. So lines of the template can be marked and assigned to a context, meaning this part of the template should be repeated for all the occurrences of the selected meta-model element. Inside each assigned part of template code, it is again possible to assign the smaller parts of the template to another context, probably included in the current selected meta-model part. For instance one part of the template is assigned to *Entity* class in the meta-model to generate a class per each entity and inside it another part is assigned to *Attributes*, which are owned by *Entity*, to generate the code for local variable declaration.

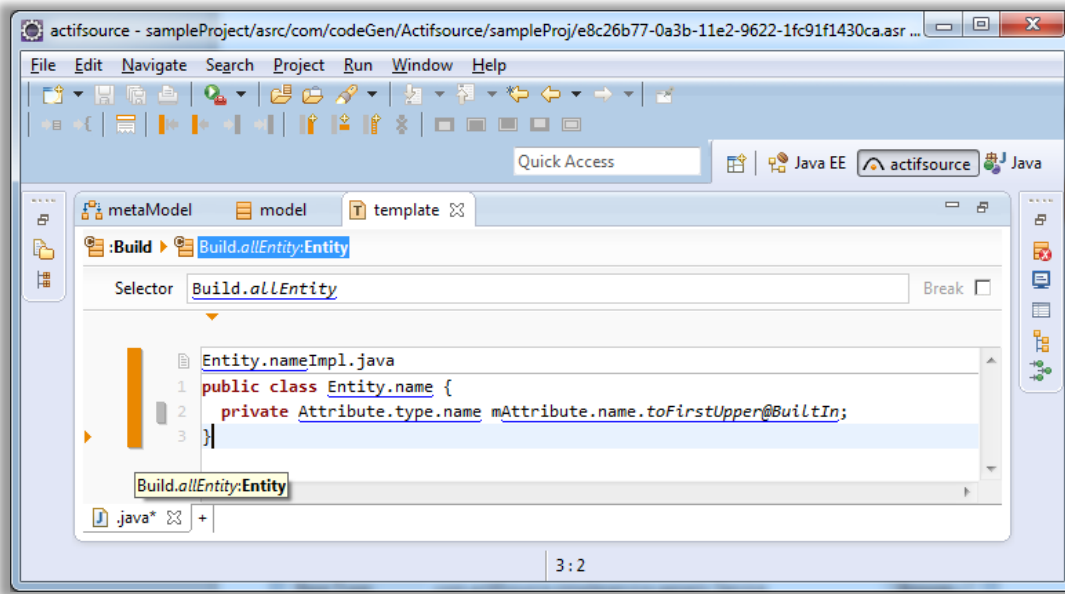


Figure 4-11 The template body assigned to “Entity”, specified in “Selector” field and marked by vertical orange bar on the left

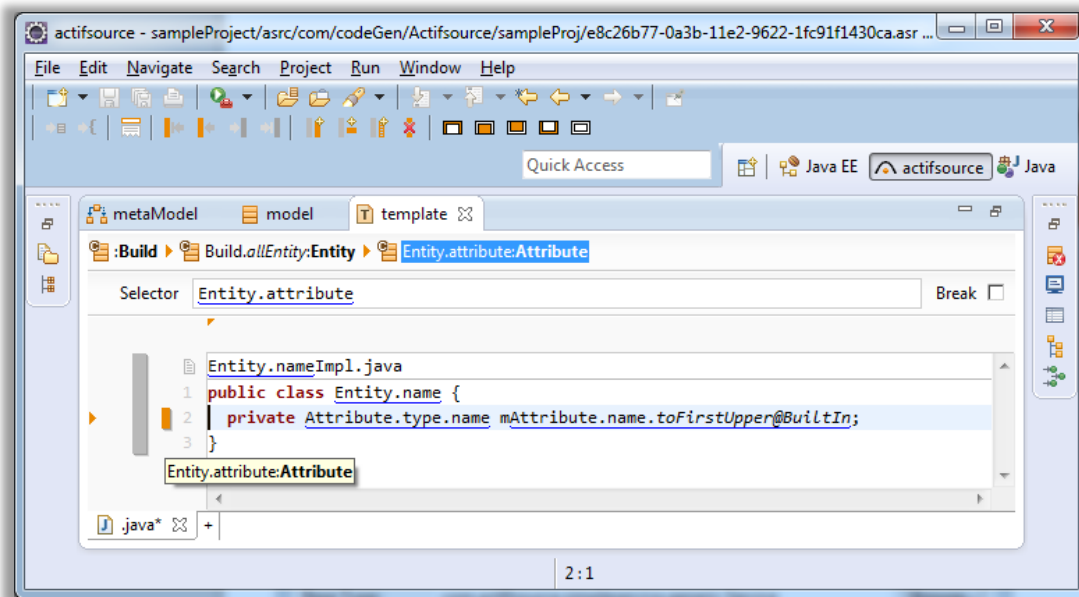


Figure 4-12 Inside the template partially assigned to “Attribute”, specified in “Selector” field and marked by vertical orange bar on the left

As explained, templates are written declaratively, by marking the parts of the code in the GUI and assigning them to meta-model, based on their type. Then the GUI marks different parts of the template code, based on the scope of the meta-model to which they are assigned. This assignment is not only for lines, but also for the columns inside each line it is possible to repeat a

pattern for all the occurrences of a type. A good example is the generation of a method signature, which needs to include all the input parameters of a method. So Lines are assigned to type “Method” and the part of the code, which is inside the parentheses in the first line of the method, is assigned to “Parameter” type.

Java methods can be written for each type in meta-model, in order to handle complex tasks or to make the code more readable, e.g. defining a simple method *getType* which is used for finding the type of a *Parameter*, instead of writing “Parameter.type.name” all over the template. The IDE helps the developer to write these java methods and keep them as resources anywhere needed in the packages, e.g. together with the types for which they are written or in a separate general purpose library. This helps to have a better organization for the meta-models and their related templates and methods, and makes the meta-models easily reusable. These methods are then added to code completion menu based on the package they are put in.

Implementation of aspects is also available in the modeling environment, which can be used to add features for using in different types and parts of the meta-model. Examples of aspects that a user can implement are resource validator, range validator, refactoring aspects and literal aspects. For instance it is possible to implement a refactoring aspect, according to a change in the meta-model, to apply it for updating the models.

Actifsource is available in both free and commercial versions. The free version or Community Edition runs on an open source stack and is supported by the Actifsource Community via the developer forum, provides the code generator, meta-model editor, template editor for any program language and UML Meta model Editor. The commercial version or Enterprise edition provides additional features like Ecore import/export, Ant support for headless code generator, team support which is diff/merge for models.

#### **4.2.8.2. *Code generator usage features***

##### **Function requirements**

*Output:* Any text output can be generated using Actifsource. Also it features flexible and high quality documentation output. One possible output is generation of a new Eclipse project,

together with its settings and required files, about which there is a tutorial on the Actifsource website.

*Input:* Input models should either be developed in the graphical designer or they can be imported as UML diagrams, created in any UML tool outside Actifsource.

### **Reliability of the code generator**

Because of strongly typed meta-models, the models and templates are less error prone. They are checked against the meta-model on the fly during the development. In addition it is possible to organize the meta-models and code generators in packages, just as any Eclipse project, which makes them easier to reuse. Reusing is an important factor to have high quality software.

### **Extendibility of the product software, developed by using the tool**

Changing the input model in the model editor is instantly reflected to the generated code. Actifsource helps changing the input model by features like content-assist, and by saving the changes, the output code will be generated automatically.

Change is not only supported for the input models, but also changing the software architecture is supported by Actifsource. Since the design of the software is done inside Actifsource editors and models for software components are integrated with each other in the environment, changing the structure of the system is under Actifsource control and it will refactor automatically as much as possible and can also help to find the produced errors during this process. As the result of these changes, the output code is updated accordingly as well.

It is also possible to put protected context in the middle of the generated code, which will not be overridden by the future code generations. So the end user can change those parts of the output code manually.

### **Maintainability of the product software, developed by using the tool**

Traceability is available from models and templates to the meta-models. Direct traceability is not available from generated code to models or templates out-of-the-box. Since models are presented graphically, they have enough readability for a user to easily understand them.

When the source of an error is found, the good usability features of Actifsource, which are explained below, will help the developer to fix the problem easily and see the output.

### **Portability of the code generator**

The free edition only provides running the code generator inside the IDE. This is reasonable since the input models are also only editable and created in this environment anyway. But in Enterprise edition, there is an ANT task which makes it possible to generate code without any dependency to Eclipse.

### **Usability of the code generator**

The graphical model editor is available for editing and creating the models. It has features like content-assist and error detection. In the enterprise edition, teamwork and diff/merge tool for models are also available. Also the code generation runs manually by changing and saving the input models, so the output is updated automatically.

Since the code generation project is an Eclipse project, it is easy for developers to start working in it, especially for the ones already familiar with Eclipse. Good tutorials can also be found on Actifsource website, which helps the users with using the model editor.

### **Resource saving**

Code generation in Actifsource is automatically and the generated code is always updated by saving the changes in the input model. So basically in the case of code generation inside Actifsource Eclipse plug-in, since the code is generated incrementally, the response time is not high due to the small amount of work that the code generator should perform each time.

#### ***4.2.8.3. Code generator development features***

##### **Function requirements**

Ecore, as a popular standard for modeling is supported by Actifsource. It is possible to import and export Ecore meta-models to development environment.



### **Reliability of the code generator's development environment**

As a commercial tool, it is expected to be a reliable tool. Of course it depends on how successful the tool is and how many users it has. There are exceptions happening in the free edition environment once a while, which causes the IDE to request the user to report them.

### **Maintainability and extendibility of the code generator**

Models and meta-models are graphically represented, making them easy to understand. Templates are very readable because of the new form of developing them, which is a combination of template code and graphical GUI features. Running the generator in debug mode is not available.

Traceability between models and Templates, back to meta-model and vice versa is available, although there is no out-of-the-box solution for traceability for the generated code.

There are a couple of features affecting the ease of changing the code generator. Extending and changing the models is easy because of graphical editor, content assist and error listing. Changing the model causes automatic code generation. Simple changes in meta-models like renaming are updated automatically in the models. For the complex refactoring like changing a class's structure, it is possible to write refactoring by using the refactoring aspect.

### **Improvability and connectability of the of the code generator**

By importing Ecore meta-models, it is possible to integrate the code generator with other systems. Since different meta-models can be included in the same project, it is possible to extend the generator step-by-step, using refactoring if needed. The ability to include different meta-models in different projects encourages reusing the existing meta-models as much as possible. The same holds for the templates and the java methods, used in the templates. Aspects are also good ways to extend the features in the environment.

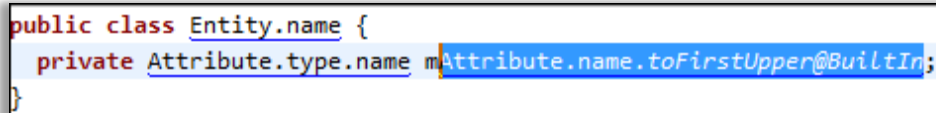
### **Portability of the code generator's development environment**

Actifsource is integrated to Eclipse as a plug-in.

## Usability of the code generator's development environment

The development environment is an Eclipse plug-in which is a familiar environment for programmers. Graphical editors with content assist are used to create models and meta-models. Since the meta-models are strongly typed, content assist in development of models and templates is based on meta-model definition and furthermore, errors are detected and highlighted. Navigation from anywhere to meta-model elements is possible by pressing ctrl + click on their name. Also navigating back and forth to previous places in the IDE is available. The result of the generation is always available instantly, after saving the changes.

Based on the syntax of the template's output language, template code is highlighted. it is also possible to introduce a customized highlighting for any desired target language. Also the indentation of the templates are similar to output, since the additional context data like the related meta-model elements for each template part is presented graphically and there is no additional scripting expressions like loops and "IF" statements inside the templates. These features, in addition to java method calls on meta-model elements, improve templates' readability. In addition there are a couple of built-in Java functions which can be called on model elements like *toUpper*, *toFirstLower* and *length* for strings.

A screenshot of a code editor window showing a Java class definition. The code is: 

```
public class Entity.name {  
    private Attribute.type.name mAttribute.name.toFirstUpper@BuiltIn;  
}
```

 The text `mAttribute.name.toFirstUpper@BuiltIn` is highlighted in blue, indicating a built-in method call.

Figure 4-13 built-in method "ToUppderFirst" used in template for attribute name

Simple refactorings like changing names in meta-model are automatically propagated to models and templates. For more complex refactorings, it is possible to use aspects to implement the refactorings and apply them to the models for changing accordingly. Also packaging changes are done easily by copy/paste or drag and drop.

Actifsource supports Eclipse team providers and also diff/merge facilities for both models and templates, which makes it possible for a team to work on the same code generator. This feature is not available in the free version.

Many useful tutorials can be found in Actifsource website, covering all the important concepts, which are available from basic to advanced topics. Looking at the Actifsource website, it seems that it lacks a complete documentation or reference for developers or at least for developers of the free version. Also looking at the development forum, it seems that it does not have a big user community.

Furthermore Actifsource team offers consulting and customized training programs in topics like Domain specific languages, model driven architecture, meta model design, template maintenance and code generation technology and existing frameworks.

## **4.2.9. XSLT**

### **4.2.9.1. *Concept***<sup>49</sup>

XSLT is a language to describe transformations which is usually from one or more source XML files to any text outputs like other XML files, HTML documents and plain text output like Java source code. An XSLT transformation, which is called a style sheet, comprises of a set of patterns to match different elements of the input tree and which are associated with a set of templates to describe the format and content of the output. A style sheet is itself a well-formed XML file.

XSLT is a declarative language. So instead of describing the sequence of required tasks for transformation, XSLT transformations define the set of rules that hold between the input and output. The rules are composed by defining a set of templates that have two fundamental parts. The first part is the matching pattern which assigns the template to the elements of the input XML. XPath language is used to access and navigate the input tree, which also provides a set of operations and functions like arithmetic and comparison operators and string and numeric manipulation functions. Then the body of template defines the output structure and content based on the matching element, which can itself call other templates or use XPath to navigate the input tree in order to access its needed data.

---

<sup>49</sup> <http://www.w3.org/TR/xslt#section-Introduction>

The XSLT processor gets the input XMLs and the XSLT transformation and generate the output based on them. The processor first creates a source tree based on the input and the processing starts from the root node of the tree. The processing is steered by the templates and continues by creating nodes in the result tree, based from which the output is derived, or processing other nodes in the tree. In the first version of XSLT, there was only one output as the result, while in the latest version, which is XSLT 2.0, it is possible to generate multiple outputs. The input and output files are separate files.

XSLT 1.0 [22] was published by World Wide Web Consortium (W3C)<sup>50</sup> in 1999, which is a very powerful and widely used language. The browsers and other environment need to use an implementation of XSLT processor in order to support it. There are different free and commercial implementations of XSLT by different vendors like AltovaXML<sup>51</sup>, Saxon<sup>52</sup>, libxslt<sup>53</sup> and Xalan<sup>54</sup>. XSLT 2.0 [23] was published in 2007 by W3C which has some useful features discussed later, but is not still used as widely as XSLT 1.0, since it is not still supported out-of-the-box by many environments like web browsers.

XSLT files can be developed in text editors and XML editors. An advanced XML editors like Altova XMLSpy can provide lots of useful features like debugging and profiling, as well as editing features based on the XML schema like auto-completion. It also provides great visual tools for viewing and editing the XML schema.

### **XSLT 1.0 vs. XSLT 2.0 [26]**

The main improvements of XSLT's new version are:

- Possibility to output more than one file from the same template
- Variable to keep the result of a transformation, so the intermediate models can be created and kept in the memory to work on. So it is possible to work on more than one tree at a time

---

<sup>50</sup> <http://www.w3.org/>

<sup>51</sup> <http://www.altova.com/XMLSpy.html>

<sup>52</sup> <http://saxon.sourceforge.net/>

<sup>53</sup> <http://xmlsoft.org/XSLT/intro.html>

<sup>54</sup> <http://xalan.apache.org/>

- Custom functions, which can be called very easily, like expressions instead of creating custom tags, that make the templates easier to develop and more readable.
- Schema-aware processing which enables the transformation to validate the inputs and outputs, based on their specific xml-schemas, in order to make sure the transformation does not receive a wrong input or generate a wrong output. It also makes it possible to import elements, attributes and type information from a schema into the style sheet to enable type checking in order to reduce the faults during development.

#### **4.2.9.2. Code generator usage features**

##### **Function requirements**

*Input:* XML files are the typical input, although there can be others. Since XSLT and XPath use an internal data type called XDM [24] which can be accessed programmatically. So any type that can be converted to this format by the processor can be used as an input format. Relational database tables are the examples of non-xml sources that can be used as input.

*Output:* Any kind of output text can be generated by XSLT. Also it is possible to use XSL Formatting Objects (XSL-FO) along with XSLT, in order to create outputs like PDF.

##### **Reliability of the code generator**

XSLT is being used now for many years and the available processors for it are from reliable and well known vendors and they are tested for years. Also in order to develop a reliable code generator, the developer can use advanced XSL editors like XMLSpy to use features like debugging and code completion based on the schema. Using XSLT 2.0, it is possible to add checks for the input model and output text, which is done against their schema. Also it is possible to add type checking to the transformation.

##### **Extendibility of the product software, developed by using the tool**

There is no built-in mechanism like protected tag or block in XSLT to provide incremental generation and enable the developer to edit the parts of code manually. It should be implemented manually e.g. by some other code that calls the generator.

### **Maintainability of the product software, developed by using the tool**

It is up to the developers to provide traceability from the generated code to their input models. Also there is no built in mechanism provided by XSLT processors to generate the output partially, like only the parts of model that are changed. So as the input model gets larger, the generation time may also increase and that can make the developers' life harder, as they have to wait a long time to see the result of adding a single attribute or changing the type of a parameter.

### **Portability of the code generator**

XSLT processors are available for all the operating systems. Also Ant and Maven have tasks or plug-ins for running XSLT transformations.

### **Usability of the code generator**

As the input model is XML, the features of the XML editing tool are what determine how easy and user friendly the process of model manipulation is.

### **Resource saving**

XMLSpy as an advanced XSLT editor provides profiling, which can be used to optimize the performance of transformation process.

XSLT processors does not keep track of the input model changes, in order to generate partially based on the changes. So each time the XSLT code generator runs, all the transformation steps are repeated, even if the change only affects one line in a file, out of hundreds of generated files.

### ***4.2.9.3. Code generator development features***

#### **Function requirements**

The input model should be an XML or anything that can be converted to XDM data type by XSLT processor. There is no restriction on the meta-model of the input XML, since it is not necessary to determine a meta-model for the XML file. But if needed, XSLT 2.0 accepts XML schema to provide a schema-aware transformation.

### **Reliability of the code generator's development environment**

It depends on the chosen editor.

### **Maintainability and extendibility of the code generator**

One major problem with the transformations written with XSLT is its declarative nature. Usually it's hard to read XSLT code and follow what does, especially when the input meta-model is complicated and large.

Since XML files can import or include each other, it can be hard to see the real input tree which is processed by XML transformation, as It is not visible and is created in the memory when processing happens. One useful practice can be writing the input tree to an output XML and using XML tools to view or navigate it in an understandable way.

XSLT 2.0 lets the transformation to create an intermediate model, say from different input trees and then it is possible to print it out as an XML as well. This makes the code generator more understandable, since the actual model on which the transformation is done is visible to the developer, instead of just kept in memory. Also having such an intermediate representation, it is easy to add new output formats and languages to the generator, which is very similar to using intermediate representations in compilers.

No built-in traceability features are provided, but with help of editor it is possible to use features like debugging and code completion based on the schema, during maintenance or improvement.

### **Improvability and connectability of the of the code generator**

As it only supports XML schema as input meta-model, integrating code generator with systems with other meta meta-models needs more effort, compared to those which are also based on XML schema. But as XML can be used to represent nearly any kind of model and also XSLT is very powerful as a transformation language, there are no boundaries on what one can do with it, although it is not an easy language to work with.

**Portability of the code generator's development environment**

Depends on the chosen editor, but sufficient tools for all the environments are available since XSLT is a very common and widely used language.

**Usability of the code generator's development environment**

As mentioned before, the XSLT files are not usually easy to read and understand, especially for complex transformations. Its declarative characteristics make doing some of the complex tasks easy, while the reader of code may not easily understand how and where in the code they are done.

Using the intermediate models and dumping them as XML file, which is provided in XSLT 2.0 is a good way to make the generator easier to understand.

The editing and development features depend on the chosen editor.



## 5. Code generation tools comparison

After looking at the features of different code generation solutions, it's easier to find the best choices among them according to the company's requirements. In this chapter, first the requirements of the company will be stated and explained. These requirements are originated either from the existing problems or the possible improvements for the company's existing code generation solutions.

In next step, code generator features will be prioritized based on the importance level of company's different requirements. There will be three priority levels. High priority features are the mandatory ones. The second group or medium priority features are the ones which are good to have e.g. for improvement of the solution. Finally the low priority group is the set of features that are not important according to the company's requirements.

Having such a prioritized list of features, it is easier to find the most suitable tools. The low priority features can be removed from the list of features. Then the tools that do not fulfill the high priority features may be eliminated from the list. Finally, priority 2 features will help to compare between the remaining tools.

### ***5.1. Current code generation solutions in the company***

The company has a main product which is customized for different applications. Customization is done through different XML files, which are used to describe the components, services, data types and messages of the system. These XML files are the inputs for the code generation process, which result in generation of:

- 1 Java interfaces for services
- 2 Java objects for data types and messages
- 3 Stubs and proxies to connect the clients to servers by converting Objects to XML messages and vice versa.
- 4 Hibernate XML files for database definition
- 5 HTML documentation

This code generation is implemented using XSLT. Because of size and complexity of the product, the definitions are broken into different XML files. So the input is comprised of multiple XML files. An intermediate model tree is first created based on all the inputs and then this intermediate model is used as the main input for the code generation process.

Another major code generation solution is used to make it possible for the system to communicate with outside world, using Financial Information eXchange protocol (FIX) [25]. The input for this transformation is FIX protocol definition or FIXML, which is an XML file defining the format and structure of FIX messages. As output, the solution generates Java code for converting FIX messages to company's internal protocol and vice versa. Documentation is generated in the solution as well.

FIX code generator is written in Java code and it uses FreeMarker as template engine. Some initial modifications are done on the input XML, in the Java code, in order to prepare the correct input to FreeMarker template engine. Then FreeMarker engine is given the Java object models and transformation is carried out based on FreeMarker templates.

Another type of code generation which is needed in the company is generating Java code based on Java interfaces. The current solution is purely in Java and uses reflection for reading the input files, which are Java interfaces. The output format is defined in Java code as string constants and variables, as opposed to most code generation solutions which use templates to keep the output format separated from code.

## ***5.2. Problems and Improvements***

### **5.2.1. Code generator's usage complexity**

The complexity of the input model is one source of problems when using a code generator. When the input models are complicated, the new developers usually have hard times to find the correct place to create or modify the input model for generating their desired piece of code. The code generators which provide traceability from the generated code to the source model can help in such cases. Documentation for models can also help the user finding the structure of the input model and the meaning of its components. So the easier the generation of modeling

documentation, the easier it gets for the developers to work with the code generator. Also readability of the input model, like a using graphical view or outline can help the developer with easier usage of the generator.

### **5.2.2. Code generator's performance**

Another problem in the current solutions is performance. Currently the code generation time does not depend on the changes in the model. No matter how big is the change, all the input models are read again and all the output code is regenerated again. No matter if a new data class or message is added, or only an attribute is created or modified, the generation takes the same time. This slows down the process of development and maintenance. Some of the existing solutions support code generation based on the changes in the input model, which for instance can reduce the total code generation time for adding a new attributes from minutes to milliseconds. Moreover version controlling of the generated code is easier in such generators. Since the code generation is faster, there is no need for version controlling of the generated code in many cases, so it's enough to do it only for the input models. Even if there is a need to do it, since the same output will not be generated again, version controlling will be easier and cleaner.

Another useful feature regarding performance is profiling the process of code generation in time of its development. This helps the developer of the code generator to find the performance bottlenecks in code generation solution. This is a feature that should be supported by the modeling framework used with the code generator.

### **5.2.3. Complexity of the code generator's implementation**

Complexity of the code generator's code is one of the main concerns about the existing code generation solutions. Complexity is mentioned as the reason why it's hard to maintain the code generator. It also makes it hard for a new person to understand how it works, so the solution is highly dependent on its original developers.

Input model can be a source of complexity in the solution, but it is not necessarily originated only from the complicated domain. So even if the domain is complicated, still the tools for modeling and viewing the models can help the developer a lot for better understanding them. But

the current XSLT solution in the company is an example of a solution with a very complex and hard to read set of input models. XSL files import the XML files that are the sources of input data, but these XML files themselves may import some more XMLs which themselves may import others. So the solution uses the power of XSL to finally build up a big input model in the memory that is a tree of XML files importing each other. This tree is used as the input for the XSL transformation.

The problem is that this intermediate model is created only when code generation is running. It's not possible to see this model because it is temporarily kept in the memory. But even if it was written to the output as a big XML file, it wouldn't be an easy to read model because of its complexity and size. Using a modeling tool for viewing and editing the input model, with features like graphical model view and code completion based on the meta-model, can help with handling the models more easily. One should consider supported model formats when choosing such a modeling framework or tool.

But the complexity is not only about the input model. The code generator logic may also be complex and hard to understand. The XSLT solution has an inherent complexity, caused by its declarative nature. Generally it is hard for many developers to understand the logic of declarative languages, especially when the data model is complex and large. XSL is a very powerful language which is mainly used to transform an XML tree to another one and is not specific for code generation. But there are many template languages, which are easier to read and understand since they are created specifically for this purpose.

Still the simplicity of the language is not enough on its own. The other existing solution which is based on FreeMarker template engine uses Java code to modify and merge different XML input models, in order to prepare input XML file which is input for code generation process. This means a part of the code generation logic, which is model transformation and preparation, is kept in Java code. The Java code for transformation of textual data is hard to read and maintain since Java is not a language specialized for text processing. For the same reason that templates are used for code generation, which is a model to text transformation, model to model transformation rules are better to be defined in its own language and format to be more readable and

understandable. Using a language like Java may be a fast solution to implement in the beginning, but it shows its drawbacks by when the solution gets more complicated and needs maintenance.

#### **5.2.4. Need for a user friendly development environment**

An easy to use and learn tool not only helps with easier development and maintenance, but also can motivate the developers to use code generation in their daily software development more often. Creating system test cases is a good example of a task that can take advantage of code generation. Especially in the projects that have changing or vague requirements, definition of the test cases in a separate file and generating all the needed test cases can help with fast development and easy maintenance. More usage of code generation results in products of higher quality.

A code generation development environment which is easy to learn e.g. is not very complicated and has good learning materials, can help a lot with spreading the use of code generation in the company.

#### **5.2.5. Fulfilling the current features**

In addition to solving the existing problems, the new solutions should not lack any features compared to the current solutions. Some examples of current features are integration with Ant and Maven, formatting or beautifying the output source code, using manual code between the generated code and functional requirements like generating code and documentation.

#### **5.2.6. Possible further improvements**

Another reason which makes the company interested in reviewing code generation tools is the possible improvements to the solution, provided by new tools and frameworks. There are different features that may not be necessary to have in the solution, still having them is useful for the system. Writing constraints for the acceptable values of input model elements is an example of such a feature, which provides a better usability for the end user by generating warnings and errors for wrong input values. Development of a DSL for system modeling is also another example of such improvements.

### 5.3. Prioritizing the features

Now it's time to describe the code generation solution needed by company, based on the restrictions and requirements. The same categorization of code generation requirements, as described in Chapter 3 will be used to describe the solution's specifications. The function requirements will be listed, which should be later fulfilled by the selected solution. Each quality measure will be given a priority, which later will make it possible to choose the best match between the possible solutions.

#### 5.3.1. Features table for code generator usage

Category	Feature / Priority	Description
Functional Requirements (Code generator)	<ul style="list-style-type: none"><li>▪ Output: Source code (Java, C++, ...) / HIGH</li><li>▪ Output: documentation / HIGH</li><li>▪ Input: XML models / HIGH</li><li>▪ Input: Java Interface / MEDIUM</li></ul>	
Reliability (Generated code)	<ul style="list-style-type: none"><li>▪ Wrong output: reliable engine / HIGH</li><li>▪ Wrong output: usable development environment / HIGH</li><li>▪ Wrong Input model: modeling support based on meta-model/ MEDIUM</li><li>▪ Wrong Input model: constraint definition on input models / MEDIUM</li></ul>	<ul style="list-style-type: none"><li>▪ Related to “usability” of the Code generator usage and development</li></ul>
Extendibility (Generated code)	<ul style="list-style-type: none"><li>▪ Manual code within the generated code / HIGH</li><li>▪ Usable modeling support / Medium</li></ul>	<ul style="list-style-type: none"><li>▪ Related to “usability” of the Code generator usage</li></ul>
Maintainability (Generated code)	<ul style="list-style-type: none"><li>▪ Traceability from code to input model / HIGH</li><li>▪ Code generation speed/ MEDIUM</li></ul>	<ul style="list-style-type: none"><li>▪ The users are software developers, so the models should not be necessarily graphical to be readable for them</li></ul>

	<ul style="list-style-type: none"> <li>▪ Readable input model / HIGH</li> <li>▪ Usable development environment / HIGH</li> </ul>	<ul style="list-style-type: none"> <li>▪ Related to “usability” of the Code generator usage and development</li> <li>▪ Related to “response time” of the code generator</li> </ul>
Portability (Code generator)	<ul style="list-style-type: none"> <li>▪ Operating system compatibility / HIGH</li> <li>▪ Ant/Maven support / HIGH</li> </ul>	
Usability (Code generator)	<ul style="list-style-type: none"> <li>▪ Auto generating documentation for input model/ MEDIUM</li> <li>▪ Modeling support: readable model view/ HIGH</li> <li>▪ Modeling support: easy text based editing e.g. DSL support / HIGH</li> <li>▪ Modeling support: graphical editing/ MEDIUM</li> <li>▪ Modeling support: based on meta-model/ MEDIUM</li> <li>▪ IDE integration for running the code generator / MEDIUM</li> </ul>	
Resource saving (Code generator)	<ul style="list-style-type: none"> <li>▪ Speed: partial code generation based on model changes / MEDIUM</li> <li>▪ Speed: partial code generation by user selection / MEDIUM</li> <li>▪ Speed: optimization facilities in development time / MEDIUM</li> <li>▪ Caching of the queries to model by the code generator / MEDIUM</li> </ul>	

Table 1 - Prioritizing features for code generator usage

### 5.3.2. Table for code generator development

Category	Feature / Priority	Description
Functional Requirements (development environment)	<ul style="list-style-type: none"> <li>Input: XML schema and Java interface / HIGH</li> </ul>	<ul style="list-style-type: none"> <li>The environment should support the current models at first place. changing meta-models may happen later, but supporting the current models help with step by step migration of the solution</li> </ul>
Reliability (development environment)	<ul style="list-style-type: none"> <li>e.g. bug in the modeling tool / MEDIUM</li> </ul>	<ul style="list-style-type: none"> <li>The errors happen in the code generator and not directly in the product software</li> </ul>
Maintainability and Extendibility (of the code generator)	<ul style="list-style-type: none"> <li>Out-of-the-box traceability between meta model and templates / HIGH</li> <li>Out-of-the-box traceability between generated code and model, meta model and templates / MEDIUM</li> <li>Development supports in IDE e.g. Debugging, Error highlight, code completion / HIGH</li> <li>Refactoring support / HIGH</li> <li>Readability of the templates / HIGH</li> <li>Readability of the input model / MEDIUM</li> <li>Organizing and re-using meta-models / HIGH</li> <li>Aspect oriented programming for templates / MEDIUM</li> </ul>	<ul style="list-style-type: none"> <li>Related to “usability” of the Code generator development</li> </ul>
Improvability and Connectability (of the code generator)	<ul style="list-style-type: none"> <li>Supporting different meta-models out of the box / MEDIUM</li> <li>Support for using DSL / MEDIUM</li> <li>Refactoring support / HIGH</li> </ul>	



Portability (development environment)	<ul style="list-style-type: none"> <li>Operating system compatibility / LOW</li> <li>IDE dependant / LOW</li> </ul>	
Usability (development environment)	<ul style="list-style-type: none"> <li>Documentation and training material / HIGH</li> <li>Development IDE support, Editor supports, based on language: code highlight, outline, Errors and warnings / HIGH</li> <li>Development IDE support, based on meta-model: Errors and warnings, Traceability, code completion / HIGH</li> <li>Development IDE support: Debugging / MEDIUM</li> <li>Graphical meta-modeling environment / MEDIUM</li> </ul>	
Financial saving (development environment)	<ul style="list-style-type: none"> <li>Commercial / free</li> </ul>	

Table 2 - Prioritizing features for code generator development

## 5.4. Tool selection and comparison

In this section, the introduced code generation solutions will be filtered and discussed based on the company's priorities. First some of the tools will be filtered out since they do not fulfill one or more high priority requirements. Then the features of the remaining tools will be compared side by side per each requirement category. The comparison is based on implementation of one of the main company's code generation scenarios.

## 5.4.1. Filtering out some of the tools

### 5.4.1.1. *Atom Weaver*

The first tool to be eliminated is Atom Weaver. The most important reasons for not choosing it are:

- The existing input models cannot be imported to Atom Weaver since it does not support their meta-models. Atom Weaver has its own format for defining the meta-models, so if it is chosen as the solutions, all the meta-models and models should be redefined from scratch. But even if there are no problems with spending time and effort for such a migration, it seems risky to convert all the system models to a format that is not common and widely used.
- It is not possible to run the code generator in standalone mode or call it from ant tasks for automation

### 5.4.1.2. *JET*

JET is the next tool to remove from the list. Although it has good meta-modeling and improvement supports, but its usability features are not fulfilling the needs of the company for a code generation solution. Even with a powerful tool like Rational Software Architect, there are some main features that are missing. Since this tool is basically created for implementation of known patterns, it is more useful for creating an M2T project based on an input schema. So it is a useful tool for initiating a code generation project, but since its focus is on software patterns, it provides facilities to form the structure of the output solution and creating and binding the templates to input meta-model. But it doesn't provide any features for maintenance and improvement of the solution.

Maintenance of the code generation solution is one of the main priorities of the company which does not have powerful support in the JET development tools. Traceability is not supported out-of-the-box and it should be implemented by the user. Refactoring is another important feature which is not supported in these tools. For instance renaming the input meta-models will not update the templates in Rational Software Architect.

#### **5.4.1.3. *AndroMDA***

The main disadvantage of AndroMDA is that it doesn't provide a development environment for its code generators. There is no editing support for development of custom cartridges and all the files should be developed as text files. If a code generation solution can be developed using the out-of-the-box cartridges, then it is useful to use AndroMDA as the code generation solution. But when there is a need to develop a new cartridge or edit an existing cartridge dramatically, the developer has to take care of creating all the needed files in the cartridge without any tool support or assistance. This will cause the maintenance and development process to be slow and bug prone.

#### **5.4.1.4. *MetaEdit+***

Although it is the one of the leading model-driven solutions, there is one its characteristic that does not match the company's requirements. In order to introduce the meta-model and input model, one should either use the graphical tool to define it or import it as a GOPPRR compliant model. Graphical interface is not always the fastest and easiest way of defining the models. It's always good to have such a graphical tool e.g. for viewing the models, but a text based approach is always needed when the user of the tool is a software developer and not a business expert.

Sometimes the input model is in hand, like a protocol definition, and a code generator should be developed for it. First step should be introducing the meta-model to the development environment. But defining such a meta-model can be tiresome, complex and error prone when using a graphical tool. Furthermore such protocols usually have a meta-model e.g. XML schema. While many tools allow importing meta-models to the environment, specially the popular formats like XML schema, MetaEdit+ only accepts them as GOPPRR meta-models. GOPPRR is a standard only used by MetaEdit+ and not a usual format used by many.

This means migration of the current solution to a new solution based on MetaEdit+ needs a lot of effort and its maintenance and improvement may need the same effort later.

#### **5.4.1.5. *FreeMarker***

FreeMarker is a free-format template engine with some restricted capabilities. The main drawback of FreeMarker is that it is not possible to introduce a meta-model of the input model to it. This means the logic related to input model parsing is kept in the code, instead of a high level definition or meta-model. So basically the code generator is not developed based on types and meta-level information. Java objects are the links between the input models and templates, which means the editors cannot support meta-model sensitive supports like error detection, code completion and traceability for the developer of the templates. This not only makes it hard to develop a complex code generator, but also makes it hard to maintain or refactor.

#### **5.4.1.6. *XSLT***

The main problem with XSLT is its complexity. The code written in this language can get very complex and hard to read. Maintenance and also understandability of the generator code for a new developer can be very cumbersome.

Furthermore there is no out-of-the-box solution for partially generating the code and this can result in waste of a lot of time during development, when the input model is very large.

### **5.4.2. Side by side comparison**

The remaining tools are compared side by side in this section based on the sample task implemented by each of them. For each category of features, one table is provided and again each feature is mentioned for both code generation usage and code generation development environment.

Some of the aspects like code generation speed are eliminated from the tables, since such features need measurement and benchmarking which is out of the scope of this work. Instead for such aspects, the related features which have effect on them are stated.

XSLT and FreeMarker are included in the comparison table only for comparing them with the new possible solutions. XSLT 1.0 and 2.0 are included in the same column.

#### 5.4.2.1. Comparison of the code generators: usage

Functional Requirements	Priority	Xpand	Acceleo	Actifsource	XSLT	FreeMarker
<b>Output:</b> Source code e.g. Java, C++, etc.	HIGH	Yes	Yes	Yes	Yes	Yes
<b>Output:</b> documentation	HIGH	Yes	Yes	Yes	Yes	Yes
<b>Input:</b> XML models	HIGH	Yes	Yes	Yes	Yes	Yes
<b>Input:</b> Java Interface	HIGH	Yes – meta model available in EMF	Yes – meta model available in EMF	Commercial: Can be imported as Ecore model  Free version: No	No – needs to be parsed into xml	Yes – but data should be prepared in java code

Table 3 - Comparison of the code generators' usage, Functional Requirements

Reliability	Priority	Xpand	Acceleo	Actifsource	XSLT	FreeMarker
<b>Wrong output:</b> usable development environment	HIGH	Yes – details in “Usability” table	Yes – details in “Usability” table	Yes – details in “Usability” table	No – details in “Usability” table	Better than XSLT but no connection between template and meta-model
<b>Wrong Input model:</b> modeling support based on meta-model	MEDIUM	Yes – EMF modeling tools	Yes - EMF modeling tools	Yes	N/A - it's used as template engine	N/A - it's used as template engine

<b>Wrong Input model:</b> constraint definition on input models	MEDIUM	Yes – check against meta-model and also “check” language for restriction on model values	Yes – check against meta-model	Yes – check against meta-model and also range aspects for type values	No	No
--	--------	--	--------------------------------	---	----	----

Table 4 - Comparison of the code generators' usage, Reliability

Extendibility	Priority	Xpand	Acceleo	Actifsource	XSLT	FreeMarker
Manual code within the generated code	HIGH	Yes – protected blocks	Yes – Protected blocks and also @generated annotation in Java doc	Yes – Protected blocks	No	No
Usable modeling support	MEDIUM	Yes – EMF modeling tools	Yes - EMF modeling tools	Yes	Yes – depending on the editor	N/A - it's used as template engine

Table 5 - Comparison of the code generators' usage, Extendability

Maintainability of the generated code	Priority	Xpand	Acceleo	Actifsource	XSLT	FreeMarker
Out-of-the-box Traceability from code to input model	HIGH	No – possible to implement manually by accessing meta-mode	Yes	No – possible to implement manually by accessing meta-model	No – should be implemented manually	No – should be implemented manually
usable development environment	HIGH	Yes – details in “Usability” table	Yes – details in “Usability” table	Yes – details in “Usability” table	details in “Usability” table	details in “Usability” table

Table 6 - Comparison of the code generators' usage, Maintainability

<b>Portability of the code generator</b>	<b>Priority</b>	<b>Xpand</b>	<b>Acceleo</b>	<b>Actifsource</b>	<b>XSLT</b>	<b>FreeMarker</b>
Operating system compatibility	HIGH	No restriction	No restriction	No restriction	No restriction	No restriction
Ant/Maven support	HIGH	Yes	Yes	Yes	Yes	Yes

Table 7 - Comparison of the code generators' usage, Portability

<b>Usability of the code generator</b>	<b>Priority</b>	<b>Xpand</b>	<b>Acceleo</b>	<b>Actifsource</b>	<b>XSLT</b>	<b>FreeMarker</b>
Auto generating input model documentation	MEDIUM	Yes – using EMF tools	Yes – using EMF tools	Yes – HTML documentation for MetaModel	Yes – depending on the editor	No
IDE integration for running the code generator	MEDIUM	Yes - Eclipse	Yes- Eclipse	Yes- Eclipse	Yes – depending on the editor	No – should be implemented manually
Modeling support: graphical view / edit	MEDIUM	Yes – EMF modeling tools	Yes - EMF modeling tools	Yes	Yes – depending on the editor	N/A - it's used as template engine
Modeling support: easy text based editing	HIGH	Yes – EMF modeling tools and easy to develop DSL	Yes - EMF modeling tools and easy to develop DSL	Yes	Yes – depending on the editor	N/A - it's used as template engine

Table 8 - Comparison of the code generators' usage, Usability

Resource saving	Priority	Xpand	Acceleo	Actifsource	XSLT	FreeMarker
optimization facilities in development time	MEDIUM	Yes – using EMF tools	Yes –using EMF tools, profiler	No	No	No
Partial code generation based on model changes	MEDIUM	Yes - keeps track of model changes for generating only for the modified models (EMF)	No - but can be implemented for EMF models and also possible to run generator per module	Yes - Updates the output code real time based on changes in model, only changed parts	No	No
Partial code generation based on user selection	MEDIUM	Yes – by setting work flow file	Yes – generation can be run per module	Yes – The code is generated based on model changes automatically	No	No
Caching of the queries to model by the code generator	MEDIUM	Yes	Yes	Yes	No	No

Table 9 - Comparison of the code generators' usage, Resource saving

#### 5.4.2.2. Comparison of the code generators: development

Functional Requirements	Priority	Xpand	Acceleo	Actifsource	XSLT	FreeMarker
<b>Input:</b> XML schema and Java interface	HIGH	Yes – accepts both out of the box	Yes – handles both via EMF	Yes – meta model can be defined and also imported as Ecore model	Input should be XML	Input should be XML or prepared as Java or Python objects

Table 10 - Comparison of the code generators' development, Functional Requirements



<b>Reliability of the development environment</b>	<b>Priority</b>	<b>Xpand</b>	<b>Acceleo</b>	<b>Actifsource</b>	<b>XSLT</b>	<b>FreeMarker</b>
<b>Input:</b> XML schema and Java interface	HIGH	Open source, since 2003, released regularly	Open source, since 2005, released regularly	Both commercial and open source versions, open source shows bug prone	XML schema as meta model accepted by some editors and also the processor itself only in XSLT 2.0	Simply used for transformation, no connection between the meta model and development environment

Table 11 - Comparison of the code generators' development, Reliability

<b>Maintainability and Extendibility of the code generator</b>	<b>Priority</b>	<b>Xpand</b>	<b>Acceleo</b>	<b>Actifsource</b>	<b>XSLT</b>	<b>FreeMarker</b>
Out-of-the-box traceability between meta model and templates	HIGH	Yes	Yes	Yes	No	No
Out-of-the-box traceability between generated code and model, meta model and templates	MEDIUM	No	Yes	No	No	No
Development supports in IDE e.g. Debugging, Error highlight, code completion	HIGH	details in “usability table”	details in “usability table”	details in “usability table”	details in “usability table”	details in “usability table”
Refactoring support	HIGH	Yes – rename, extract templates and also refactoring in extension and Check files	Yes – rename module, template, query and variable, extract up template and query, Pull up modules	Yes – changes in models and meta model reflected to generated code and templates, complex re-factorings handled by aspects	No	No

Readability of the input model	MEDIUM	Depends on the modeling, code highlight and outline available, Can use EMF tools for viewing models	Depends on the modeling, code highlight and outline available, Can use EMF tools for viewing models	Depends on the modeling, code highlight and outline available, graphical view available	depending on the editor (Possible to dump the intermediate trees in XSLT 2.0)	N/A - it's used as template engine
Readability of the templates	HIGH	Readable template language, traceability to meta-model available	Readable template language, traceability to meta-model available	Templates very similar to output, traceability to meta-model available	complicated language, no traceability to meta-models	Readable template language, no traceability to meta-models
Organizing and re-using meta-models	HIGH	Yes - Eclipse	Yes- Eclipse	Yes- Eclipse	No – should be implemented manually	No – should be implemented manually
Aspect oriented programming for templates	MEDIUM	Yes	Yes	Yes	No	No

Table 12 - Comparison of the code generators' development, Maintainability

<b>Improvability and Connectability of the code generator</b>	<b>Priority</b>	<b>Xpand</b>	<b>Acceleo</b>	<b>Actifsource</b>	<b>XSLT</b>	<b>FreeMarker</b>
Supporting different meta-models out of the box	MEDIUM	EMF, Uml2, uml2 profile, Java meta-model, XSD meta-model	XMI format, EMF models by default	Ecore models	No	No
Refactoring support	HIGH	Yes – rename, extract templates and also refactoring in extension and Check files	Yes – rename module, template, query and variable, extract up template and query, Pull up modules	Yes – changes in models and meta model reflected to generated code and templates, complex re-factorings handled by aspects	No	No

Table 13 - Comparison of the code generators' development, Improvability and Connectability

<b>Portability of the generator development environment</b>	<b>Priority</b>	<b>Xpand</b>	<b>Acceleo</b>	<b>Actifsource</b>	<b>XSLT</b>	<b>FreeMarker</b>
Operating system compatibility	LOW	No restriction	No restriction	No restriction	No restriction	No restriction
IDE dependant	LOW	Yes, Eclipse	Yes, Eclipse	Yes, Eclipse	No	No

Table 14 - Comparison of the code generators' development, Portability

<b>Usability</b>	<b>Priority</b>	<b>Xpand</b>	<b>Acceleo</b>	<b>Actifsource</b>	<b>XSLT</b>	<b>FreeMarker</b>
Documentation available	HIGH	Yes	Yes	Not so complete	Yes	Yes
training material available	HIGH	Yes	Yes	Yes	Yes	Yes
Development IDE support: Run in debug mode	MEDIUM	Yes	Yes	Yes	Yes - depending on the editor	Depends on the used editor, API available for implementation in editors
Development IDE support, Editor supports, based on language: code highlight, outline, Errors and warnings	HIGH	Yes	Yes	Yes	Yes - depending on the editor	Yes - depending on the editor
Development IDE support, based on met model: Errors and warnings, code completion	HIGH	Yes	Yes	Yes	Yes - depending on the editor	No
Traceability between code, template, input model and meta-model	MEDIUM	Yes - except traceability from generated code to input model	Yes – all available	Yes - except traceability from generated code to input model	No – only navigation from template to meta model available	No
Graphical meta-modeling environment	MEDIUM	Yes	Yes	Yes	Yes - depending on the editor	No – no connection to meta-model
Special handling for white space	MEDIUM	Yes – distinguishes template white space and the rest e.g. loops indentation	Yes – Highlights the white spaces which will appear in the output	Yes – the templates do not have any additional white spaces because of visual template language e.g. to loops needed	No	No – trimming directives available and also non-outputting white spaces are removed automatically

**Table 15 - Usability**

Financial saving	Priority	Xpand	Acceleo	Actifsource	XSLT	FreeMarker
Commercial/ Free	MEDIUM	Free	Free	Commercial and Free	Free – editors both free and commercial	Free – editors both free and commercial

Table 16 - Financial saving

### 5.4.3. Which tool to choose

Using the same solution for all the code generation tasks in the company can have many benefits. All the developers can use the same learning material, share their problems and have an online community for exchanging their knowledge on code generation. Internal workshops are more profitable since they will cover lots of people. Also there can be a common library for reusing pieces of different code generation solutions which increases the reliability and integrability between different systems.

But looking at side-by-side comparison, there may be some features that make one think about using different tools for different purposes, since each tool may match some tasks better. The tools can have their own specific features which are not available in others and may be useful in solving different problems.

For example looking at Acceleo and Xpand, there are some useful features which are only available in each. Acceleo has a focus on traceability and provides great traceability feature in its development environment. It is possible to trace from each part of the generated code to all the related elements in templates, models and meta-models. It is also possible to find all the generated codes for a specific template or model. This feature is very useful in time of maintenance and extending the code generator.

Xpand does not support such powerful traceability, and provide a more restricted set of such features. But on the other hand, out-of-the-box it provides a great feature regarding performance. It keeps track of changes in the input model and runs generation only for the files that will be affected by the change. So the developer does not have to wait a long time to see the result of a small change. This is also very useful on maintenance and development process and saves a lot of time for the developer. It is even possible to customize the strategy, based on which the

changes are detected. Furthermore Xpand comes with Check language which allows the developer to put any restriction on the input models for code generator.

Actifsource on the other hand provides great usability and refactoring features. All of these features are very useful and only one of them can be used, in case only one tool is chosen. So it depends on the type of task e.g. size and complexity of the input model, to choose the best tool. For example the feature mentioned for Xpand, which is about partial generation, is only important and useful when the code generation input is a large model.

In order to the best tool a task is defined by company, which includes most important requirements of the company's current solutions, stated in 5.1. The first three generation tasks are included in the defined sample task. The side by side comparison table in previous section is based on the practical experiments doing the sample task.

Based on side by side comparison and the priority of each feature, Acceleo is the suggested code generation solution for the following main reasons:

- Full traceability between generated code, templates, input models and meta models
- Modeling is based on Eclipse modeling framework (EMF) which not only is a complete and sophisticated modeling solution, but also makes the code generator solution easily integrable with other solutions and a lot of available modeling tools. Since EMF is a commonly used framework and also its implementation follows standard model driven concepts, it is a safe choice for keeping and developing the modeling of a system on.
- Acceleo is open source, while it has an active development team with regular releases and also active user community
- Caching the queries to models and also profiling features makes it a good choice regarding performance
- The template editor has several useful features like code completion based and type safety, different refactoring features, query testing console, debugging, code highlight and white space handling.

- Lots of free learning materials on the Internet
- Acceleo is an implementation of OMG's standard model-to-text specification

## **6. Results and Conclusion**

### ***6.1. Background study on code generation***

Code generation benefits are categorized into three main groups in this work:

1. Separation of high level abstractions and system code
2. Code generation makes the process of implementation faster
3. Code generation increases the software quality

A comparison between code generators and compilers, as widely used tools for increasing the abstraction of software development, shows that one major difference between them is the fact that compilers do not change the semantics of the input and only translate the same program to a lower level language. Code generators on the other hand may only transform based on some parts of an input model, or may add the logic into input model which they have saved in their templates.

### ***6.2. Tools comparison process***

Looking at different software evaluation methods, the main conclusion is the need for customization of the comparison process based on context and application. All the comparison works include the same main steps like preparation of criteria list and defining the measures for them. But there is no single process or criteria list that can be used as for comparison of all the software.

In this work, the software quality attributes are chosen as initial criteria list which is then customized and simplified based on two main factors:

1. The certain requirements of every code generation solution
2. The problems and requirements of the company, in which the tool is going to be used



As a result of investigating the general requirement of code generation solutions, the criteria are defined for two different target environments. First is the code generator's development environment, in which the tool is developed and the other is the usage environment in which a developer uses the code generator to develop some other software.

The comparison of the criteria is made possible through finding the set of the features in each of the solutions which affect each criterion. The features are recognized during tools review phase, chapter 4. So for each of the criteria, a feature-based side-by-side comparison is provided that shows what features are available for each of the solutions. The full categorization of the criteria is available in section 3.2.5, Code generation comparison criteria.

### ***6.3. Tools comparison result***

The results of comparison between different code generation solutions can be grouped by main concerns of the company. They are stated below for both code generator's usage and development environments, followed by the main characteristics of a proper solution which are called the essential requirements for a code generation solution. Based on the results and company's main requirements and priorities, Acceleo was chosen as the suggested tool 5.4.3.

#### **6.3.1. Code generation usage**

In code generation usage environment, the user should first easily find the place to edit the input model for getting the right output, and also should be able to easily modify the models and then see the generation's result fast enough. So from the code generation's user point of view, traceability from code to the models, maintaining and extending the generated code and also the performance of the code generator are the main concerns.

The readability of the input models and the ease of modifying them, which are basically the important factors for maintainability of the generated code, highly depend on the format of the input models and also the available tools for their manipulation.

Traceability from the generated code to the input model should usually be implemented manually, especially when there is no access to code generator's development environment at the

time of code generator's usage. Access to meta-model information during the development of the code generator is essential for the implementation. Also there are code generator development environments that provide traceability out-of-the-box, from the generated code to input models. This can come handy for the code generator's user when they have access to these development environments e.g. it is integrated to their environment like Eclipse plug-ins.

The response time of the code generator is important especially when there is a large set of input models that also results in lots of code lines. The ability to generate the code based on some parts of the code is essential here. This part can be either based on user's selection or can be the modified part of the input model, detected automatically by the tool.

### **6.3.2. Code generation development**

The main concerns in code generation development environment are understandability of the solution as well as the ease of maintaining it. In order to maintain or extend a code generator, a developer first needs to understand its implementation easily. Readability of the input model and the templates of the code generator help the developer to understand the solution better. Also usability features of the development environment like traceability, running in debug mode or code highlight, not only increase the understandability of the system, but together with good documentation and training materials help with easy development of the solution.

### **6.3.3. Code generator's essential requirements**

Looking at the required features for a good code generation solution, it can be concluded that a main feature of the development environment which is the source of many other features is the connection between the environment and meta-model of the input model. In other words, when the environment understands the meta-model, it can provide many important usability features like code completion and highlight, error and warning detection and traceability. This not only eases the development in such environment, but also makes the output more maintainable and reliable.

Another important point which is again related to modeling is using a standard and widely used modeling framework and meta meta-model. This not only makes the solution connectable and

integrable with other solutions based on this standard framework, but also makes it possible to access more useful tools for manipulation and working with the models. Graphical editors and viewers for model or DSL development tools are the examples of such tools.

Many useful features of the code generator development environment can only be found in the tools and environments which are specifically created for development of code generators. Such environments can provide certain features which should be implemented manually when implementing the code generator by a generic tool or language like XSLT or Java language. Such environments, which are specialized and designed for code generator development, provide readable and easy template languages, designed for this purpose. They may also provide some useful features out-of-the-box like traceability among generated code, templates, input models and meta-models. Another feature available in such environments, which is based on a code generation concepts, is the partial code generation based on the input model changes. This feature is available out-of-the-box in some of these solutions and has a great impact on performance in large code generation projects.

#### **6.3.4. Importance of modeling in code generation**

As it can be seen in the characteristics of a good code generation solution, most of the features are highly dependent on the format of the input model and the framework in which the model is supported and developed. So it is a very important result that when trying to choose a proper code generation solution, the modeling part is very crucial and it is not enough to focus only on the template engine - which is considered as the code generator tool in many cases.

The models are the essential parts of a code generation solution since abstracting out a system's concepts into the models is exactly what code generation is about. So their format and structure has a major impact on the code generation solution and needs special focus and attention during developing or choosing a code generator. This is a good reason why the best available code generation tools can be found as a part of model driven frameworks.

## 7. Evaluation and further works

During this work, each evaluation criterion is compared for different tools based on the features of the tools that affect the criterion. It is possible to quantify each quality aspect more accurate and detailed. In “Competitive engineering: a handbook for systems engineering, requirements engineering” [8] chapter 5, *Scales of Measurei*, is a thorough chapter about quantifying the quality measures. The main reason behind the feature based comparison tables is to have a fast and easy to use comparison framework.

The list of quality aspects of code generation tools are simplified and prioritized based on the company’s requirements. As the result is customized due to these requirements, the final comparison table is not a general purpose table for all possible use cases. Some of the quality aspects which are removed due to their low importance may be important in other use cases. It is possible to find all the features for all the software quality aspects for code generators in a more extensive work.

Furthermore when measuring usability, feedbacks from developers who can work with such tools will be very useful. This can be done by preparing a simple task for the developers that they can use different tools to accomplish such tasks. Gathering the feedbacks from these developers is possible by methods like using questionnaires. A set of quality measures should be prepared as the base for question to the developers.

# Bibliography

- [1] "Success Stories," Object Management Group, Inc., 2 July 2012. [Online]. Available: [http://www.omg.org/mda/products\\_success.htm](http://www.omg.org/mda/products_success.htm). [Accessed 16 November 2012].
- [2] "Fourth-generation Programming Language," 4 August 2011. [Online]. Available: [http://en.wikipedia.org/wiki/Fourth-generation\\_programming\\_language](http://en.wikipedia.org/wiki/Fourth-generation_programming_language). [Accessed 26 November 2012].
- [3] M. Mernik and V. Zumer, "Domain-specific languages for software engineering," *IEEE Computer Society*, 2001.
- [4] *ISO/IEC 9126: Information technology - Software Product Evaluation - Quality characteristics*.
- [5] B. Kitchenham and L. Pickard, "Towards a constructive quality model. Part 2: Statistical techniques for modelling software quality in the ESPRIT REQUEST project.," *Software Engineering Journal* , vol. 2, no. 4, pp. 114-126, 1987.
- [6] T. Gimp, Principles of software engineering management, vol. 4, Wokingham: Addison-Wesley, 1988.
- [7] A. Tsoukias, "LusWare: A Methodology for the Evaluation and Selection of Software Products," *Software Engineering. IEE Proceedings*, vol. 144, no. 3, pp. 162-174, 1997.
- [8] T. Gimp, Competitive engineering: a handbook for systems engineering, requirements engineering, and software engineering using Planguage, Butterworth-Heinemann, 2005.
- [9] "OMG's MetaObject Facility," Object Management Group, Inc., 27 June 2012. [Online]. Available: <http://www.omg.org/mof/>. [Accessed 26 November 2012].
- [10] "Object Management Group," Object Management Group, Inc., [Online]. Available: <http://www.omg.org/>. [Accessed 26 November 2012].
- [11] N. Wirth, "Extended Backus-Naur Form (EBNF)," ISO/IEC 14977 (1996): 2996.
- [12] OMG, "MOF 2 XMI Mapping (XMI)," 9 August 2011. [Online]. Available: <http://www.omg.org/spec/XMI/2.4.1/>. [Accessed 26 November 2012].
- [13] OMG, "Documents Associated With Object Constraint Language, Version 2.0," 01 May

2006. [Online]. Available: <http://www.omg.org/spec/OCL/2.0/>. [Accessed 26 November 2012].
- [14] "List of Eclipse Modeling Framework based software," 10 August 2012. [Online]. Available: [http://en.wikipedia.org/wiki/List\\_of\\_Eclipse\\_Modeling\\_Framework\\_based\\_software](http://en.wikipedia.org/wiki/List_of_Eclipse_Modeling_Framework_based_software). [Accessed 26 November 2012].
- [15] "M2T-JET-FAQ," Eclipse.org, [Online]. Available: <http://wiki.eclipse.org/M2T-JET-FAQ>. [Accessed 26 November 2012].
- [16] "JET FAQ What is JMerge?," [Online]. Available: [http://wiki.eclipse.org/JET\\_FAQ\\_What\\_is\\_JMerge%3F](http://wiki.eclipse.org/JET_FAQ_What_is_JMerge%3F). [Accessed 26 November 2012].
- [17] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT), V1.1," January 2011. [Online]. Available: <http://www.omg.org/spec/QVT/1.1/>. [Accessed 26 November 2012].
- [18] OMG, "MOF Model To Text Transformation Language (MOFM2T), 1.0," January 2008. [Online]. Available: <http://www.omg.org/spec/MOFM2T/1.0/>. [Accessed 26 November 2012].
- [19] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," 15 March 2001. [Online]. Available: <http://www.w3.org/TR/wsdl>. [Accessed 27 November 2012].
- [20] A. van Kesteren and A. Gregor, "DOM4," 5 April 2012. [Online]. Available: <http://www.w3.org/TR/dom/>. [Accessed 27 November 2012].
- [21] G. Seshadri, "Understanding JavaServer Pages Model 2 architecture," JavaWorld. com 12 (1999): 29-99.
- [22] J. Clark, "XSL Transformations (XSLT) Version 1.0," 16 November 1999. [Online]. Available: <http://www.w3.org/TR/xslt>. [Accessed 26 November 2012].
- [23] S. Michael Kay, "XSL Transformations (XSLT) Version 2.0," 23 January 2007. [Online]. Available: <http://www.w3.org/TR/xslt20/>. [Accessed 26 November 2012].
- [24] A. Berglund, M. Fernández, A. Malhotra, J. Marsh, M. Nagy and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)," 14 December 2010. [Online]. Available: <http://www.w3.org/TR/xpath-datamodel/>. [Accessed 26 November 2012].
- [25] "The Financial Information eXchange Protocol, FIXML 4.4 Schema Specification 20040109," 6 October 2006. [Online]. Available:

- <http://www.fixprotocol.org/specifications/fix4.4fixml>. [Accessed 26 November 2012].
- [26] J. Herrington, *Code generation in action*, Manning Publications Co., 2003.
- [27] J. D. Haan, "15 Reasons Why You Should Start Using Model Driven Development," 25 November 2009. [Online]. Available: <http://www.theenterpriseearchitect.eu/archive/2009/11/25/15-reasons-why-you-should-start-using-model-driven-development>. [Accessed 16 November 2012].
- [28] J. D. Haan, "Why Aren't We All Doing Model Driven Development Yet?," 16 April 2011. [Online]. Available: <http://www.theenterpriseearchitect.eu/archive/2011/04/16/why-arenat-we-all-doing-model-driven-development-yet>. [Accessed 16 November 2015].
- [29] R. Pompa, "JET Tutorial Part 2 (Write Code that Writes Code)," 31 May 2004. [Online]. Available: [http://www.eclipse.org/articles/Article-JET2/jet\\_tutorial2.html](http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html). [Accessed 26 November 2012].
- [30] R. Pompa, "JET tutorial part 1 (introduction to JET)," 31 May 2004. [Online]. Available: [http://www.eclipse.org/articles/Article-JET/jet\\_tutorial1.html](http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html). [Accessed 26 November 2012].
- [31] D. Bhattacharyya, "Model-Driven Architecture using JET2 in Rational Software Architect or Rational Software Modeler," 21 January 2010. [Online]. Available: <http://www.ibm.com/developerworks/rational/library/10/modeldrivenarchitectureusingjet2inrsaorrm/index.html>. [Accessed 26 November 2012].
- [32] R. Peterson, "Create powerful custom tools quickly with Rational Software Architect Version 7.0," 9 January 2007. [Online]. Available: [http://www.ibm.com/developerworks/rational/library/07/0109\\_peterson/](http://www.ibm.com/developerworks/rational/library/07/0109_peterson/). [Accessed 26 November 2012].
- [33] ""Creating Transformations and Transformation Extensions." Help - Rational Software Architect," [www.ibm.com](http://pic.dhe.ibm.com/infocenter/rsahelp/v8/index.jsp?topic=/com.ibm.xtools.transfo rm.authoring.doc/topics/ttransauthover.html), [Online]. Available: <http://pic.dhe.ibm.com/infocenter/rsahelp/v8/index.jsp?topic=/com.ibm.xtools.transfo rm.authoring.doc/topics/ttransauthover.html>. [Accessed 26 November 2012].
- [34] J. Musset, "Acceleo User Guide," 2008. [Online]. Available: <http://www.acceleo.org/pages/home/en>. [Accessed 26 November 2012].
- [35] A. Håkansson, "Portal of Research Methods and Methodologies for Research Projects and Degree Projects," in *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP*, Las Vegas USA, 2013.

