**Project 4 – Task 3**

**Code Generation**

|  |  |  |
| --- | --- | --- |
| DATE | : | 25 October 2015 |
| COURSE | : | ADSA Model-driven Engineering |
| COURSE CODE | : | 192135450 |
| GROUP | : | 01 |
| STUDENTS | : | Aaqib Saeed, 1651854  Muhammad Arif Wicaksana, 1507850 |

Table of Contents

[Introduction 3](#_Toc433703360)

[Improvement over Task 2 Result 3](#_Toc433703361)

[Model-to-Text Transformation 4](#_Toc433703362)

[Description 4](#_Toc433703363)

[Design alternatives 4](#_Toc433703364)

[Transformation Template Implementation 4](#_Toc433703365)

[The Examples 6](#_Toc433703366)

[Conclusion 6](#_Toc433703367)

# Introduction

Model-to-text (M2T) transformation is a method to generate textual artifacts from models. It has been used for automating several software engineering tasks such as the generation of documentation, task lists, source code, etc. The most popular application of M2T transformation is for code generation. Code generation aims at generating running code from a higher level model in order to create a working application, very much like compilers are able to produce executable binary files from source code.

In the last part of project 4 (BDD), we are working on the code generation of test cases based on the stories provided by user. The input for the code generation is the result of model transformation from BDD model to Java model in the previous task (model transformations). We use jBehave as the reference to produce the code. As the consequence, the generated code is designed to use the necessary annotations used by jBehave. The result of the M2T transformation is a bunch of test-case code skeletons readily used by developers. In the last section of this report, we demonstrate the capability of our generated code to be directly used in jBehave.

# Improvement over Task 2 Result

We start off with the discussion over improvements that we made from the result of task 2. Previously, we have defined the rules to transform from BDD model to Java model. One of the rule is SentenceVariable2Parameters, which responsible for transforming story’s sentence variable into method’s parameter (argument). Back then, we only assigned the parameter’s name with the name of the story’s variable. This approach missed important information, such as the argument’s type and also makes it difficult to parse the value of the argument, if mentioned in the story. This problem comes partially from the metamodel design, so we solve this by slightly modify the BDD metamodel to add an attribute Value in element Variable to contain the supposed Java method’s argument value, and to add attribute defaultValue to element Parameter in Java metamodel. Now, the transformation rule is changed, as can be seen in Figure 1. **<explanation about type: Java!primitiveType …>:** To capture the data type of the variable, we created a ‘type’ attribute in Variable(BDD metamodel), which is a textual description i.e. int or string. While transformation, we convert that text to Java primitive type. As the end goal is to generate code and variables from BDD metamodel will be converted into parameters of Java metamodel, this information is important to generate data type of parameter while code generation.

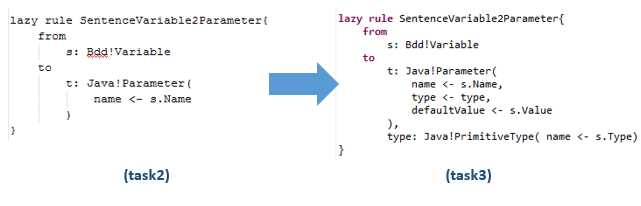


Figure 1 - Modification over lazy rule SentenceVariable2Parameters

# Model-to-Text Transformation

## Description

The result of code generation task is the code skeleton used for jBehave. Thus, we intend to follow the code used by it. At the simplest implementation, jBehave requires two Java files: *steps* file, and *stories* file. The former is the test used to verify the story. It contains the methods for ‘When’, ‘Then’, and ‘Given’ statement with certain annotations. The latter one allows us to run the story processing as JUnit test. Thus, the goal of the code generation is to generate those two files based on a Java model resulted from the model transformation before.

We use the simplest form of the stories file, and it is simple to generate. Most part of it are static text which can be easily hardcoded, and the dynamic texts needed is for the filename, class name, and the reference to the steps file.

The content of steps file is completely dependent over the story. The method name is derived from each ‘When’, ‘Then’, and ‘Given’ sentence. The method should be assigned with input argument if required, and it should be annotated accordingly. Therefore, the template to generate this file needs a lot of references from the Java model in order to fill in the dynamic part.

## Design alternatives

<<Alternatives found>>

## Transformation Template Implementation

For this project, the text-transformation language used is XPand. Both steps and stories files are generated using a single template file (Template.xpt). Firstly, we import the reference metamodel, in this case Java metamodel. Then, we define a code template named main (Figure 2) which is bound to Java element of type Package. This template does not do much, except to call another template, class (explained afterwards), for each element with the type Class in the Java model.



Figure – template main

Template class (Figure 3) is used to generate the body of the Java class for each file. It is bound to the element Class of Java model. Template class consists of two parts: the class body of steps file and of stories file, and we make use of XPand feature to create file (including the appropriate directory) for each of them. The generated file name refers to the name of the referred type Class in the Java model (line 8 and 28 in Figure 3), and the package name is defined statically. Next, we define the appropriate imported jBehave classes for each file. After that, the opening of the class is defined, with the name of the class is directly derived from the name of referred type Class of the Java model and put the appropriate suffix (either ‘Steps’ or ‘Stories’). The content of stories file itself mostly static. To create the methods in steps file, template method is called for all type method encountered in the Java model reference.



Figure 3 - Template class. Left: body of class in steps file. Right: body of class in stories file

Template method (Figure 4) is used to generate the method. The methods in steps file is annotated, thus the first thing to do in this template is to call another template, annotation, to generate the annotation based on the type annotation encountered in the Java model. Then, the body of the method is defined. The name of the method is created by replacing white space in the attribute name of type Method in the Java model with the underscore (\_) character. This is intended to make the method name readable. Then, to generate the input argument of the method (if any), template parameter is called for each type parameters defined in the respective type method in the Java model. The content of the generated method is left empty, since it is up to the developer to fill it in with the appropriate test case.

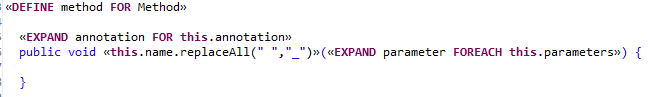


Figure 4 - Template method

Template annotation (Figure 5) is very simple. It refers to the type type of the respective annotation from the Java model to get the annotation name. Then, to fill in the argument of the annotation, the content of sentenceText from the respective element annotation of the Java model is used verbatim.



Figure 5 – Template parameter and annotation

Template parameter (Figure 5) is also simple. To get the type of the argument (int, String, etc), it refers to the name of the element type of the respective parameters from the Java model. To get the argument name, it refers to the name of the respective parameters from the Java model. To get the value of the argument, it refers to the element defaultValue of the respective parameters.

# The Examples

<TBA>

# Conclusion

<TBA>