**­­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](#_Toc434132872)

[Improvement over Task 2 Result 3](#_Toc434132873)

[Model-to-Text Transformation 4](#_Toc434132874)

[Description 4](#_Toc434132875)

[Design alternatives 5](#_Toc434132876)

[Transformation Template Implementation 6](#_Toc434132877)

[Examples 8](#_Toc434132878)

[Conclusion 12](#_Toc434132879)

# Introduction

Model-to-text (M2T) transformation is a method to generate textual artifacts from models. It has been widely used for automating several software engineering tasks such as the generation of documentation, task lists, configuration file, source code, etc. Undoubtedly, 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[[1]](#footnote-1). Once the code is generated, common IDE tools can be used to refine the source code produced during the generation, if necessary, compile it, and finally deploy it.

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 a Java model, which is the result of model transformation from BDD model to Java model in the previous task (model transformations). We use jBehave as the reference of the BDD application. As the consequence, the generated code is designed to use the necessary annotations and classes used by jBehave in order to make it run properly. The result of the M2T transformation is code skeletons readily used by developers. In the last section of this report, we demonstrate the capability of our generated code to be directly tested using 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 lazy rule is SentenceVariable2Parameters, which is 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 of the argument’s type. We overcome this limitation by modifying the rule (Figure 1). To capture the data type of the method’s argument, during transformation, we convert the attribute Type of element Variable of the BDD model, 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.

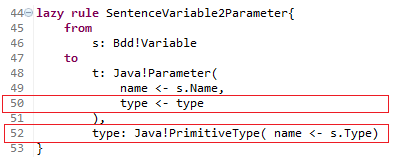


Figure 1 - Modification over lazy rule SentenceVariable2Parameters (additional lines in red boxes)

# 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 its simplest implementation, jBehave requires two Java files: *steps* file (Figure 2), and *stories* file (Figure 3). The former is the test used to verify the story. It contains the methods for ‘When’, ‘Then’, and ‘Given’ statement with certain annotations, and used to define the test steps. The latter one allows us to run the story as a test in a testing framework (TestNG, Spring Test, JUnit, etc). In this work, we choose JUnit, since it is considered as the most popular testing framework in Java. Thus, the goal of the code generation is to generate those two files based on a Java model resulted from the model transformation.

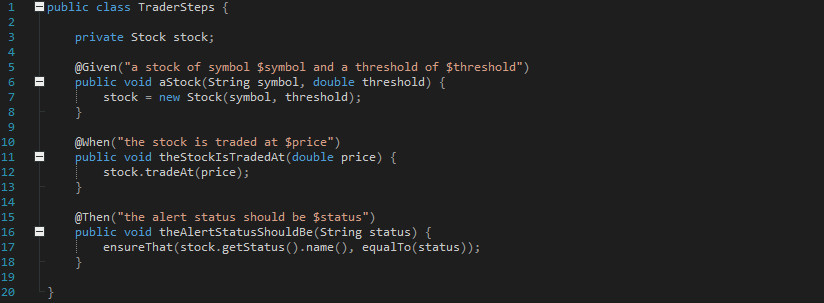


Figure 2 - Example of steps file, taken from jBehave website[[2]](#footnote-2)

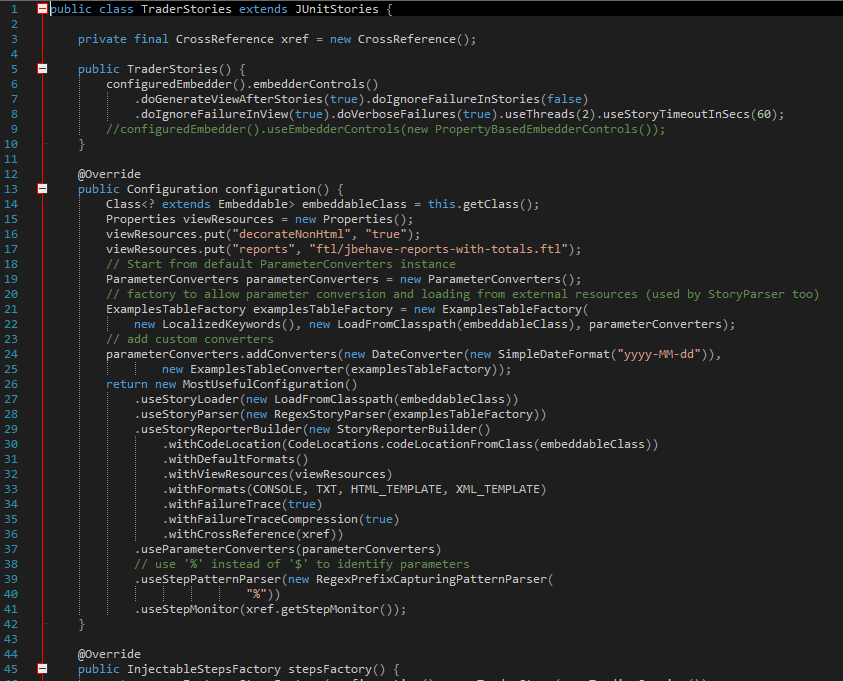


Figure 3 - Excerpt of an example of stories file, taken from jBehave website2

Since model itself is an abstraction, which represents only the essential information of the artifact, consequently model transformation results in information loss. In the case of code generation, the technological details of the generated code are missing. To fill in the lost information, template file is used. There are two types of content in the generated code: *static* and *dynamic* content. Static content can be hardcoded in the template file and generated to the code regardless the source model used. In this work, the example of static content are the required Java syntax (declaration, semicolon, etc), import statements for relevant classes, and so on. The dynamic content depends on the information provided by the source model. Examples are the file name of the generated code, the sentence to fill the annotation, text to define the method’s name, etc.

The content of steps file is mostly 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. The detailed discussion about this file is presented later.

The example of stories file in Figure 3 looks daunting. However, in this work, we will use the simplest form of the stories file to make it concise yet not losing the essence of how jBehave works. Most part of the file are static text which can be easily hardcoded to the template file, and the dynamic texts needed to fill in the ‘hole’ is for the filename, class name, and the reference to the steps file. The detailed discussion about this file is presented later.

## Design alternatives

During the work, alternative solutions were encountered. The first alternative considered is to provide internal implementation of the generated methods. However, we quickly found out that the implementation completely depends on the software being developed, and cannot generated solely based on the story provided. This consideration leads us to generate method skeleton without internal implementations.

The second alternative is related to the generation of meaningful names for methods. As method names are generated from sentences which can be long, we initially tried to follow camel case convention[[3]](#footnote-3), but later abandoned due to lack of string processing functionality in XPand.

The third alternative is related to the Java package names which holds Steps and Stories classes. In the beginning we put both type of classes in a single package, but we found out that it is against JBehave’s convention and makes it difficult to manage large projects with a lot of classes. Thus, we separate both classes into two different packages.

The more detailed design reasoning is explained in the next section.

## 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 (line 1 in Figure 4), in this case Java metamodel (file Java.ecore). Then, we define a code template named main (Figure 4) 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.

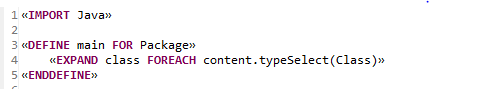


Figure 4 – import statement and declaration of template main

Template class (Figure 5) is used to generate the body of the Java class for each stories and steps file. It is bound to the element Class of Java model. Template class consists of two parts: the class body of steps file (line 8 to 19) and of stories file (line 21 to 51). We start from the body of steps file first.

The filename of steps file (line 8) is based on the Class name of the Java model, where each white space is removed and all words are joined, and finally appended with ‘Steps.java’. The package is defined statically (line 9), and note that the corresponding directory structure is also created in the previous line. Then, the necessary jBehave class needed for the annotations are imported (line 11 to 14). A new annotation haven’t discussed yet is the Named. It is used to annotate the method’s argument name, and refers to the story file. Moving forward, the declaration of the class is defined (line 16 to 18). The class name declaration is similar to the technique used for filename declaration, only without the suffix .java. Not much to do here, except calling template method (explained later) for each element Method defined in the Java model.

Now we move to the stories file. The filename is similar to the steps file name, except that it is appended with ‘Stories.java’ (line 21). The package name is statically defined and is different compared to steps file, hence the different generated directory structure. The declaration of imported classes (line 24 to 33) are static, except for line 33 where it imports the class of steps file above. The class declaration is started in line 35, and the class name declaration is similar to the filename declaration, only without suffix .java. The testing framework used is JUnit, hence the stories class is defined as a subclass of JUnitStory (line 35). We extend JUnitStory, not JUnitStories because in this work we are going to use only a single story. The content of the class (line 36 to 49) basically verifies and tests the story, based on the corresponding steps file (line 48).

As mentioned before, the stories class used here is the simplest one, hence relatively short compared to the example in Figure 3. However, the template can be easily modified and extended to accommodate more complicated stories file, since most of the content is static.

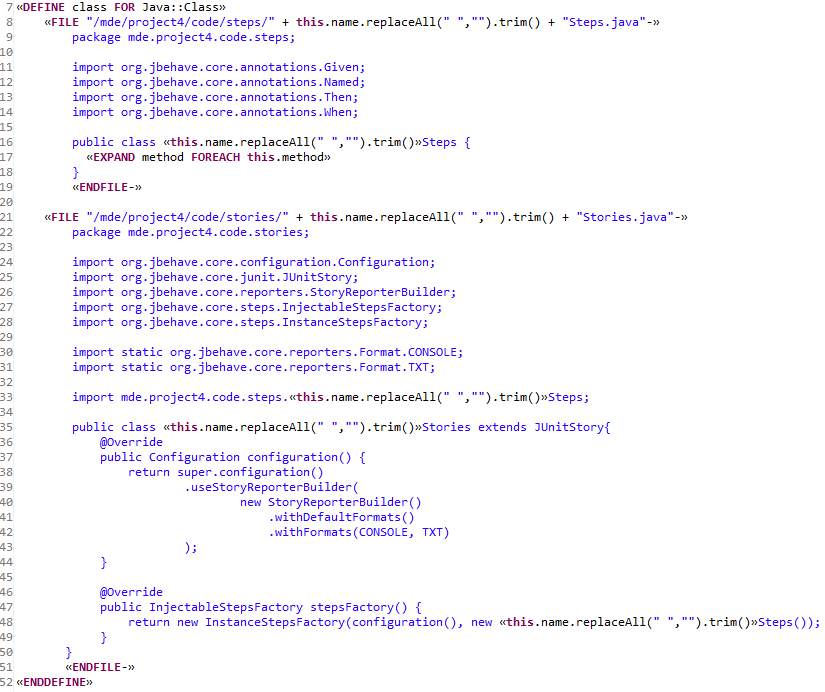


Figure 5 - Template class.

It is also worth to mention that in order to generate source code which is well-structured syntactically and easy to read, we put symbol ‘-‘ right before the end of each «FILE » declaration (end of line 8 and 21), in order to remove unnecessary white space preceding each generated line of code produced by the M2T engine. The visual presentation of generated code is crucial, so that it is as if written by human and so developers are not discouraged to work over them.

Template method (Figure 6) is used to generate the methods in steps file. The methods are annotated, thus the first thing to do in this template is to call another template, annotation (explained later), to generate the annotation based on the respective 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, to make the method name readable by human. Then, to generate the input argument of the method (if any), template parameter (explained later) is called for each type parameters defined in the respective type method in the Java model. If there are more than one arguments, then each argument is separated with comma, except for the last argument. The content of the generated method is left empty, since it is up to the developer to fill it in with the appropriate test steps.

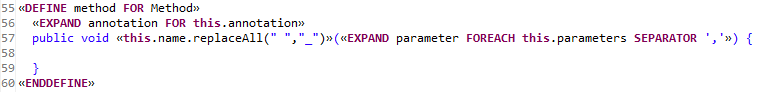


Figure 6 - Template method

Template annotation (Figure 7 line 63) 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 7 - Template parameter and annotation

Template parameter (Figure 7 line 62) generates the argument for the method, if any. Each argument is preceded by annotation @Named.

# Examples

To demonstrate the code generation capability, we develop a simple application and test it using jBehave. Basically, this demo comprises of all phases of this project. Suppose there is a customer ordering a calculator program. Let us call it calculatorApp. The required feature is quite simple. It should perform addition operation of two integer numbers. The customer then writes the requirement in the form of story (Figure 8). The scenario is to add two valid numbers. The story syntax follows jBehave guideline. The variables number1, number2, and result are assigned in the last line (10, 10, and 20 respectively).

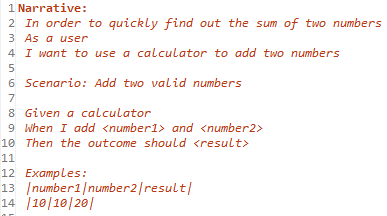


Figure 8 - The story written by the customer

Then, the story is modeled using our BDD metamodel. The result is illustrated in Figure 9. It should be noted that, although the story’s narrative is not represented in the BDD metamodel, this is not a problem. Narrative is only used for human clarity purpose. It is not used during the test, or included in any code. Some relevant attributes not displayed in Figure 9 (left) are the following. The attribute name of Story is “Addition Operation”. The attribute Value of each variable is assigned with the respective value in the story, with attribute Type is set to “int”. After performing model transformation, the resulted Java model is depicted in Figure 9 (right).

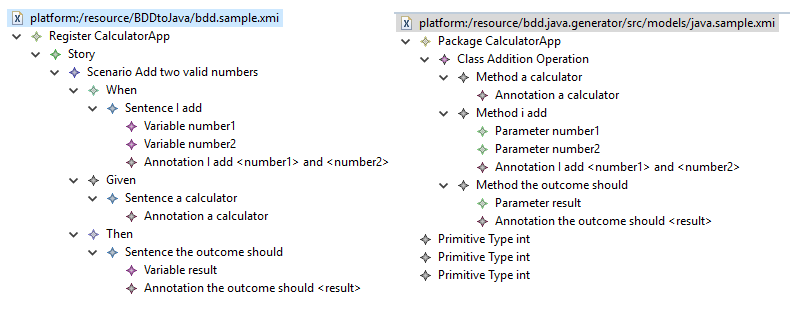


Figure 9 - The BDD model (left), and the Java model resulted from model transformation (right)

The next step is to perform code generation. Using XPand with the template discussed before, the steps file AdditionOperationSteps.java (Figure 11) and the stories class AdditionOperation Steps.java (Figure 12) are generated (Figure 11), including its respective directory structure. Note that the red marks in the code is because the required classes are not available in the code generator project.

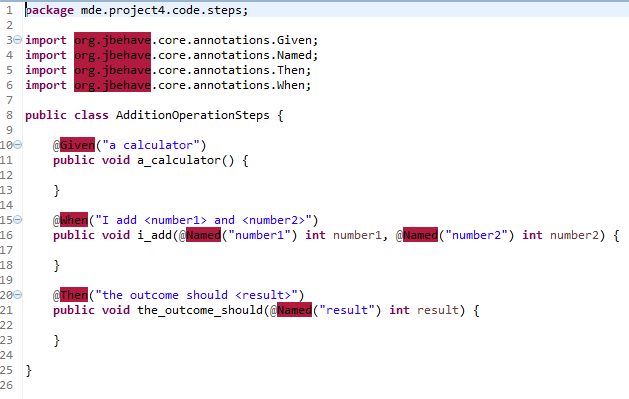


Figure 10 - steps file AdditionOperationSteps.java

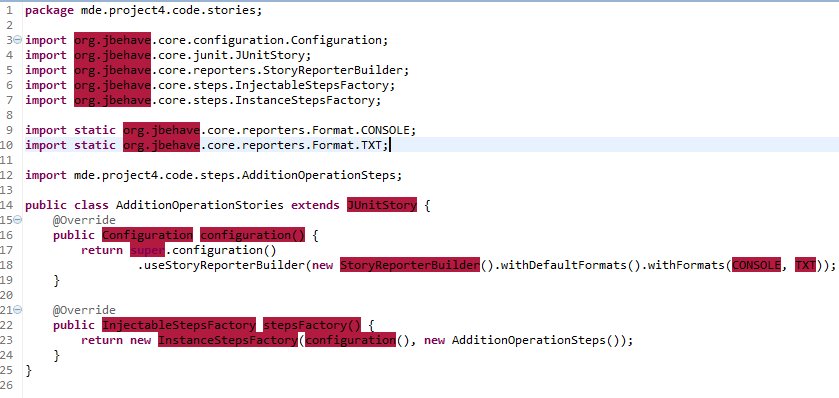


Figure 11 - stories file AdditionOperationStories.java

The generated source code above are just a skeleton. In order to use it for meaningful test, we need to add the test steps in the AdditionOperationSteps.java. Before that, a new project is created, namely calculatorApp (Figure 12). The project is created using Maven, and the required dependencies (jbehave, jbehave-web, and junit) are described in the pom.xml. Then, the whole generated directories above are copied to the src/test/ directory of the calculatorApp project. The story file itself is put in the src/test/resources directory. This is the default directory used by jBehave to locate the story file. Suppose that the calculator application is written as in Figure 13, and the AdditionOperationSteps.java is modified by the developer to include the test steps as in Figure 14.

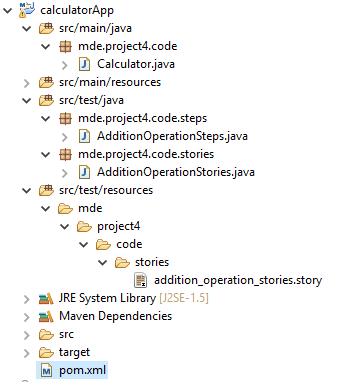


Figure 12 - Directory structure of the CalculatorApp project

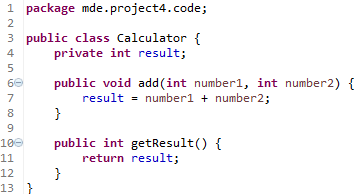


Figure 13 - The program under test, Calculator.java

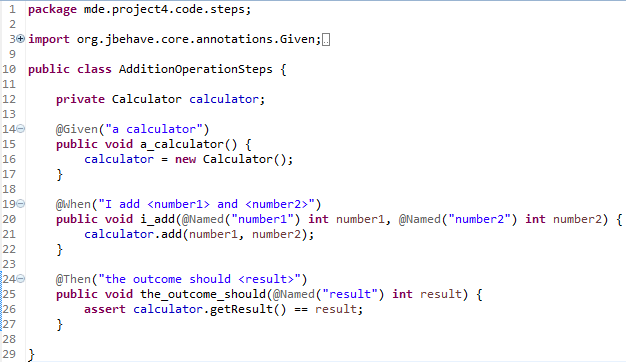


Figure 14 - modified AdditionOperationSteps.java

Then, the project is built using “Maven build”, resulting no error (Figure 15). It shows that the generated code can be directly compiled. As the final step, the project is run as JUnit test. The result is illustrated in Figure 16. jBehave also generates a test report. Since we use jbehave-web, the report is in HTML format, which resides in directory target/jbehave/view. The report is presented in Figure 17.

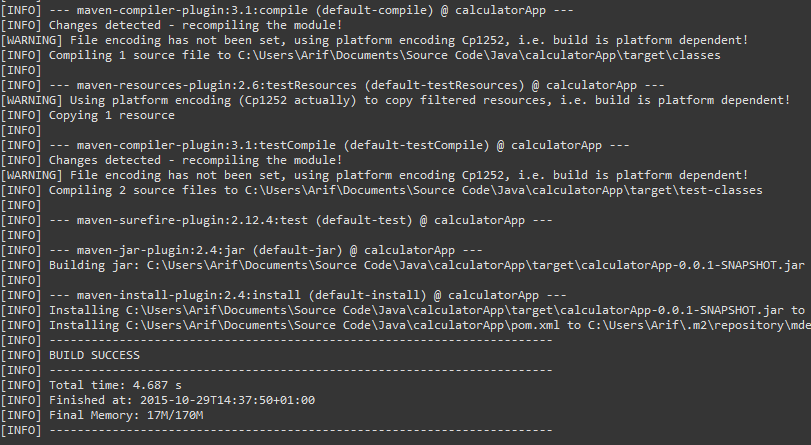


Figure 15 - Project is successfully built using Maven

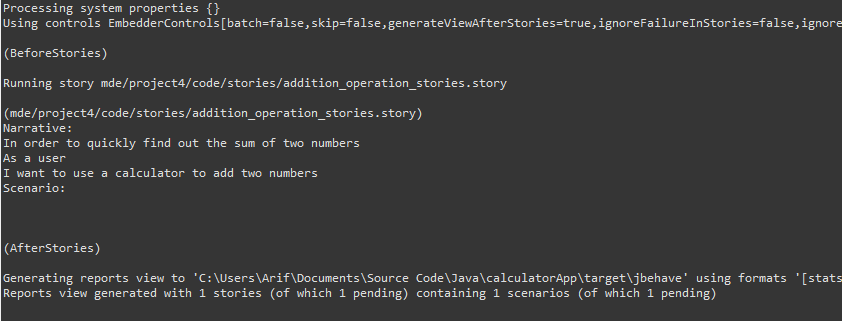


Figure 16 - The result of running the project as JUnit test



Figure 17 - jBehave report

The report shows that the stories ‘*Addition Operation Stories’* is successfully tested. However, it is still unclear what causes the pending. jBehave does not produce enough printout as hint over this. We have tried our best effort to look for in the jBehave documentation to solve this, but to no avail. Regardless, it can be argued that this is more of jBehave implementation problem. The fact that the generated code is successfully compiled and able to run the test without any error shows that the goal of this project is achieved.

All demo files are also submitted together with the report.

# Conclusion

We have presented our work on using MDE techniques to support BDD-based software development by automatically generate jBehave test-case code skeleton based on the story defined by the user. Firstly, the abstraction of stories used in jBehave and Java is performed to create the metamodels. Then, based on the metamodels, transformation rule is defined to create a Java model based on the given BDD model. And finally, the template for code generation is defined to generate the code skeleton based on the Java model resulted from model transformation. This work certainly helps the developer which utilize BDD, because they do not need to write the steps and stories file from scratch anymore. By making model from the customer’s story, the developer can get the needed code automatically, hence saving time and potential human error. Through this demonstration, it clearly can be seen that MDE offers a lot of benefits for software development process.

This work can be extended further. Currently, the developer still needs to create the model based on the customer’s story. The next potential improvement is by utilizing the concrete modeling tool such as XText to create the concrete textual syntax of the BDD. The concrete syntax can be designed to completely resemble the way jBehave defines a valid story. In this way, customer may write the story using the concrete modeling tool, and the resulted story can be directly transformed into Java model, and then to be used to generate the code skeleton. Thus, developers can be detached from the test file coding routines (as the code skeleton provides) and story development activities, and able to focus on developing the application and the required tests.

# Deliverables

There are one report and two folders are submitted. The first folder, ‘bdd.java.generator’, is the model-to-text project directory used for this report. The second one, ‘calculatorApp’, is the project file used for the example.

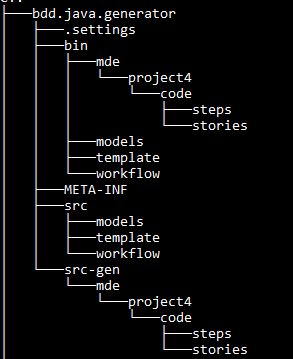


Figure 18 - Structure of model-to-text project directory

In bdd.java.generator (Figure 18), the template file Template.xpt is located in directory src/template. The source model Java.sample.xmi and the Java metamodel Java.ecore is located in src/models. The generated code is located in directory src-gen.

1. Model-Driven Software Engineering, Marco Brambila et al., Morgan & Claypool, 2012. [↑](#footnote-ref-1)
2. http://jbehave.org/reference/stable/developing-stories.html [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/CamelCase [↑](#footnote-ref-3)