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

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]. 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.

```
44 lazy rule SentenceVariable2Parameter{
45     from
46         s: Bdd!Variable
47     to
48         t: Java!Parameter(
49             name <- s.Name,
50             type <- type
51         ),
52         type: Java!PrimitiveType( name <- s.Type)
53 }
```

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

[1] Model-Driven Software Engineering, Marco Brambila et al., Morgan & Claypool, 2012.

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.

```java
public class TraderSteps {

    private Stock stock;

    @Given("a stock of symbol $symbol and a threshold of $threshold")
    public void aStock(String symbol, double threshold) {
        stock = new Stock(symbol, threshold);
    }

    @When("the stock is traded at $price")
    public void theStockIsTradedAt(double price) {
        stock.tradeAt(price);
    }

    @Then("the alert status should be $status")
    public void theAlertStatusShouldBe(String status) {
        ensureThat(stock.getStatus().name(), equalTo(status));
    }

}
```

Figure 2 - Example of steps file, taken from `jBehave` website[2]

```java
public class TraderStories extends JUnitStories {

    private final CrossReference xref = new CrossReference();

    public TraderStories() {
        configuredEmbedder().embedderControls()
            .doGenerateViewAfterStories(true).doIgnoreFailureInStories(false)
            .doIgnoreFailureInView(true).doVerboseFailures(true).useThreads(2).useStoryTimeoutInSecs(60);
        //configuredEmbedder().useEmbedderControls(new PropertyBasedEmbedderControls());
    }

    @Override
    public Configuration configuration() {
        Class<? extends Embeddable> embeddableClass = this.getClass();
        Properties viewResources = new Properties();
        viewResources.put("decorateNonHtml", "true");
        viewResources.put("reports", "ftl/jbehave-reports-with-totals.ftl");
        // Start from default ParameterConverters instance
        ParameterConverters parameterConverters = new ParameterConverters();
        // factory to allow parameter conversion and loading from external resources (used by StoryParser too)
        ExamplesTableFactory examplesTableFactory = new ExamplesTableFactory(
            new LocalizedKeywords(), new LoadFromClasspath(embeddableClass), parameterConverters);
        // add custom converters
        parameterConverters.addConverters(new DateConverter(new SimpleDateFormat("yyyy-MM-dd")),
                new ExamplesTableConverter(examplesTableFactory));
        return new MostUsefulConfiguration()
            .useStoryLoader(new LoadFromClasspath(embeddableClass))
            .useStoryParser(new RegexStoryParser(examplesTableFactory))
            .useStoryReporterBuilder(new StoryReporterBuilder()
                .withCodeLocation(CodeLocations.codeLocationFromClass(embeddableClass))
                .withDefaultFormats()
                .withViewResources(viewResources)
                .withFormats(CONSOLE, TXT, HTML_TEMPLATE, XML_TEMPLATE)
                .withFailureTrace(true)
                .withFailureTraceCompression(true)
                .withCrossReference(xref))
            .useParameterConverters(parameterConverters)
            // use '%' instead of '$' to identify parameters
            .useStepPatternParser(new RegexPrefixCapturingPatternParser(
                                "%"))
            .useStepMonitor(xref.getStepMonitor());
    }

    @Override
    public InjectableStepsFactory stepsFactory() {
```

Figure 3 - Excerpt of an example of stories file, taken from `jBehave` website[2]

[2] `http://jbehave.org/reference/stable/developing-stories.html`

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], 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.

[3] `https://en.wikipedia.org/wiki/CamelCase`

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.

```
1 «IMPORT Java»
2
3 «DEFINE main FOR Package»
4     «EXPAND class FOREACH content.typeSelect(Class)»
5 «ENDDEFINE»
```

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.

```
7  «DEFINE class FOR Java::Class»
8      «FILE "/mde/project4/code/steps/" + this.name.replaceAll(" ","").trim() + "Steps.java"-»
9          package mde.project4.code.steps;
10
11         import org.jbehave.core.annotations.Given;
12         import org.jbehave.core.annotations.Named;
13         import org.jbehave.core.annotations.Then;
14         import org.jbehave.core.annotations.When;
15
16         public class «this.name.replaceAll(" ","").trim()»Steps {
17           «EXPAND method FOREACH this.method»
18         }
19         «ENDFILE-»
20
21     «FILE "/mde/project4/code/stories/" + this.name.replaceAll(" ","").trim() + "Stories.java"-»
22         package mde.project4.code.stories;
23
24         import org.jbehave.core.configuration.Configuration;
25         import org.jbehave.core.junit.JUnitStory;
26         import org.jbehave.core.reporters.StoryReporterBuilder;
27         import org.jbehave.core.steps.InjectableStepsFactory;
28         import org.jbehave.core.steps.InstanceStepsFactory;
29
30         import static org.jbehave.core.reporters.Format.CONSOLE;
31         import static org.jbehave.core.reporters.Format.TXT;
32
33         import mde.project4.code.steps.«this.name.replaceAll(" ","").trim()»Steps;
34
35         public class «this.name.replaceAll(" ","").trim()»Stories extends JUnitStory{
36             @Override
37             public Configuration configuration() {
38                 return super.configuration()
39                         .useStoryReporterBuilder(
40                                 new StoryReporterBuilder()
41                                     .withDefaultFormats()
42                                     .withFormats(CONSOLE, TXT)
43                         );
44             }
45
46             @Override
47             public InjectableStepsFactory stepsFactory() {
48                 return new InstanceStepsFactory(configuration(), new «this.name.replaceAll(" ","").trim()»Steps());
49             }
50         }
51         «ENDFILE-»
52 «ENDDEFINE»
```

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.

```
55 «DEFINE method FOR Method»
56   «EXPAND annotation FOR this.annotation»
57   public void «this.name.replaceAll(" ","_")»(«EXPAND parameter FOREACH this.parameters SEPARATOR ','») {
58
59   }
60 «ENDDEFINE»
```

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.

```
62 «DEFINE parameter FOR Parameter»@Named("«this.name.trim()»")«this.type.name.trim()» «this.name.trim()»«ENDDEFINE»
63 «DEFINE annotation FOR Annotation»@«this.type.trim()»("«this.sentenceText.trim()»")«ENDDEFINE»
```

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).

```
1  Narrative:
2  In order to quickly find out the sum of two numbers
3  As a user
4  I want to use a calculator to add two numbers
5
6  Scenario: Add two valid numbers
7
8  Given a calculator
9  When I add <number1> and <number2>
10 Then the outcome should <result>
11
12 Examples:
13 |number1|number2|result|
14 |10|10|20|
```

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).

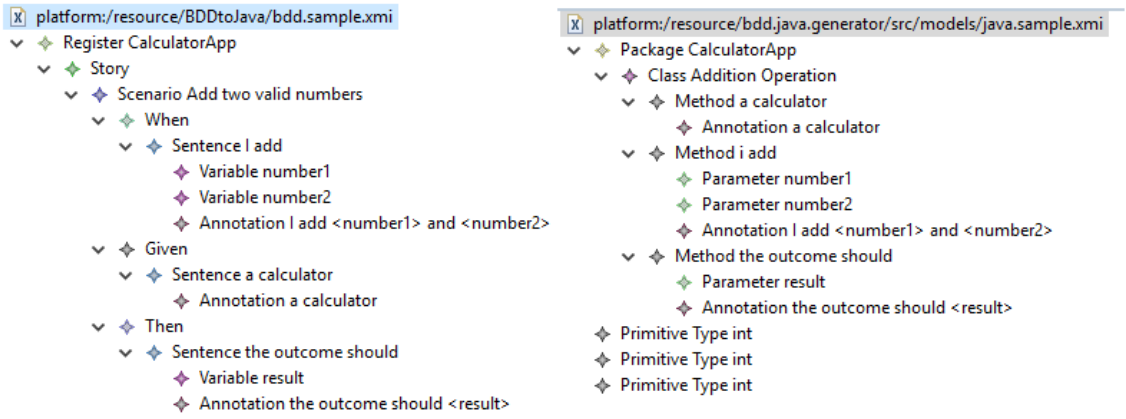

Figure 9 - The BDD model (left), and the Java model resulted from model transformation (right)

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 `AdditionOperationSteps.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.

```
1  package mde.project4.code.steps;
2
3  import org.jbehave.core.annotations.Given;
4  import org.jbehave.core.annotations.Named;
5  import org.jbehave.core.annotations.Then;
6  import org.jbehave.core.annotations.When;
7
8  public class AdditionOperationSteps {
9
10     @Given("a calculator")
11     public void a_calculator() {
12
13     }
14
15     @When("I add <number1> and <number2>")
16     public void i_add(@Named("number1") int number1, @Named("number2") int number2) {
17
18     }
19
20     @Then("the outcome should <result>")
21     public void the_outcome_should(@Named("result") int result) {
22
23     }
24
25 }
26
```

Figure 10 - steps file `AdditionOperationSteps.java`

```java
package mde.project4.code.stories;

import org.jbehave.core.configuration.Configuration;
import org.jbehave.core.junit.JUnitStory;
import org.jbehave.core.reporters.StoryReporterBuilder;
import org.jbehave.core.steps.InjectableStepsFactory;
import org.jbehave.core.steps.InstanceStepsFactory;

import static org.jbehave.core.reporters.Format.CONSOLE;
import static org.jbehave.core.reporters.Format.TXT;

import mde.project4.code.steps.AdditionOperationSteps;

public class AdditionOperationStories extends JUnitStory {
    @Override
    public Configuration configuration() {
        return super.configuration()
                .useStoryReporterBuilder(new StoryReporterBuilder().withDefaultFormats().withFormats(CONSOLE, TXT));
    }

    @Override
    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(configuration(), new AdditionOperationSteps());
    }
}

```

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.

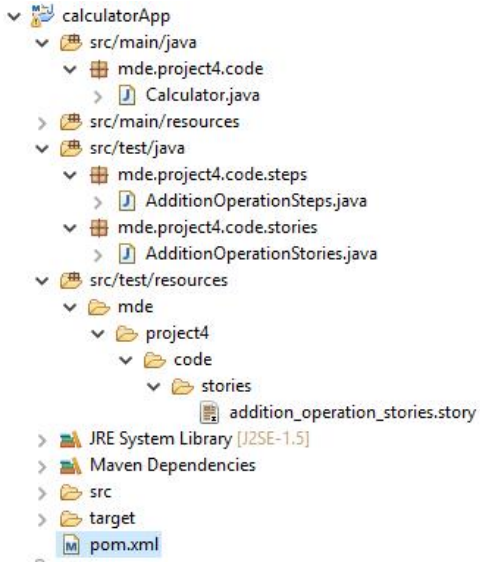

Figure 12 - Directory structure of the `CalculatorApp` project

```java
package mde.project4.code;

public class Calculator {
    private int result;

    public void add(int number1, int number2) {
        result = number1 + number2;
    }

    public int getResult() {
        return result;
    }
}
```

Figure 13 - The program under test, `Calculator.java`

```java
package mde.project4.code.steps;

import org.jbehave.core.annotations.Given;

public class AdditionOperationSteps {

    private Calculator calculator;

    @Given("a calculator")
    public void a_calculator() {
        calculator = new Calculator();
    }

    @When("I add <number1> and <number2>")
    public void i_add(@Named("number1") int number1, @Named("number2") int number2) {
        calculator.add(number1, number2);
    }

    @Then("the outcome should <result>")
    public void the_outcome_should(@Named("result") int result) {
        assert calculator.getResult() == result;
    }

}
```

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.

```
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ calculatorApp ---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
[INFO] Compiling 1 source file to C:\Users\Arif\Documents\Source Code\Java\calculatorApp\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ calculatorApp ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ calculatorApp ---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
[INFO] Compiling 2 source files to C:\Users\Arif\Documents\Source Code\Java\calculatorApp\target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ calculatorApp ---
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ calculatorApp ---
[INFO] Building jar: C:\Users\Arif\Documents\Source Code\Java\calculatorApp\target\calculatorApp-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ calculatorApp ---
[INFO] Installing C:\Users\Arif\Documents\Source Code\Java\calculatorApp\target\calculatorApp-0.0.1-SNAPSHOT.jar to
[INFO] Installing C:\Users\Arif\Documents\Source Code\Java\calculatorApp\pom.xml to C:\Users\Arif\.m2\repository\mde
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.687 s
[INFO] Finished at: 2015-10-29T14:37:50+01:00
[INFO] Final Memory: 17M/170M
[INFO] ------------------------------------------------------------------------
```

Figure 15 - Project is successfully built using Maven

```
Processing system properties {}
Using controls EmbedderControls[batch=false,skip=false,generateViewAfterStories=true,ignoreFailureInStories=false,ignore

(BeforeStories)

Running story mde/project4/code/stories/addition_operation_stories.story

(mde/project4/code/stories/addition_operation_stories.story)
Narrative:
In order to quickly find out the sum of two numbers
As a user
I want to use a calculator to add two numbers
Scenario:



(AfterStories)

Generating reports view to 'C:\Users\Arif\Documents\Source Code\Java\calculatorApp\target\jbehave' using formats '[stats
Reports view generated with 1 stories (of which 1 pending) containing 1 scenarios (of which 1 pending)
```

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.

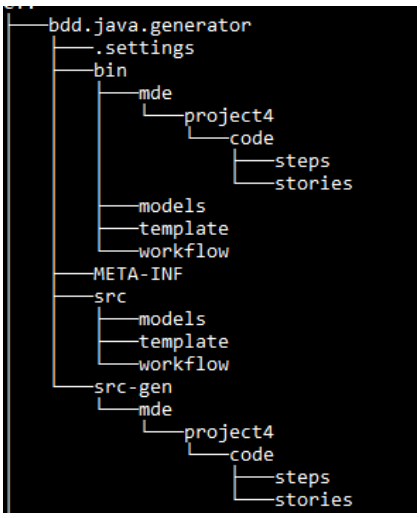

Figure 18 - Structure of model-to-text project directory

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`.