Project 4 - Task 2

Model Transformation from BDD to Simplified Programming Language

DATE	:	8 October 2015
COURSE	:	ADSA Model-driven Engineering
COURSE CODE	:	192135450
GROUP	:	01
STUDENTS	:	Aaqib Saeed, 1651854 Muhammad Arif Wicaksana, 1507850

Table of Contents

Introduction

In Model-driven engineering, model transformation is a technique to transform one model to another based on predefined transformation rule, in an automatic way. This method is very useful because it allows us to automate the building of a target model, hence saving the effort and reduce potential error.

In the second part of project 4, we continue what we left off from the first task. After defining the metamodel of BDD and simplified programming language, now we are going to discuss about the transformation from a BDD model into the model of simplified programming language. This step is required to produce the Java model, which would be the input for the Java source code generation in the last part of this project.

In order to better understand our transformation, it is worth mentioning about the big picture of our project. As stated before, the goal is to automatically generate Java source code based on user story. The source code becomes the skeleton used by developers to create the test cases. In order to allow it directly compiled, we plan to generate source code skeleton of `jBehave`. Thus, the generated code mimics the Java code used in `jBehave`.

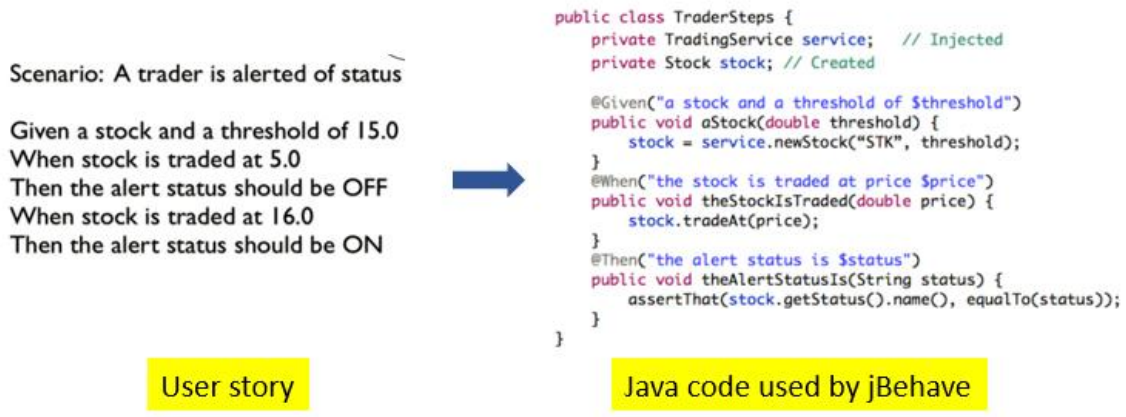

Figure 1 - User story (left), and the expected auto-generated Java code (right)

Figure 1 shows the example from `jBehave` website about a user story, and the Java code used. The Java code with such pattern is the goal of our project. The story becomes the Java class. Then, each sentence of 'given', 'when', and 'then' is translated into a method which annotated with its respective type. The annotation itself use the sentence as its variable. These transformation requirements are the motivation behind our improvement of the metamodel and the base of our transformation rule.

Improvements over Metamodel from Task 1

Before discussing the transformation, it is important to mention the changes we have made over the metamodel from the first part of the project.

Updated BDD metamodel

In current BDD metamodel (Figure 2), three classes are added: `Register`, `Variable`, and `Annotation`. Firstly, `Register` becomes the top-level class and is used as a container of class `Story`. It comes from the fact that a software project may consists of several stories. It is equivalent to Java package. Secondly, class `Sentence` now contains `Annotation` and `Variable`. We discuss the `Variable` first. This class is used to represent the variable assignment from the story. As for `Annotation`, this is a workaround to provide the information to the target model about the type of annotation should be provided for each method (in `jBehave`, each method is annotated with its respective clause; either `@when, @given`, or `@then`).

Ideally, the information about which annotation should be used can be provided by checking on which class (`When, Given`, or `Then`) a `Sentence` is contained, and the user does not need to specify which annotation should be used. However, to our best effort, we cannot implement this yet in our transformation rules. Thus, our current approach is by adding new class `Annotation` with containment relation from class `Sentence`.

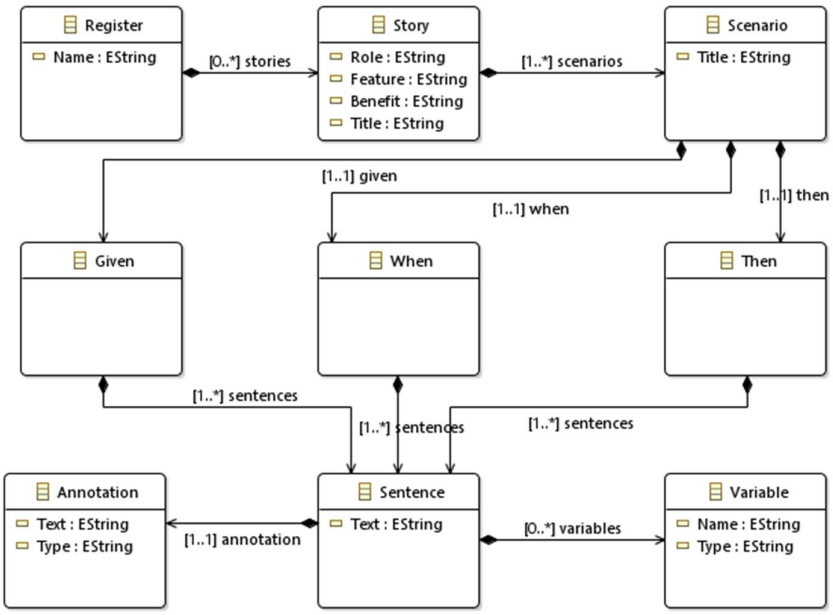

Figure 2 - Updated BDD metamodel

Updated simplified programming language metamodel

Compared to our previous metamodel, significant changes has been made over the Java metamodel. We take inspiration from the Java metamodel provided during the exercise session, because we consider it better represents the Java code. We add new class, `Annotation`, to accommodate the annotation used in the generated source code later. Each `Method` may contain one `Annotation` at maximum. The diagram of the updated Java metamodel is depicted in Figure 3.

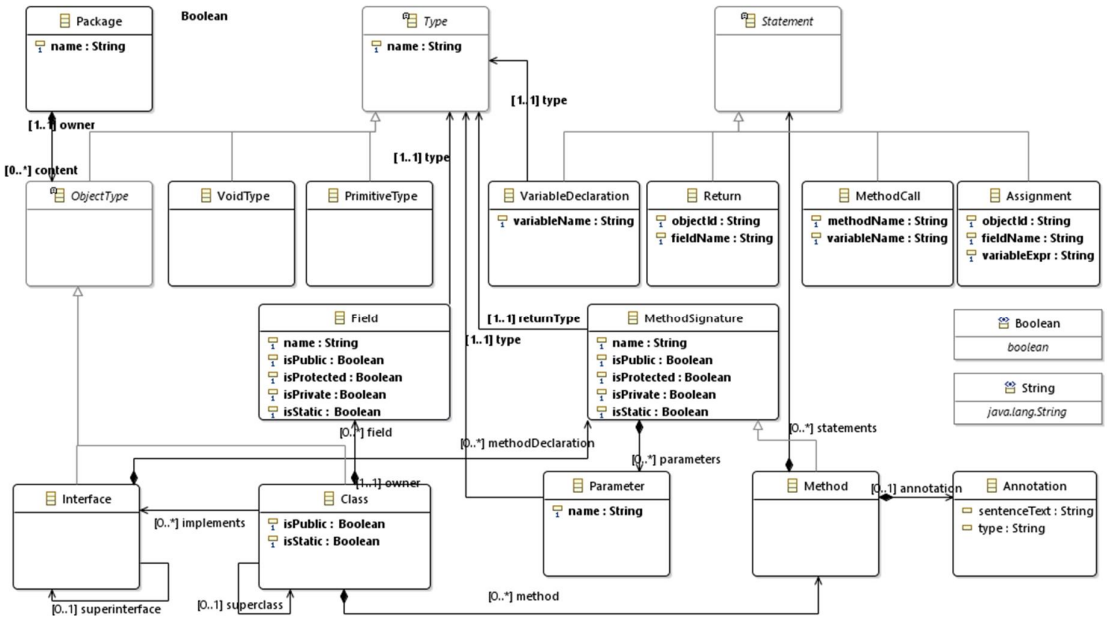

Figure 3 - Improved simplified programming language metamodel

Description of Model Transformation

The idea of the transformation is as the following. Each user story is equivalent to Java class. Each sentence from `Given`, `When`, and `Then` is equivalent to a Java method (this is the way `jBehave` works). Then, each method will be annotated based on the annotation type given by the user. If the sentence contains variable, it will be transformed into method's parameter. Finally, all Java classes in target model is contained in a package, which is the transformation result from BDD register.

In order to perform the transformation, there are two alternatives language available: ATL and QVT. After doing some research, we decided to use ATL for this project. The reason is that we feel more comfortable with the way ATL works, and we consider it to be simpler to use than QVT.

In this project, the transformation is illustrated in Figure 4. Model transformation is done by the ATL transformation engine. The engine requires source model (`bdd.sample.xmi`, explained in the section Example) and the transformation rule `bddToJava.atl`. This ATL file is the main discussion in this section. The metamodel of BDD (`bdd.ecore`) and Java (`java.ecore`) described in the previous section are used by the ATL file as the reference to define the rule. Finally, the engine will generate the output of the transformation, which we define as `java.sample.xmi`, and we use it in the section Example.

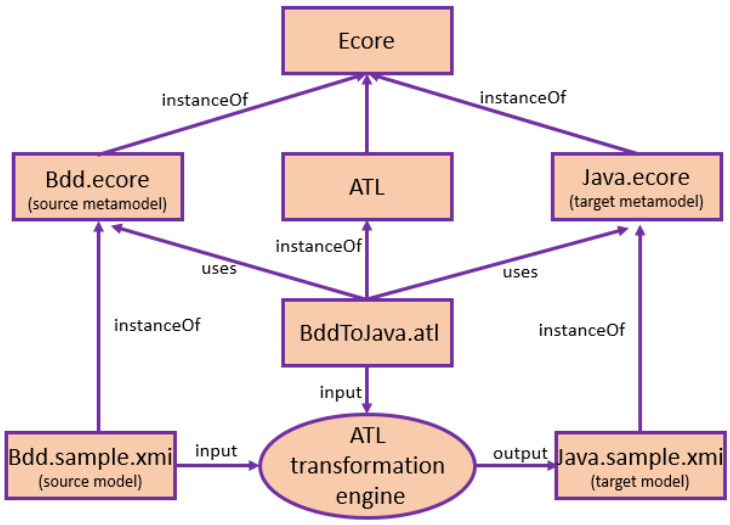

Figure 4 - Model transformation in this project

To implement the transformation description above, there are five transformation rules and some helper functions defined as described below.

Register to Java package

We define the transformation from BDD register to Java package as the rule `BDDRegister2JavaPackage` (Figure 5). The source is BDD's `Register`, and the target is Java's `Package`. Class `Package` contains a single attribute, `name`, which is directly assigned from attribute `Name` in `Register`. As can be seen in Figure 5, Package has relation named `content` to `ObjectType`, of which the parent class of `Class`. To define that the Java package contains Java class, which is the result of transformation from BDD story, `content` is assigned with the result of transformation of `Story` to `Class`.

```
rule BddRegister2JavaPackage {
    from
        s: Bdd!Register
    to
        t: Java!Package (
            name <- s.Name,
            content <- s.stories->select(i | i.oclIsTypeOf(Bdd!Story))->
                       collect(e | thisModule.Story2Class(e))
            )
}
```

Figure 5 – BDD register to Java package

User story to Java class

In order to transform user's story to Java class, we define rule `Story2Class` (Figure 6). Here, the source of transformation is class `Story` of BDD metamodel, and the target is class `Class` of Java metamodel.

The attribute name of `Class` is directly assigned with the attribute `Title` of `Story`. The property `isPublic` is set to `true` to mark the access modifier of the class as `public` for later reuse in the code generation. For the Java class methods, we write helper functions (explained later) to collect `Given`, `When` and `Then` components for each scenario. Next, we collect all the sentences from `Given`, `When` and `Then` components. These collected sentences will be transformed into methods. The important point to note here is that, systems functionality will be provided by different scenarios, of which may contain same sentences. The end goal is to generate code that follow the DRY (Don't repeat yourself) approach and achieve the behavior as proposed by the scenarios.

```
lazy rule Story2Class{
    from
        s: Bdd!Story
    to
        t:Java!Class (name <- s.Title,
            isPublic <- true,
            method <- s.getGiven()->collect(i | i.generateGivenMethods()).
                        append(s.getWhen()->collect(i | i.generateWhenMethods())).
                        append(s.getThen()->collect(i | i.generateThenMethods()))
        )
}
```

Figure 6 – User story to Java class

Sentence to method

Each sentence is translated into a Java method. This requirement is implemented by rule `Sentence2Method` (Figure 7). BDD's `Sentence` is used as the source of transformation. The attribute `isPublic` of class `Method` is always assigned as `true`, to indicate that all generated methods are public. The attribute `name` is assigned using helper class `getMethodName` (explained later in helpers section). The attribute `parameters` is assigned with the transformation result of BDD's `Variable` into Java's `Parameter` (discussed afterward). Annotation is assigned using `SentenceAnnotation2JavaAnnotation` rule. Annotation is added to model the method annotations to represent, to which component (Given, When and Then) the methods belong. The annotation is required in the generated Java code as it is required by `jBehave` libraries.

```
lazy rule Sentence2Method{
    from
        s: Bdd!Sentence
    to
        t: Java!Method(
            isPublic <- true,
            name <- s.getMethodName(),
            parameters <- s.variables->collect(v | thisModule.SentenceVariable2Parameter(v)),
            annotation <- thisModule.SentenceAnnotation2JavaAnnotation(s.annotation)
        )
}
```

Figure 7 – Sentence to Java method

Sentence variable to method parameter

The transformation from sentence's variable to method's parameter is straightforward (rule `SentenceVariable2Parameter`, Figure 8). Here, the attribute `name` of `Parameter` is simply assigned from Variable's `name`.

```
lazy rule SentenceVariable2Parameter{
    from
        s: Bdd!Variable
    to
        t: Java!Parameter(
            name <- s.Name
        )
}
```

Figure 8 – Variable from sentence to parameter

Helpers

The helper functions are defined to structure the transformation and to increase reuse. The complete list of helper function used is depicted in Figure 9. Basically, there are three types of helpers used here:

1. Helpers to collect individual components of stories from scenarios
   Three helper functions are defined to collect `Given, When,` and `Then` components from all stories, named after `getGiven`, `getWhen,` and `getThen` respectively, and return them as sequence. These helpers are used in the rule `Story2Class`.
2. Helpers for generating methods from sentences
   Three helper functions, `generateGivenMethods`, `generateWhenMethods` and `generateThenMethods` are used to transform each sentence of `Given`, `When` and `Then` components respectively, into a method using `Sentence2Method` lazy rule.
3. A helper to generate modified method name
   The helper function `getMethodName` is used to get lowercase method name from the sentence text. It can be further refined to get more meaningful method name.

```
---To generate modified method name
helper context Bdd!Sentence def: getMethodName() : String =
    self.Text.toLower();

---For collecting indvidual components of stories from scenarios
helper context Bdd!Story def : getGiven() : Sequence(Bdd!Given) =
    self.scenarios->collect(e | e.given)->asSequence();

helper context Bdd!Story def : getWhen() : Sequence(Bdd!When) =
    self.scenarios->collect(e | e.when)->asSequence();

helper context Bdd!Story def : getThen() : Sequence(Bdd!Then) =
    self.scenarios->collect(e | e.then)->asSequence();

---For generating methods from sentences
helper context Bdd!Given def : generateGivenMethods() : Sequence(Java!Method) =
    self.sentences->collect(i | thisModule.Sentence2Method(i));

helper context Bdd!When def : generateWhenMethods() : Sequence(Java!Method) =
    self.sentences->collect(i | thisModule.Sentence2Method(i));

helper context Bdd!Then def : generateThenMethods() : Sequence(Java!Method) =
    self.sentences->collect(i | thisModule.Sentence2Method(i));
```

Figure 9 – List of helpers used by the rules

Example

In this section, we provide an example to verify and to make clearer the transformation described before.

Suppose a software company is developing a banking application. The customer wants to describe the expected behavior of the application in the event of withdrawing money from a bank account. There are two cases here. First, when the account holder has sufficient balance in his account.

Second, when the balance is insufficient. These two cases are written as two scenario. To make it more concrete, the account balance is assigned with $100. For the first scenario, the account holder wishes to draw $20. Since the balance is sufficient, the ATM should dispense $20, and the account should be deducted to $80. For the second scenario, the holder wants to withdraw $110, which is exceeding his account balance. Hence, the ATM should not dispense any money, and the account balance should remain intact.

The description of the story above is modeled in our BDD as depicted in Figure 10. Please note that some attributes of the model element are not displayed there. For example, in `Story`, the attribute Title is assigned with '*Check Account*'. This important because this attribute is used to assign the class name of the target Java model.

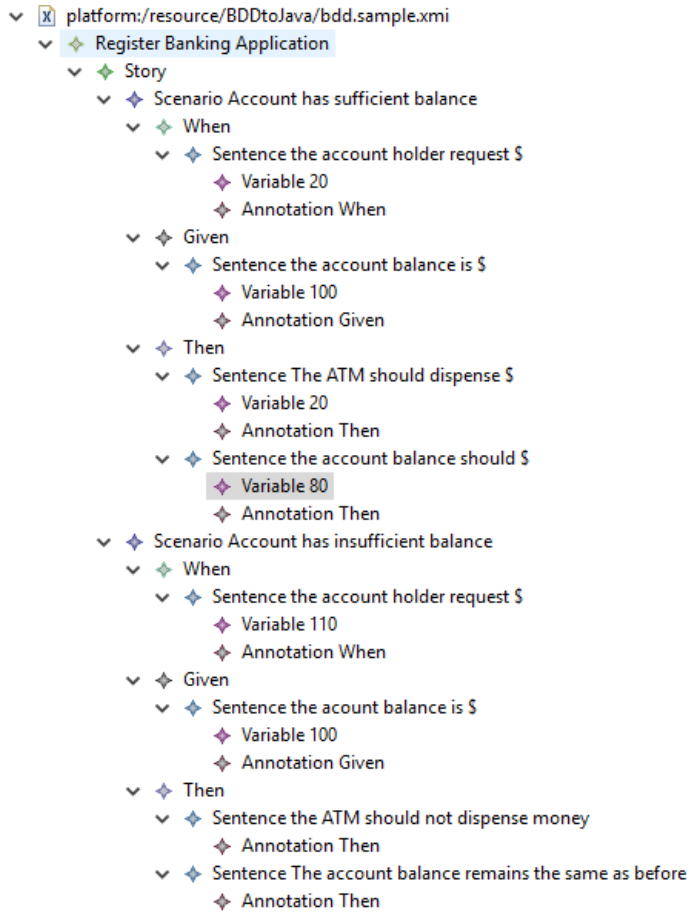

Figure 10 – an example of BDD model

Using the ATL transformation provided by Eclipse, we get the target Java model as the result, as presented in Figure 11. Here, we can verify that the transformation works as we wish. The Java package is named after the BDD's register. The Java class is derived from user's story. Each sentence from the story is converted to a method. The variable is mapped onto method's parameter and BDD annotation is transformed into a java annotation, which can be seen inside

a method's hierarchy. Overall, the resulted Java model provides sufficient information to be transformed into Java source code as shown in Figure 1. At this moment, we believe that the current target model already provide sufficient information to do the code generation. The template language used for model-to-text transformation will provide the details in the generated source code, such as the necessary imported classes used by `jBehave`, syntax of the classes, the methods together with the annotations, and so on

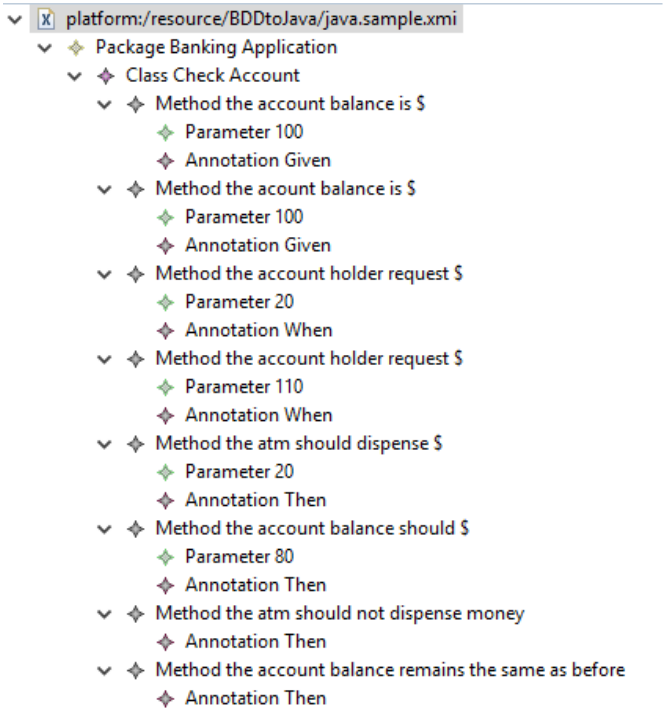

Figure 11 – the resulted Java code model

Other examples of our model transformation are presented in Figure 12 and 13. The first story checks the behavior of the application if the user is able to log in, and when the site visitor is able to register. The next story is about the behavior of enabling/disabling bug tracking in an application. The explanation of the transformation results is similar to the example given above.

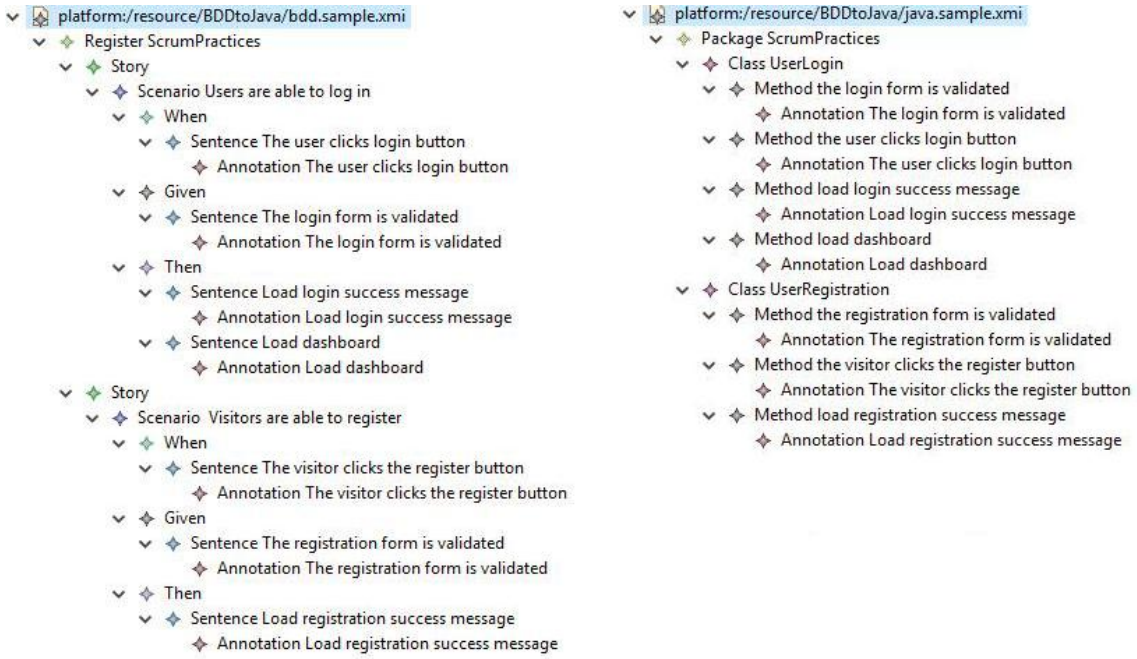

Figure 12 - Model transformation of story of login and registration behavior

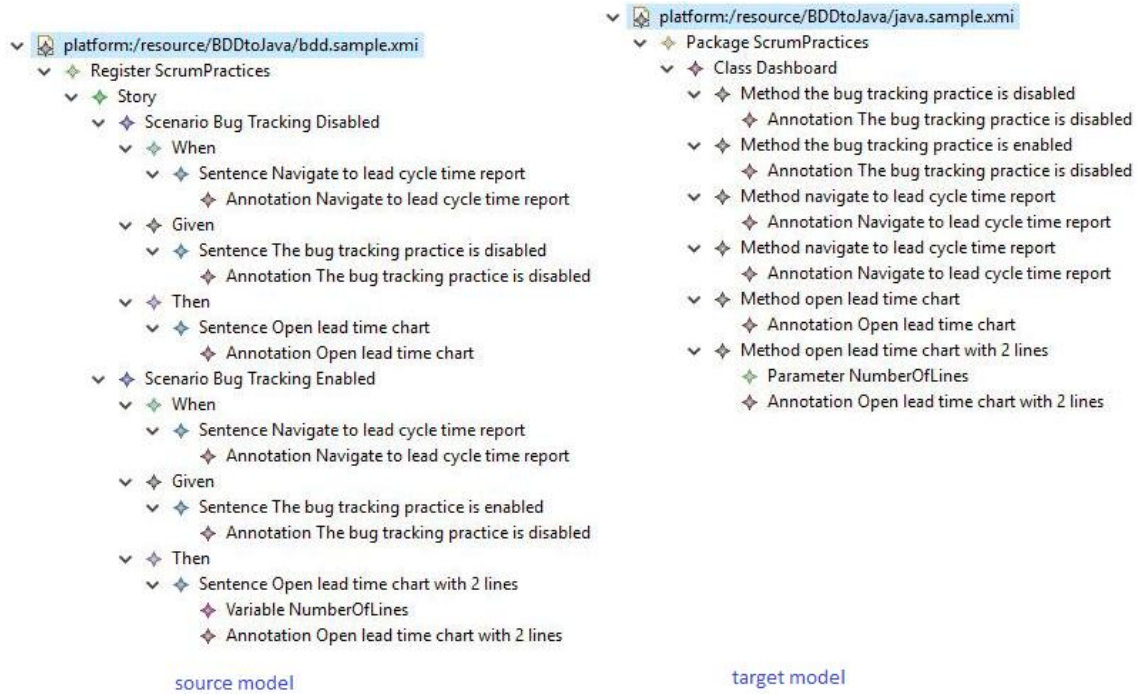

Figure 13 – Model transformation of story of bug tracking behavior

Possible limitations

While we have demonstrated that all generated Java model provides sufficient information to generate the code, there are some identified limitations over our approach.

1. We define a story as a Java class, and put all sentences from all scenarios of that story as methods in the Java class. Therefore, it is possible that stories have similar sentence, hence resulting in similar methods as well. For example, in Figure 11, we can see that there are two methods 'the account balance is', which come from different scenarios. The only difference between those two methods is the value of the parameter.
2. Since our transformation dictates that all sentences contained in scenarios are translated into a single Java class (which is derived from the story), there is no way to find out which sentence is belong to which scenario. We do not consider this limitation into our transformation because the objective is to produce a code that satisfy the way `jBehave` works. Moreover, another reason is that there is no requirement mentioned related to reverse engineering from java code to BDD model.
3. We define the Java method's name as the lowercase of the sentence (thanks to `getMethodName` helper). However, the result is a quite long method name, and sometimes does not feel meaningful. We are still looking for a way to generate method name that is short enough yet meaningful.

Conclusion

So far, we have discussed model transformation from BDD model into Java model, and touch a little bit about the expected generated source code in the next part of this project. In order to accomplish this, we have improved our metamodel from the previous task. We define several ATL rules to do the transformation, and as can be seen from the section Example, our model transformation is able to generate the Java model with sufficient information to be transformed into Java code in the next part of this assignment. We also identified some limitations over our transformation approach.