Project Task 1

Metamodeling of Behavior-Driven Development Support for Software Development

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

Table of Contents

Introduction

Behavior-driven development (BDD) is a software development process which uses stories from users as the base of the development. A story is a descriptive narration explaining about the desired behavior of the software being developed for certain scenarios. BDD extends test-driven development (TDD) method by writing test case in natural language instead of programming language. The purpose of BDD is to minimize communication barrier between technical people, such as developers and testers, and the customer, by providing a common vocabulary between them. In this way, miscommunication between customer and developer can be minimized, because customer are able to be more expressive on describing their idea in the form of stories, and developer can immediately translate those stories into code.

This assignment is intended to develop tools to support BDD software development. From user's stories, the tools will generate relevant source code in a certain programming language. As the first part of the project, in this report we present the metamodel of BDD and the software structures to which a story should be mapped. Finally, an examples of how to use the model is also presented.

Modeling Decisions

BDD

To get insights about how BDD works, we take inspiration from existing BDD tools. There are many BDD tools available for various programming language. To name a few, there are `Cucumber` for Ruby, `Behat` and `Kahlan` for PHP, `Behave` for Python, `Jasmine` for JavaScript, `Concordion` and `jBehave` for Java. Since the primary development tool used in this project is Eclipse, it is natural to select Java as the target language from user's story to make the development process simpler.

The next step is to select the Java-based BDD tools to be modeled. `Concordion` requires user to write the story in HTML, while `jBehave` uses descriptive story in a certain format, which is the one that we would like to develop. Therefore, `jBehave` is preferred over `Concordion`.

As mentioned before, `jBehave` uses a descriptive story from user. The story consists of scenario(s) which follows a certain rule. Figure 1 shows an example of story used in `jBehave`, taken from the official website[1]. The story is about a stock trading system, which describe on what occasion should the trader get an alert for his stock price.

[1] http:// http://jbehave.org/

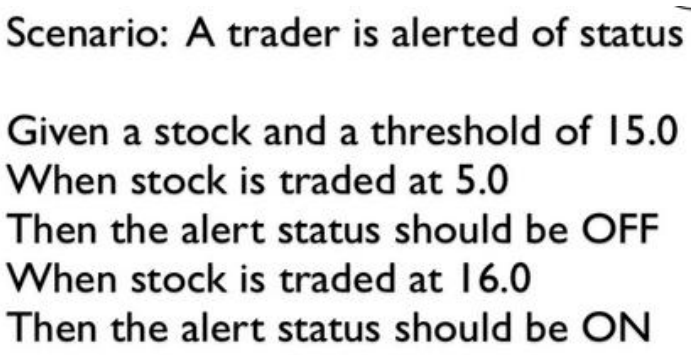

Figure 1 - Example of story with a scenario in it

The initial status is that there is a stock, and the threshold is set to 15.0. There are two different circumstances where the stock is traded at two different values, and the expected outcome for the respective situation.

From the example, we can draw conclusions and make generalization. Firstly, a scenario may consist of one or more scenarios. Secondly, a scenario is composed of three directives:

- `given` (the initial state),
- `when` (certain condition of the scenario),
- `then` (the expected outcome)

Each scenario also has `title` as an identifier. Each `given`, `when`, and `then` directives is followed by one or more description, which we refer it as `sentence` (Figure 2).

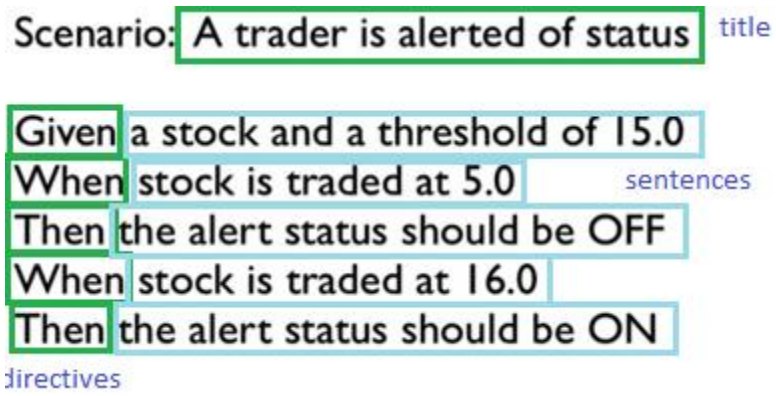

Figure 2 – Composition of a scenario

From the example before, each directive only has a single `sentence`. However, it can be generalized to put several sentences in a single statement, by using `and` operator as conjunction. For example, for the first `When` statement, we can describe it as "*When stock is traded at 5.0 and stock is available*", which consists of two sentences: 1. '*when stock is traded at 5.0*', and 2. '*stock is available'*.

The description above is important on our metamodeling design.

Simplified Programming Language: Java

Java is preferred as the target programming language, because of the object orientation support. We studied Modisco's `java.ecore` file for inspiration to model simplified version of Java programming language. Moreover we closely studied how `jBehave` converts scenarios to methods via annotated descriptions and find Java is a good choice to model.

Metamodel of BDD and Simplified Java

BDD Metamodel

Based on the description above, we develop the metamodel of BDD as can be seen in Figure 2. Firstly, we define `Story` as the topmost class. It contains several attributes for description purposes: `Role`, `Feature`, and `Benefit`. `Role` is used to provide the narrative of the story, `Feature` is used for the functionalities description, and `Benefit` is for the purpose of the story.

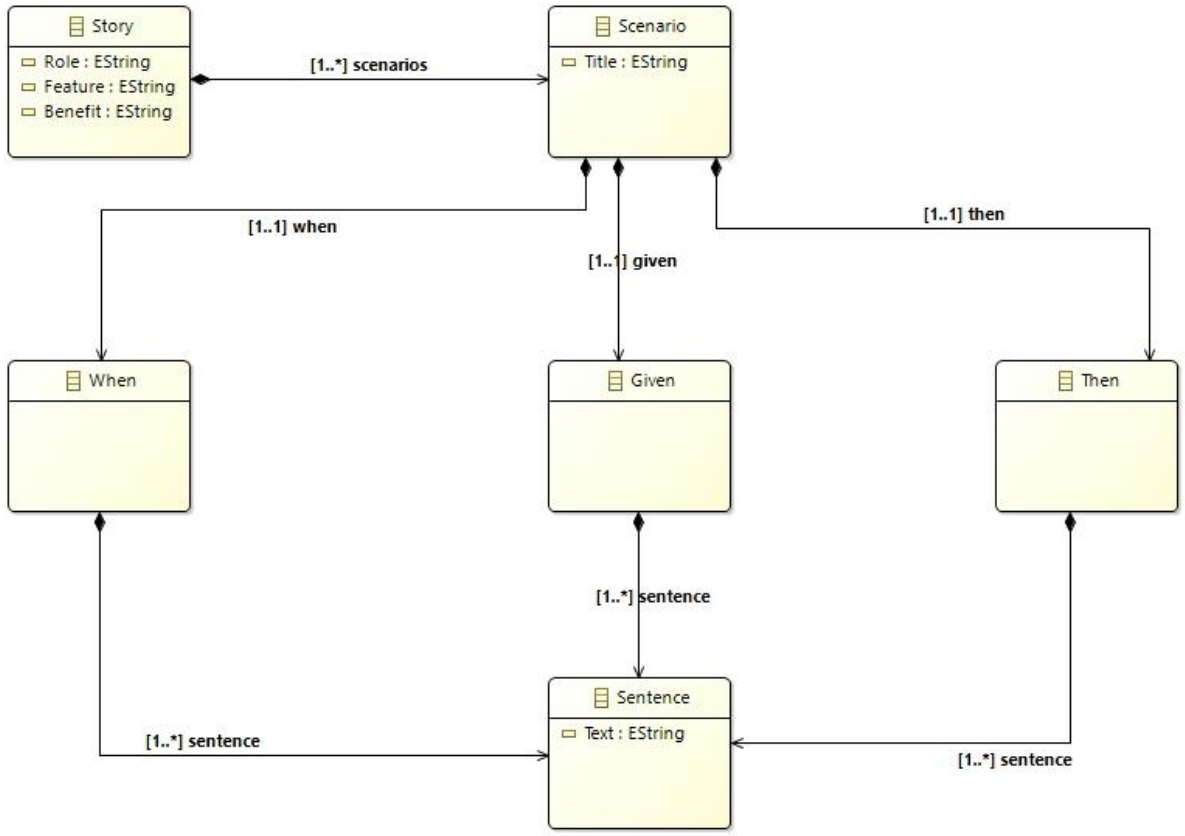

Figure 3 Metamodel of BDD

A `Story` consists of one or more `Scenario`. This relationship is symbolized using component arrow in the diagram. `Scenario` only has single attribute, '`Title`', which is used as an identifier. Scenario consists of one for each of three types of directives: `When`, `Given`, and `Then`. Each of them contains one or more sentences. Here, the usage of operator `and` to merge sentences are not included into the model. The functionality of the operator to split sentences is implemented in the tool which translate the story into source code.

Simplified Java Metamodel

As Java is preferred to be the target programming language, we tried to model all the basic important parts (Figure 4). We define two top level elements named `Package` and `Object`. A `Package` class is created to model actual package in Java, which contains all the classes. The packages gives project a structure and it is easy to separate different concerns per package and class can only contains in a package.

Another top level class `Object` is defined which is similar to object class in Java, from which all the rest of the objects will be derived. It acts as the base class for every other class we create. Two classes are derived from `Object` abstract class named `Type` and `TypedElement`.

We use `Type` to cover primitive data types provided by the language. We defined this class to model those elements which can only be contains in a class. It acts as a parent class for `Member` from which we derived `Function` and `Variable` to model methods and user-defined variables respectively. From `TypedElement` class, which we defined as a base class to model actual classes and interfaces so that user can specify his own classes and interfaces, we derived `Class` and `Interface`. A `Class` can contains `PrimitiveTypeVariable`, `Function` and `Variable`. `PrimitiveTypeVariable` and `Variable` contain an attribute `isParameter` of Boolean type, which we use in mapping parameters from story. A `Package` can contains only `TypeElement`, which means that a `Package` can contains only classes and interfaces for the sake of simplicity. Two enumeration lists are defined namely, `PrimitiveTypes` and `Visibility` to cover basic primitive types and access modifiers. Each `Package` and classes derived from `Object` contain attribute named `Name` which represent the name of the object. We keep it simple to make it easy to test and iterate to add new functionality as required in later stages of the project.

Alternatives

During the metamodeling design, we encountered two alternatives from approaches above.

Alternative 1: Putting sentence as an attribute of directive classes

Initially, we modeled the sentence as attribute of `When`, Given, and `Then` classes. However, this approach has its shortcoming where each sentence itself contains structure, which would be easier to describe as a dedicated classes. Another reason is because `When`, `Given`, and `Then` classes may have many sentences in it. Therefore, it is more appropriate to define sentence as a dedicated class, as our metamodel suggests. In this way, it would be easier to translate the structure of the story into source code.

Alternative 2: Defining 'and' operator as child class of 'Sentence'

We had an option to make operator `and` as child of the sentence class. However, we found out later that in the generated plugin, the placement of child nodes are arranged alphabetically by Eclipse. This nature makes 'and' operator to not be put between sentences, hence making it difficult to parse each sentence later. Therefore, the separation of sentences in each directive through the use of operator 'and' is better to be implemented in application logic during translation into source code.

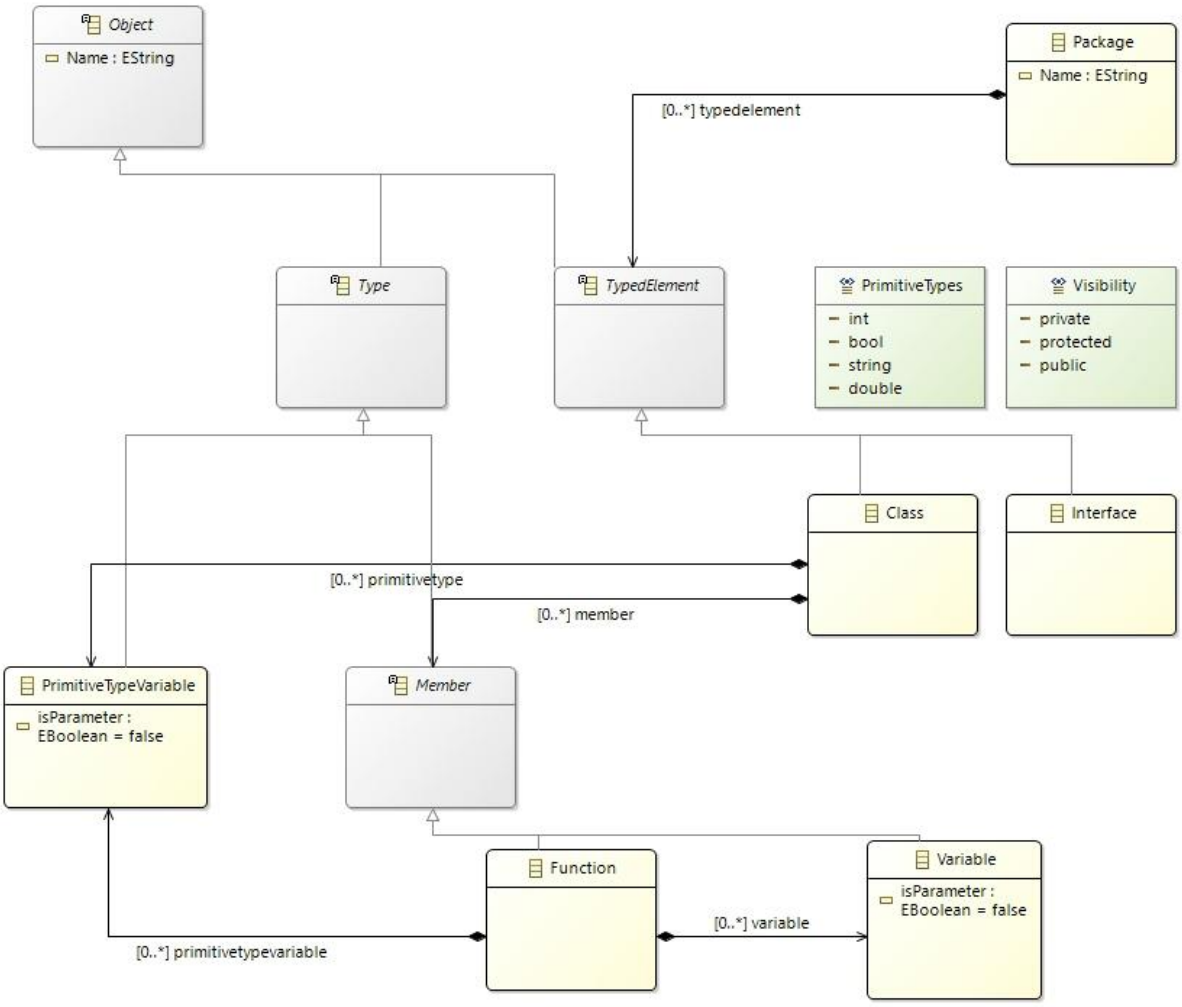

Figure 4 – Metamodel of simplified Java

Examples

To validate our metamodel, we present several examples. Up to this point, there is no relationship between BDD and simplified Java metamodel. Therefore, the examples for both metamodel are presented separately. All example files are submitted together with the report.

Examples of BDD Metamodel

Story: Stock Trading Alert (`BDD-example1.sample`)

To start with, we use story from `jBehave` website as presented in Figure 1. We have discussed the intention of the story in the previous section. It can be seen that the story consists of two scenarios: (i) when stock is traded at 5.0, and (ii) when it is traded at 16.0. The story elements being fitted into our model is depicted in Figure 5. It shows that our model represents a story as we defined before.

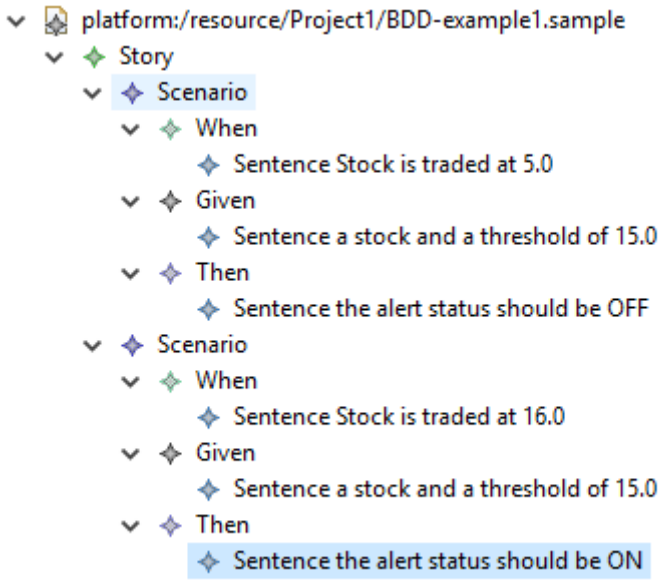

Figure 5 – Model for model of stock trading alert (`BDD-example1.sample`)

Story: Account Holder Withdraws Cash (`Story2.sample`)

The next example is more complicated. Here, an account holder is willing to withdraw cash from an ATM. The scenarios are as follows:

Scenario 1: Account has sufficient funds	**Scenario 2: Account has insufficient funds**
Given the account balance is $100 And the card is valid And the machine contains enough money When the Account Holder requests $20 Then the ATM should dispense $20 And the account balance should be $80 And the card should be returned	Given the account balance is $10 And the card is valid And the machine contains enough money When the Account Holder requests $20 Then the ATM should not dispense any money And the ATM should say there are insufficient funds And the account balance should be $20 And the card should be returned

The model for the above mentioned story is depicted in Figure 6. This story consists of two scenarios (i) when the account has sufficient funds and (ii) when account has insufficient funds. This example demonstrates the ability of our model to represent multiple sentences for the directives.

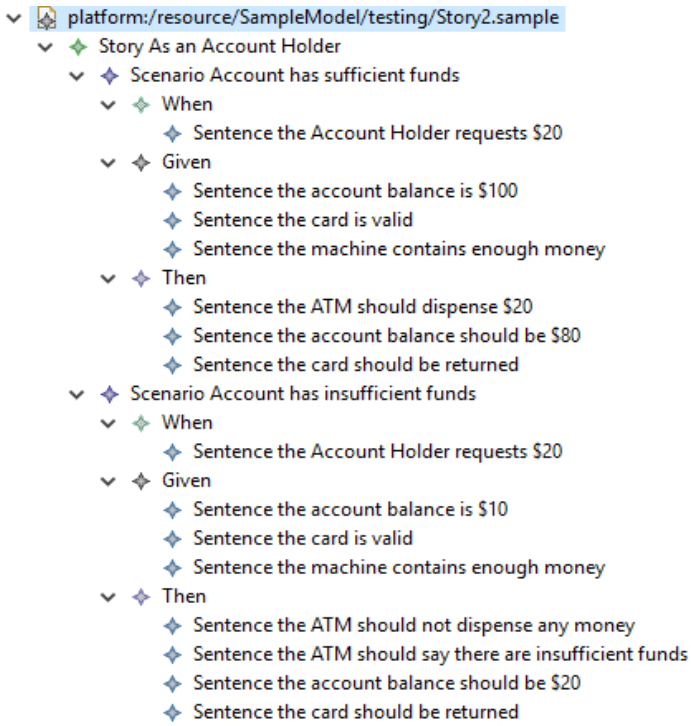

Figure 6 – Model of story of 'account holder withdraws cash' (`Story2.sample`)

Examples of Simplified Java Metamodel

Simple Program (`SimplifiedJava-example1.psample`)

Figure 7 depicts the example model generated from simplified Java programming metamodel, A Package named `mde.project1.example1` containing two classes `HelloWorld` and `Helper` with its own functions/methods and variables.

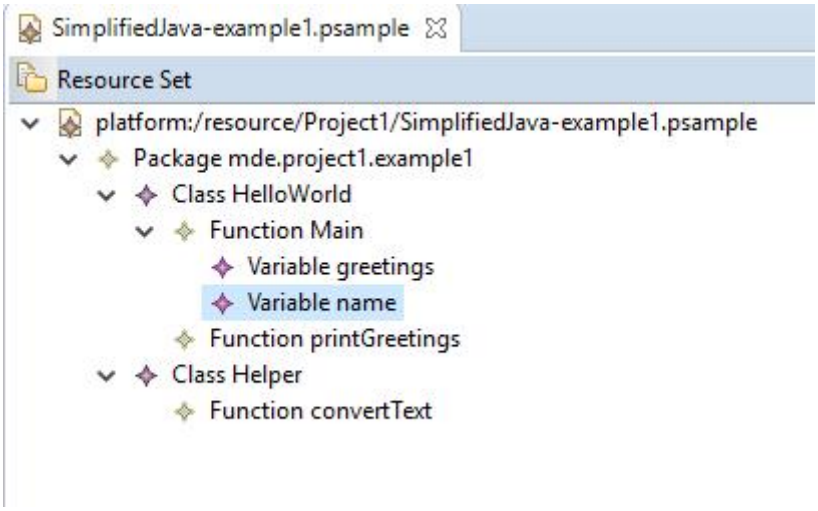

Figure 7 – Model of simplified Java program (`SimplifiedJava-example1.psample`)

A Program with Multiple Functions (`P2.psample`)

Figure 8 shows another model example generated from Simplified Java metamodel. A package named ATM consists of a class Account contains 7 functions which map to the ATM story described above. This example shows that we need to define mappings in concrete syntax to map stories to classes and methods.

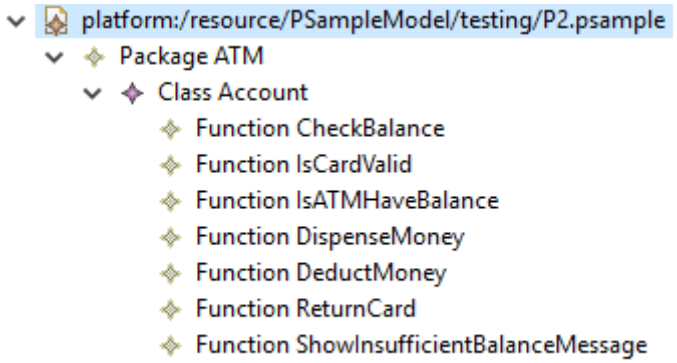

Figure 8 – Model of a simplified Java program with multiple functions (`P2.psample`)

Conclusion

We have developed the metamodel of BDD and simplified Java as the target language. We also demonstrated the validation of our metamodel by using a sample story. It can be seen that our metamodel is sufficient at this point to represent the essence of BDD, which we derive from `jBehave`, and also for the simplified Java program.

The next step is to transform models of BDD stories to models of simplified Java language. Along the way, there is possibility to refine our current metamodels, of which is relatively simple but we consider to be sufficient at this stage, to make the transformation working properly.