9. Exercises on Abstract Syntax of DSLs

Objectives

- To perform a simple scope analysis by studying existing source code
- To design a simple DSL using meta-modeling
- To iteratively refine the meta-model through instance creation

I estimate that this exercise can be completed within 6 hours. Please read the entire description before starting to work on individual tasks.

We are going to design a simple domain specific language that (in later exercises) will be used to generate the assertion class of Junit (`org.junit.Assert.java`). This exercise attempts to replicate in a small scale a realistic re-engineering scenario: where you have a lot of repetitive code, and would like to refactor this code into one or more abstract models, which are much smaller, and a code generator, that can recreate the code. The outcome should be that the system implementation is more concise and there will be a smaller chance of introducing errors (especially if you wrote a good set of meta-model constraints and checked them).

Obviously, because we have to be able to complete the exercise in rather limited time, the whole task is somewhat artificial. The assertion class is small, and clearly not worth of replacing with generated code. In reality however, you meet large systems for which raising the abstraction level obtains huge savings.

Task 1. Find your check out of the Junit code base from the previous exercises. The Assert class is found in `src/main/java/org.junit/Assert.java`. Open the file and read through it trying to understand the main contents. This is the only JUnit file that we will work with in the remainder of the exercises.

In the upcoming exercises, we will be generating (almost) all methods that start with the "assert" prefix. Each of assert methods comes in two versions: with and without a message. For example the simplest assertion method is available in these two versions:

```
static public void assertTrue(String message, boolean condition)
```

```
static public void assertTrue(boolean condition)
```

Observe that once you know how to implement the first of these methods, the second one is trivial: it just calls the first one with a `null` message. So in the code generation scenario, if we have a sufficient specification for the first one, we can generate the second one automatically, based on this input. Today, we are designing a DSL in which we will be writing

the specifications for methods. Please ignore the assertion methods *without* messages, since we will be able to recover all we need about them from the models of methods with messages (so these are the only ones we need to model).

Note that all of these methods have a similar structure: they have 1–3 parameters, and their body is essentially a call to a fail method depending on a simple expression over these parameters. Try to write down (on paper) this expression for each of the assert methods. This will give as a scope of what kind of expressions need to be supported in our DSL.

For example for

```
assertEquals(String msg, double expected, double actual, double delta)
```

You would write the following expression:

```
Math.abs(expected - actual) <= delta
```

(compare this expression with the implementation of `assertEquals` in `Assert.java`).

Repeat this for all the public assert methods (ignore `assertThat` for simplicity). After the task is completed you will have a number of expressions written down on paper, or in a file.

Task 2.* We aim at a language that looks similar to this:

```
Equals (double expected, double actual, double delta)
              asserts Math.abs(expected - actual) <= delta
```

Create a meta-model (in ecore) representing the abstract syntax for this language.

Observe that we use a certain process pattern here: it is useful to create some informal model instances (like the above expressions) *before* you will create the language for describing them; even before you create the meta-model. It is usually much easier to sketch the examples, than to start abstract meta-modeling right away. Once you sketched the examples, working on the meta-model (and later on the grammar), becomes easier.

As an inspiration, my meta-model had approximately the following non-abstract classes: Assertion, Parameter, SimpleType, Expression, Identifier, Binary expression, unary expression, function call, Constant, Null. In total a dozen of classes and one enumeration type. It took me about 30 minutes to create the first version and move to instance creation (to Task 3).

In your ecore model there should be a root class, say `Model`, that owns all the assertions in a model. Use containment (aggregation, black diamond) to bind elements to their parents in the syntax tree. Xtext, which we will use for code generation next week, requires that all model elements are owned directly or indirectly by the root model element (`Model` class here) via containment. A non-contained meta-class (just accessible via references) cannot be instantiated in the standard ecore tree editor.

Task 3.* Create at least three instances of your meta-model representing the assert methods in the Assert.java file (it can actually be one instance with specifications for three assertion functions). Use the expressions you noted down in Task 1 to help you. Whenever you cannot express some instance the way you would like, please refine the meta-model.

Try to create some new assertion methods that do not existing in `Assert.java`, but still make sense. Try to model one as an instance of your meta-model.

Your meta-model is probably not final yet. We will test it and improve it further, when working on code generation in the upcoming exercise sessions.

Hand In: Screenshot of your meta-model and the syntax trees of your instances, showing abstract syntax trees of expressions representing the individual assertion functions.

Scoping notes. Some notes that I took, during the exercise myself. You may find them useful to scope the task down to a manageable size:

- Make sure that you did not miss the list of potential class names in the description of Task 2.
- JUnit has a kind of *internal* DSL for specifying assertions (implemented as the matchers API). It is used by `assertThat`. Let's ignore this fact for the purpose of our exercise, unless someone wants to reuse the design of matchers in her DSL. I also ignored the `assertThat` function in the meta-model design. I will not be generating it.
- The DSL does not need to know about the distinction between message and non-message assertions. The code generator later on will be able to generate both from the same model instance. Similarly, let us ignore all methods that involve arrays (we can later reintroduce them in the code generator).
- To avoid blowing up the number of meta-classes I modeled function calls using Strings (for function names). So I do not have a class for `Math.abs` in my meta-model, but `Math.abs` appears as a string in an instance of a function call meta-class. I did the same for binary and unary operators. If you want to model the operators kinds explicitly, I noted that down that these are the types used in the Assert.java file: `==, &&, !=, equals, <=, -`
- I assumed that the file header (imports, boilerplate), constructor, fail functions, format functions, and some array equality checking classes will be part of the static code included by code generator. They will not be described in the DSL.
- Ignore deprecated methods.
- Finally, a large part of the meta-model is actually a simple expression language. This could have now been borrowed from the XBase language for free, but I decided to model from scratch, to make the experience simpler.