*19*

## DSLs as Programmer Utility

This chapter was written by Birgit Engelmann and Sebastian Benz. You can reach them via **birgit.engelmann@gmail.com** and **sebastian.benz@gmail.com**.

*In this chapter we look at an example of a DSL that is used as a developer utility. This means that the DSL is not necessarily used to implement the core application itself, but plays an important role as part of the development process. The example discussed in this chapter is Jnario, an Xtextbased DSL for behavior-driven development and unit testing.*

Tests play an important role in software development1: they

ensure that an application behaves as expected. However, the benefit of tests goes beyond checking whether a system’s output corresponds to the input. They provide valuable feedback about the quality your application’s design. For example, if a class is hard to test, this can be an indication of too many dependencies or responsibilities, which in turn hinders future maintainability. Furthermore, tests are effective means for developers to elaborate, together with the stakeholders, how the desired features should work. If done right, tests created in such a way can be used as executable specifications of the application’s behavior.

In particular, when tests are used in this latter way, using a general-purpose programming language for writing tests is not

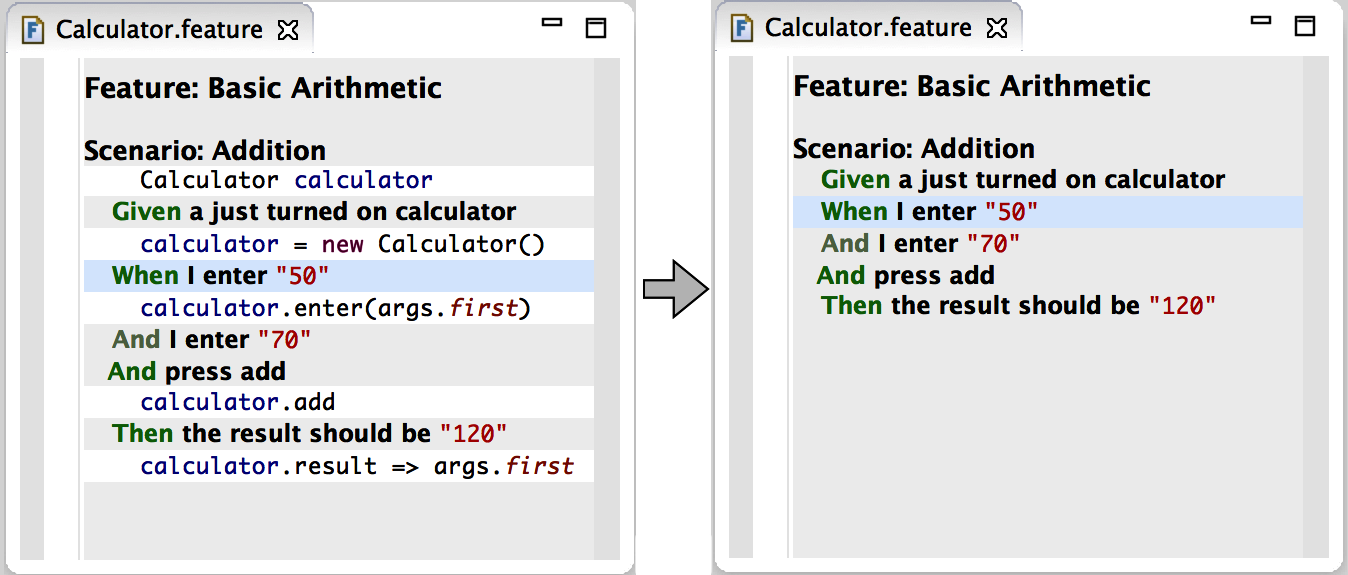
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| very suitable, because it is hard for non-developers to read, un- | | |  | |
| derstand and perhaps even write the specified behavior2. It is | | |  | |
| desirable to describe an application’s behavior using a textual format that can be understood by business users, developers and testers. At the same time, it should be possible to make these specifications executable in order to use them to check automatically whether the application fulfills its expected behavior.  *19.1 The Context*  The core idea of Jnario is to describe features of your application using scenarios. For each scenario, preconditions, events and expected outcomes are described textually with *Given*, *When* and *Then* steps. Here is an example acceptance specification for | | |  | |
| adding values with a calculator. It is written using Cucumber3, a popular Java-based tool for behavior-driven development. | | |  | |
| **Feature**: Basic Arithmetic  **Scenario**: Addition  **Given** a just turned on calculator  **When** I enter "50"  **And** I enter "70"  **And** press add  **Then** the result should be "120" |
| This format was introduced six years ago by Dan North4, and since then has been widely adopted by practitioners. There are tools for most programming languages to turn these scenarios into executable specifications. This is accomplished by mapping each step to corresponding execution logic. For example, in Java/Cucumber you need to declare a separate class with a method for each step: | | |  |
| **public class** CalculatorStepdefs { **private** Calculator calc;  @Given("^a calculator$") **public void** a\_calculator() { calc = **new** Calculator();  }  ...  } |

The method’s annotation contains a regular expression matching the text in the **Given** part of a scenario. The method’s body contains the actual implementation of the step. The tool then executes a scenario by instantiating the class and executing each step with the associated method. The problem with this approach is that the overhead of adding new steps is quite high, since for every new step, a new implementing method has to be created as well.

### 19.2 Jnario Described

|  |  |  |  |
| --- | --- | --- | --- |
| To make it simpler to define executable scenarios for Java appli- | | |  |
| cations, we decided to create Jnario5. In Jnario you can directly add the necessary code below your steps. In our example we create a calculator, pass in the parameters defined in the steps (via the implicit variable *args*) and check the calculated result: | | |  |
| ...  **Feature**: Basic Arithmetic  **Scenario**: Addition  Calculator calculator  **Given** a just turned on calculator calculator = new Calculator()  **When** I enter "50"  calculator.enter(args.first)  **And** I enter "70"  **And** press add calculator.add  **Then** the result should be "120" result => args.first |

This reduces the effort of writing scenarios by not needing to maintain separate step definitions. It is still possible to reuse existing step definitions in scenarios. The editor even provides code completion showing all existing steps. In our example, the step **And I enter "70"** reuses the code of the step **Given I enter "50"** with a different parameter value. A step is resolved by removing all keywords and parameter values from its descriptions and then matching the normalized description to the description of steps with an implementation.

.

You might think now that mixing code and text in your specs makes everything pretty unreadable. Actually, this is not a problem, as you can hide the code in the editor to improve the

|  |  |
| --- | --- |
| readability, as shown in Fig. 19.1. This is accomplished using |  |
| the code folding feature of Eclipse for hiding the step implementations. When editing a step in an editor, its implementation will automatically be shown6. |  |
| Feature definitions in Jnario compile to plain Java JUnit tests, which can be directly executed from within Eclipse. Figure 19.2 shows the execution result of the Calculator feature in Eclipse. |  |

Scenarios are good for writing high-level acceptance specifications, but writing scenarios for data structures or algorithms quickly becomes tedious. This is why Jnario provides another language for writing unit specifications. This languages removes all the boilerplate from normal JUnit tests, helping you to focus on what is important: specifying facts about your implementation. A fact can be as simple as a single expression asserting a simple behavior:

|  |
| --- |
| **package** demo **import** java.util.Stack  **describe** Stack{  **fact new** Stack().size => 0 } |

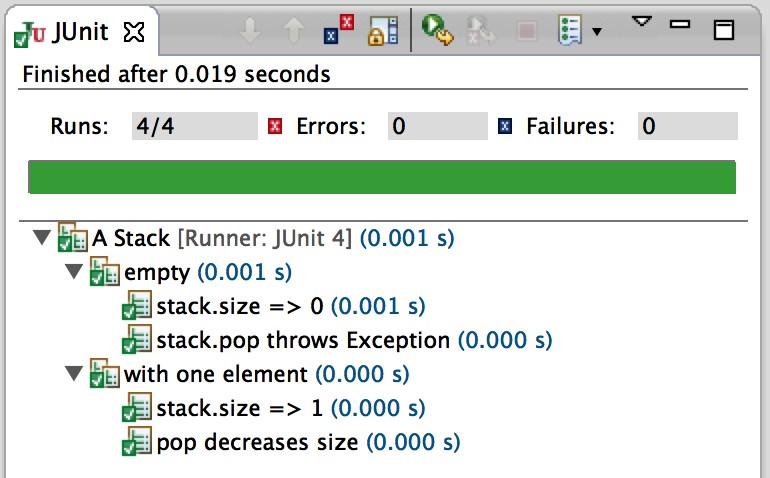
We use **=>** to describe the expected result of an expression. More complex facts can have an additional description:

|  |
| --- |
| **describe** Stack{  **fact** "size increases when adding elements" {  **val** stack = **new** Stack<String> stack.add("something") stack.size => 1  }  } |

Objects can behave differently in different contexts. For example, calling **pop** on a non-empty stack decreases its size. However, if the stack is empty, **pop** results in an exception. In Jnario we can explicitly specify such contexts using the **context** keyword:

|  |
| --- |
| **describe** "A Stack" { **val** stack = **new** Stack<String> **context** "empty" {  **fact** stack.size => 0  **fact** stack.pop **throws** Exception  } **context** "with one element" { **before** stack.add("element") **fact** stack.size => 1 **fact** "pop decreases size" { stack.pop stack.size => 0  }  }  } |

You can execute unit specifications as normal JUnit tests. Note that Jnario uses the description as test name or, if not present, the actual expression being executed. If you look at the executed tests in Fig. 19.3, you can see that your specifications effectively document the behavior of your code.



Jnario is not just another testing framework. It is actually two domain-specific languages specifically made for writing executable specifications. The main advantage of using a DSL for writing tests is that it becomes possible to adapt the syntax to the skills of the intended users. In our case this means that specifications written in Jnario can be understood by users without a programming background.

Another advantage of using a DSL for writing tests is the possibility of adding features that are not available in a general purpose programming language, but are really helpful when writing tests. If you think about current testing frameworks, they usually "stretch" the syntax of the underlying programming language to be able to write expressive tests. Compare that to a programming language in which the syntax is specifically designed for the purpose of writing tests. For example, a common scenario is to test a class with different sets of input parameters. Writing such tests in Jnario is really easy, as it has a special table syntax:

|  |
| --- |
| **describe** "Adding numbers" { **def** additions {  | a | b | sum | | 1 | 2 | 3 |  | 4 | 5 | 9 |  | 10 | 11 | 20 |  | 21 | 21 | 42 |  }  **fact** additions.forEach[a + b => sum] } |

Tables in Jnario are type safe: the type of a column will be automatically inferred to the common supertype of all cells in a column. You can easily iterate over each row in a table and write assertions by accessing the column values as variables. If you execute the example specification, it will fail with the following error:

|  |
| --- |
| java.lang.AssertionError: examples failed  | a | b | sum | | 1 | 2 | 3 |  | 4 | 5 | 9 |  | 10 | 11 | 20 | X (1)  | 21 | 21 | 42 |  (1) Expected a + b => sum but a + b is 21 a is 10 b is 11 sum is 20 |

This demonstrates another advantage of Jnario: it tries to give you as much information as possible about which assertion failed, and why. A failed assertion prints the values of all evaluated sub-expressions. This means you don’t need to debug your tests any further in order to find the exact reason why an assertion failed.

These are just some examples that demonstrate the advantages of test-centric domain-specific language. Having full control over the syntax of the language and its translation into Java code allows us to add features that are helpful when writing tests, but which would never make sense in a general purpose language[[1]](#footnote-1).

### 19.3 Implementation

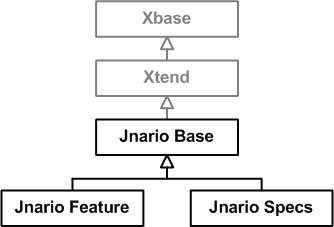
Both Jnario languages – the Feature and Spec languages – are developed with Xtext. Xtext is used for a number of reasons.

*Eclipse Integration* Jnario targets Java developers, which makes Eclipse the best tooling choice, since it is currently the mostly used Java IDE.

*Standalone Support* Although Xtext provides tight Eclipse integration, all language features, such as parser and compiler, are not restricted to Eclipse. This is important, as it should be possible to compile Jnario specifications with maven or from the command line.

*Resuable Expression Language* Implementing a DSL with a custom expression language requires a lot of effort. Xtext provides Xbase, a statically-typed expression language based on Java, which can be integrated into DSLs with relatively little effort. This eliminates the need to implement a custom expression language and ensures compatibility with existing Java code8.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | *19.3.1 Language Definition*  Xtext’s Xbase provides a reusable expression language that can be embedded into DSLs. The Xtend language, which also ships |  |   with Xtext, is a general-purpose programming language for the JVM. Xtend enriches Xbase with additional concepts, such as classes, fields and methods. In Jnario we needed similar concepts, which is why we decided to base the Jnario languages on Xtend. Additionally, reusing Xtend had the advantage that we could reuse a lot of the existing tooling for Jnario.  In Jnario we introduced new expressions, for example more |

8 As we mentioned earlier in this book, Xbase supports extension methods, type inference, lambda expressions

expressive assertions, which can be used in both languages. In order to avoid reimplementing these in the feature and the specs language, we created a base language with all common features used by Jnario. The resulting language hierarchy is shown in Fig. 19.4. This example demonstrates that by, carefully modularizing language features, it possible in Xtext to effectively reuse languages together with their tooling in different contexts. Referring back to the design part of the book, these are all examples of language extension.

As we mentioned earlier, Jnario has assertions with improved error messages. An example is the assert statement, which con-

sists of the **assert** keyword followed by an expression evaluating to a Boolean:

assert x != **null**

Adding a new expression to the Xtext base language works by overriding the corresponding grammar rules. In our example, we added the new assertion expression as a primary expression[[2]](#footnote-2):

|  |
| --- |
| XPrimaryExpression returns xbase::XExpression:  XConstructorCall |  XBlockExpression | ...  Assertion; |

The **Assertion** expression itself is defined as a separate rule. Notice how it again embeds concepts from Xbase, namely **XExpression**:

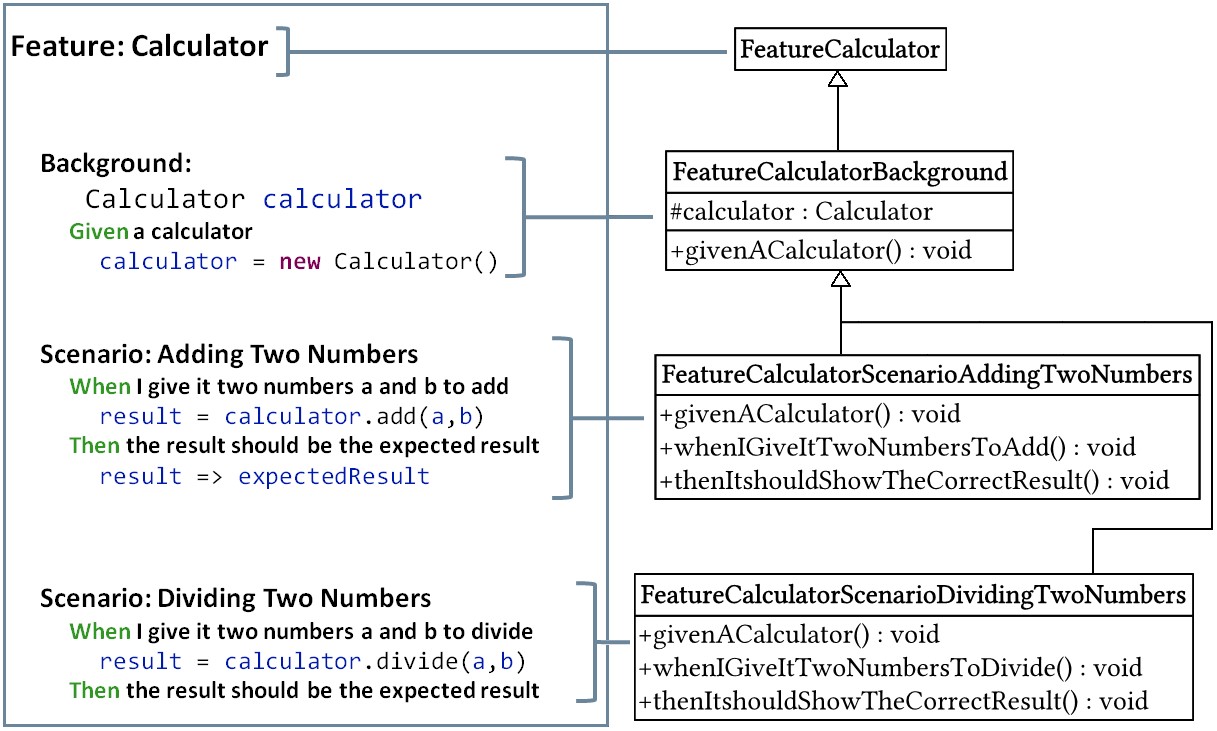
|  |
| --- |
| Assertion returns xbase::XExpression:  ’assert’ expression=XExpression; |

Tables are another example for an extension of Xtend. Defining the grammar for a table in Xtext is pretty straightforward10. A

table has a list of columns and rows. Each column in the cells in a row are separated by ’|’:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | Table:  ’def’ name=ID ’{’  (’|’ (columns+=Column)\* (rows += Row)\*)?  ’}’;  Column: name=ValidID ’|’;  Row:  ’|’ {Row} (cells+=XExpression ’|’)\*; |   However, this is not really a problem since users will naturally use that notation. Also, a formatter could be implemented to format the table correctly. In addition, an editor template could be defined to create an example table that is formatted in a meaningful way.  Cells in the table can contain arbitrary expressions. We reused the typing infrastructure provided by Xtext to calculate the type of each column in the table: a column’s type is the common supertype of all expressions in the respective column. Here is the essential part of the code:  @Inject **extension** ITypeProvider @Inject **extension** TypeConformanceComputer  **def** getType(ExampleColumn column){ **val** cellTypes = column.cells.map[type] return cellTypes.commonSuperType  }   |  |  | | --- | --- | | In the next section we will have a look at how we map these custom concepts to the generated Java classes.  *19.3.2 Code Generation*  Jnario specifications are compiled to plain Java JUnit4 tests. Xtext provides a framework for mapping DSL concepts to corresponding JVM concepts. An example mapping for the feature language is shown in Fig. 19.5. |  | | Scenarios and Backgrounds11 are mapped to Java classes, in |  | |

which each step is mapped to a JUnit test method. The additional **@Test** annotations required by JUnit are added during the mapping process. The expressions can be completely transformed by the existing Xtend compiler. However, in order to



support custom expressions such as **assert**, the Xtend compiler needs to be extended. Consider the following example:

**val** x = "hello" **val** y = "world"

assert x + y == "hello world"

The execution of these statements will result in the following error message:

java.lang.AssertionError: Expected x + y == "hello world" but x + y is "helloworld" x is "hello" y is "world"

|  |  |
| --- | --- |
| Note that the error message contains the original failing expression **x + y == "hello world"** rendered as text, together with the value of all subexpressions. Creating such error messages is not possible in plain Java, as the underlying AST cannot be |  |
| accessed. However, in a DSL like Jnario we can include this information when generating the Java code from our specifications12. To do so, we subclass the XtendCompiler and add a |  |
| custom generator for our assertion expression: |  |

|  |
| --- |
| @Override **public void** doInternalToJavaStatement(XExpression obj,  ITreeAppendable appendable, **boolean** isReferenced) { **if** (obj **instanceof** Assertion) { toJavaStatement((Assertion) obj, appendable, isReferenced);  } **else** {  **super**.doInternalToJavaStatement(obj, appendable, isReferenced);  }  } |

The custom generator translates an assertion into the corresponding Java code:

**private void** toJavaStatement(Assertion assertion, ITreeAppendable b) { XExpression expr = assertion.getExpr(); toJavaStatement(expr, b, **true**);

|  |
| --- |
| b.newLine();  b.append("org.junit.Assert.assertTrue(");  b.append("Expected ");  b.append(serialize(expr));  b.append(" but "); appendSubExpressionValues(expr, b); b.append(", "); toJavaExpression(expr, b); b.append(");");  } |

We first get the actual expression of the assertion, which is then compiled by invoking **toJavaStatement(expr,...)**. The Xtend compiler automatically creates a temporary variable for each subexpression13. The value of these variables will be

used later to generate the error message. The assertion itself is mapped to the JUnit method **Assert.assertTrue(message,**

**result)**. The key to showing the expression in a textual form as part of the error message is that the message is built by serializing the expression as it is written in the Jnario file together with the values of the subexpressions taken from previously created temporary variables. Finally, execution result expression is compiled by invoking **toJavaExpression(expr, ...)**.

#### 19.3.3 Tooling

For Jnario, good tooling is essential, since the hurdles of learning a new language should be as low as possible, and being able to work efficiently is a major goal of Jnario.

The Feature language aims at being readable both for software developers and for domain experts who have no programming background. The best way to support both groups is to provide two views on a Jnario Feature file. The editor uses specialized folding to show or hide the implementation code of the scenarios and features14. Specialized folding means we used the existing folding component from Xtext and extended it to disable the normal folding functionality. We then introduced a button to trigger the folding of the implementation logic. In addition, syntax highlighting comes in handy when referring to existing steps. We used different colors to show whether a step is implemented (i.e. has its own code associated with it), not implemented (i.e. needs to be implemented before the specification can be run), or is a reference to an existing step (in which case the existing step’s implementation code is automatically reused).

To improve productivity further, editor features such as quick fixes for importing necessary libraries or for creating new class

|  |  |
| --- | --- |
|  |  |
| *19.3.4 Testing*  Testing is an important part of developing a DSL. This of course applies to Jnario as well. However, Jnario is a special case, since it itself is a language for writing tests. This gave us the chance |  |
| to bootstrap the implementation of Jnario16. The advantage of |  |
| bootstrapping is that bugs or missing features quickly become apparent just from using the language to build test for the language (we ate our own dog food). Here is an example in which we test Jnario’s **throws** expression: |  |

|  |  |
| --- | --- |
| When we execute this scenario, the given unit specification will |  |
| first be parsed and compiled into Java source code17. The Java |  |

files were added. Auto completion for steps in features is supported, since you can reuse other steps by referencing them, even if they are not implemented. Another important productivity feature is the debug support of Xtext, which lets you step through your DSL file instead of using the generated code15.

|  |
| --- |
| **Scenario**: Testing exceptions with Jnario  **Given** a unit spec  ’’’ describe "expecting exceptions in Jnario"{ fact new Stack().pop throws EmptyStackException }  ’’’  **Then** it should execute successfully |

source code will then be compiled into a Java class file using the normal Java compiler. The generated class is then loaded via the Java classloader and executed with JUnit. This process is greatly simplified by the testing infrastructure provided by Xtext, which provides APIs for compiling and classloading DSL artifacts. The advantage of this approach is that the whole compiler chain is tested, and tests are relatively stable, as they are independent of changes in the internal implementation.

### 19.4 Summary

This chapter introduced Jnario, a DSL for testing, which was developed based on Xtext. Jnario is a good example for a DSL that is targeted towards developers with the goal of easing the development process. It is also a good example of an Xtextbased language that makes use of Xtext’s reusable expression language Xbase. Jnario is currently used in various domains, for example to specify and test the behavior of automotive control units, web applications and Eclipse-based applications.

.

1. ! [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)