NatSpec User Guide

Outline

1 Summary for Busy Managers

The goal of any software development process is to deliver software to customers that satisfies their needs. Transforming requirements into a shipped product is essentially the job that needs to be done. Requirements are often described in natural language. In modern software development, checking whether those requirements are met and implemented correctly is typically done by using unit tests. Such unit tests are usually described in a programming language like Java. The language gap between describing the requirements in natural language and implementing them as unit tests makes it difficult for customers and developers to communicate effectively.

NatSpec bridges this gap. It provides means for the specification of requirements with your customers in natural language while enabling their execution as a unit test at the same time. Thus, NatSpec allows for writing readable, maintainable, and executable requirements and test specifications that are understood by customers and developers. An exemplary NatSpec specification is shown below:

```
Create airplane Boeing-737-600
Set totalSeats to 2
Create flight LH-1234
Set plane to Boeing-737-600
Create passenger John Doe
Book seat for John Doe at LH-1234
Assume success
```

A specification like this is fully sufficient to generate a test case which ensures that the described scenario is implemented correctly by your software. Have you ever dreamed of a solution being that simple?

2 Guide for Busy Requirements Engineers (and Customers)

It is your job to describe the functions a software is meant to deliver. Only if this is done in a very concrete way, developers will be able to realize them as expected. However, it is often hard to find a precise and yet readable way to give a requirements specification. NatSpec helps you to build precise scenario specifications that use natural language. There are no limitations on the structure or terminology. Furthermore, NatSpec specifications can be used by developers to validate the functions they implement against your expectations.

2.1 Creating a Natural Scenario Specification

In NatSpec one can write tests in natural language in form of stories or scenarios. Each scenario is meant to describe a function of the application under development in terms of actions and expected behavior. Typically, a scenario involves creating some input data, invoking a few application services, and checking the actual results against expected results. To use NatSpec you should first create a new NatSpec project: `File > New > Other... > NatSpec Project`. Then add a new test specification to the folder for end-user tests (`src-endusertests`): `File > New > Other... > NatSpec Scenario File (.natspec)`.

2.2 Initializing some Input Data

As NatSpec uses natural language, initializing the input data in a scenario is as easy as writing a plain English sentence. For an exemplary application that deals with airplanes, flights and passengers to book air travels you could write something like:

```
Create airplane Boeing-737-600
Set totalSeats to 2
Create flight LH-1234
Set Airplane to Boeing-737-600
Create passenger John Doe
```

There are no syntactic limitations on the sentence structures. There are also no specific keywords required. You can use plain English or any other language of your choice!

2.3 Using an Application Service

Invoking an application service is equally easy:

```
Book seat for John Doe at LH-1234
```

2.4 Checking Expectations on Output

To check the service results against your expectations you can write something like:

```
Assume LH-1234 has passenger John Doe
```

That's it. Just write down scenarios capturing expected functionality. The test developers will map your text to actual test code later on and make sure that the respective functionality is automatically verified.

3 Guide for Busy Developers

It is your job to implement the functionality that your customers require. However, it is hard to implement your customers' expectations if these are not precisely described or simply incomplete. NatSpec helps you to communicate with your customers in terms of exemplary scenarios. As it uses natural language, those scenarios can be easily understood both by yourself and by your clients. It will even help you to ask your customers for more detailed specifications and to resolve misunderstandings. NatSpec provides means to describe scenario specifications testing the functions you implement. Therefore, scenario specifications are mapped to JUnit tests in an easy and comprehensible way. NatSpec makes requirements engineering more precise. It reduces the effort you invest for building sensible test cases and gives test-driven development a kick start.

3.1 Making Sentences Executable using Test-Support Classes

To make a sentence of a scenario specification executable, it is simply mapped to a Java method. This is done in so-called test-support classes. A test-support class is a simple Java class that contains test-support methods. Test-support methods describe how the syntax of a sentence in the natural specification is mapped to Java. These methods are identified by the annotation `@TextSyntax` . An example is shown in the following:

```
public class TestSupport {
    private AirlineServices services;
    public TestSupport(AirlineServices services) {
        this.services = services;
    }
    @TextSyntax("Book seat for #1 at #2")
    public boolean bookSeat(Passenger passenger, Flight flight) {
        boolean success = services.bookSeat(passenger, flight);
        return success;
    }
}
```

The method `bookSeat` makes the sentence

```
Book seat for John Doe at LH-1234
```

executable. Typically, the application service (e.g., `services.bookSeat(passenger, flight)`) that is tested with a specific scenario is to be parameterized with some input data. Thus, it is necessary to clarify where and how such input data is represented in the natural specification. This is done in the syntax mapping:

```
@TextSyntax("Book seat for #1 at #2")
```

It consists of static structures (words) and variable placeholders. The placeholders are indicated by a hashtag (#) followed by a number. This number refers to the index position of the method parameters for the according test support method. In our example #1 refers to parameter `passenger` and #2 refers to parameter `flight` . Within the test support method the meaning of the sentence is implemented using plain Java. To assign the same test support method to multiple sentences, the `@TextSyntaxes` annotation can be used, which can consist of multiple `@TextSyntax` annotations. For example, the following annotations map two sentences to one test support method.

```
@TextSyntaxes({
   @TextSyntax("Book seat for #1 at #2"),
   @TextSyntax("Book flight for #1 at #2"),
   })
```

NatSpec accepts arbitrary many syntax patterns, but one must take care to make sure that these patterns do not match the same sentences.

3.1.1 Matching Lists of Words

NatSpec supports to assign multiple words to a single parameter. This is automatically done, if the type of the respective parameter is 'java.util.List'. For example:

```
@TextSyntax("Print #1")
public void printNames(List<String> names) {
   for (String name : names) {
      System.out.println(name);
   }
}
```

matches the sentence:

```
Print John Jane Peter
```

To match a list of words, but use them as a single String argument, the annotation `@Many` can be used:

```
@TextSyntax("Print #1")
public void print(@Many String names) {
   System.out.println(names);
}
```

This does also match the sentence:

```
Print John Jane Peter
```

but passes "John Jane Peter" as argument to `print()` .

3.1.2 Matching Boolean Parameters

NatSpec supports boolean parameters, but these require special handling as it is not clear which text in a sentence represents the value 'true' and which represents 'false'.

If one wants to represent 'true' by the presence of one word, the parameter name can be used to specify this word. For example, the following listing:

```
@TextSyntax("Assume #1 failure")
public void assume(boolean no) {
        System.out.println(no ? "Failure not expected." : "Failure
expected.");
}
```

matches the following two sentences:

```
Assume no failure
Assume failure
```

For the first sentence the value 'true' is assigned to the parameter 'no'. For the second sentence that parameter is set to 'false'.

Alternatively, one can explicitly specify the words that are used to represent 'true' and 'false' by using the @BooleanSyntax annotation:

```
@TextSyntax("#1 authentication")
public void selectAuthentication(@BooleanSyntax({"Enable",
"Disable"}) boolean enable) {
System.out.println("Authentication is " + (enable ? "on" : "off"));
}
```

matches the following two sentences:

```
Enable authentication
Disable authentication
```

3.1.3 Defining Optional Implicit Parameters

NatSpec supports optional implicit parameters. Such parameters are optional in the sense that if no matching object in the context of the NatSpec script was found, null is used as argument for the parameter.

```
@TextSyntax("Simulate flight #1")
public void assume(Flight flight, @Optional FlightCrew crew) {
    if (crew != null) {
        flight.setCrew(crew);
    }
    flight.simulate();
}
```

3.1.4 Enforcing Patterns for Text Syntax Patterns

NatSpec supports the enforcing of patterns that have to apply for all syntax patterns in the same test support class. This can be useful to ensure that all syntax patterns from one test support class follow a basic construction rule. Thus, they can easily be identified when used in a NatSpec specification.

To define such construction rules you can use the @AllowedSyntaxPattern annotation. The value of the annotation is a regular expression which all syntax patterns must match. For example, the following annotation defines that only patterns which start with "Assume" can be defined in the respective test support class:

```
@AllowedSyntaxPattern("Assume.*")
public class MySupportClassWithAssumptions {
    @TextSyntax("Assume ticket is valid")
    public void assume(Ticket ticket) {
        assertTrue(ticket.isValid());
        assertEquals(0, ticket.getErrors());
    }
}
```

3.1.5 Supported Primitive Types

NatSpec supports the following primitive types:

- String
- int/Integer
- double/Double
- java.util.Date
- java.math.BigDecimal

3.1.6 Deriving Syntax from Method Names

Alternatively to the annotation `@TextSyntax` , the annotation `@NameBasedSyntax` can be used. The syntax pattern for methods which are annotated with `@NameBasedSyntax` is derived from the method name. For example:

```
@NameBasedSyntax
public void print_and_toStdout(String text1, String text2) {
   System.out.println(text1);
   System.out.println(text2);
}
```

is equivalent to the following method:

```
@TextSyntax("Print #1 and #2 to Stdout")
public void print(String text1, String text2) {
   System.out.println(text1);
   System.out.println(text2);
}
```

The underscore indicate the places where parameters can occur in the sentence and the words are split based on the camel-case method name.

3.1.7 Using NatSpec with JUnit

3.1.8 Creating a NatSpec Template that uses JUnit

To use NatSpec for the specification of JUnit tests, one must supply a NatSpec template (_NatSpecTemplate.java) that includes a method with an @Test annotation. This method must contain the @MethodBody placeholder to tell NatSpec to put the generated code into the method. For example, your template could look like this:

```
import org.junit.Test;
public class _NatSpecTemplate {
   protected MyTestSupport myTestSupport = new MyTestSupport();
   @Test
   public void runTest() throws Exception {
      /* @MethodBody */
   }
}
```

3.1.9 Using the NatSpec JUnit Template to show sentences in JUnit view

NatSpec provides a custom template that enables to show the individual sentences in the Eclipse JUnit view when running test cases. To use this template, the plug-in 'de.devboost.natspec.junit4.runner' must be added to the project dependencies. Once the dependency has been added, a template can be defined as follows:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import de.devboost.natspec.junit4.runner.NatSpecJUnit4Runner;
import de.devboost.natspec.junit4.runner.NatSpecJUnit4Template;
@RunWith(NatSpecJUnit4Runner.class)
public class _NatSpecTemplate extends NatSpecJUnit4Template {
    protected MyTestSupport myTestSupport = new MyTestSupport();
    @Test
    public void runTest() throws Exception {
        /* @MethodBody */
    }
}
```

Note that the template must extend the NatSpecJUnit4Template class and run with the NatSpecJUnit4Runner. Only if both is declared, the Eclipse JUnit view will show the individual sentences as test steps.

3.1.10 Validating Expectations using JUnit

To validate expectations expressed in natural-language specifications you can simply use JUnit methods in test-support methods. To check the expectation:

```
Assume LH-1234 has passenger John Doe
```

you can write the following test support method:

```
@TextSyntax("Assume #1 has passenger #2")
public void hasPassenger(Flight flight, Passenger passenger) {
    Assert.assertTrue(flight.hasPassenger(passenger));
}
```

3.1.11 Running NatSpec-based JUnit Tests

Every NatSpec file that uses a template containing an @Test method can be launched as JUnit test. Just perform a right-click on the NatSpec file and select 'Run As > JUnit Test'.

3.2 Making Sentences Executable using Pattern Providers

For making natural specifications executable, test-support methods define patterns of sentences and their mapping to the Java code given in the method body.

When dealing with large APIs that follow consistent conventions, it is often sensible to create such patterns programmatically. A common example are entity classes as used in persistence layers or for data transfer objects. These often follow the JavaBeans convention for accessing fields and common construction patterns like factories. It would cause lots of effort to manually implement test-support methods for all such entities.

It is much more effective and consistent to define the mapping patterns for their getters, setters and constructors in a standardized way. To do so, you can implement a custom `ISyntaxPatternProvider`. For an example see the `JavaCustomizationProvider` in the project `de.devboost.natspec.java.customization`. It programmatically registers patterns for POJO entities following the JavaBeans conventions.

Important note: Make sure you're implementing the pattern provider in an Eclipse plug-in project (i.e., an OSGi bundle). This is required to allow NatSpec to reload your pattern provider class dynamically without restarting your Eclipse instance. Thus you can make changes to the pattern provider code and instantly see the resulting changes. To get a more thorough understanding of how to implement pattern providers see

3.3 Dealing with Synonyms

Natural languages are full of synonyms. Instead of writing

```
Create airplane Boeing-737-600
```

you may also want to write

```
Create plane Boeing-737-600
```

To support synonyms you just need to create a file called `synonyms.txt` in your test project and add lines of comma-separated synonyms. The following listing gives synonyms for `airplane` , `passenger` , and `flight` .

```
airplanetype,airplane,plane,aircraft
passenger,traveler
flight,travel
```

3.4 Documenting NatSpec Scenarios

Even so we're using natural language, some scenarios or sentences need additional documentation. NatSpec provides means for documenting scenarios on different levels of granularity, ranging from documenting a single sentence to top-level scenario documentation.

3.4.1 Using Comments to Document Sentences

Comments can be used to document sentences in NatSpec scenarios. The character sequence to denote standard comments is two forward slashes followed by the comment text. For an example, see the following specification:

```
Create airplane Boeing-737-600
// We need to set the total seats so that tickets can be issued
Set totalSeats to 2
```

3.4.2 Using Comments to Define Markers in Scenarios

Markers in scenarios are a feature to define sections in scenarios. The character sequence to denote markers is three forward slashes followed by the marker text. The outline view can then be configured to only show those markers instead of entries for all sentences in the scenario.

```
/// Initialization
Create airplane Boeing-737-600
// We need to set the total seats so that tickets can be issued
Set totalSeats to 2
```

3.4.3 Using Comments With Task Tags to Denote TODOs and FIXMEs

Task tags, i.e., TODO and FIXME can also be used in standard comments to denote tasks that need to be listed in the Tasks view.

```
Create airplane Boeing-737-600
// TODO We need to initialize the airplane seats
```

3.5 Preserving the Implicit Contexts within a Specification

In natural language we typically use implicit contexts in a sequence of several sentences. For an example, see the following specification:

```
Create airplane Boeing-737-600
Set totalSeats to 2
```

The second sentence implicitly refers to the airplane created in the previous sentence. NatSpec does also support this kind of context preservation. For the second template you would define a test-support method like:

```
@TextSyntax("Set totalSeats to #2")
public void setTotalSeats(AirplaneType airplane, int seats) {
   airplane.setTotalSeats(seats);
}
```

As you can see, there is no placeholder bound to the first method parameter `airplane` . To determine the value for unbound parameters, NatSpec investigates the sentence context for matching objects. Therefore, it inspects all objects that were a result of the previous sentences in the specification and uses the latest result object with a matching type. In the following example NatSpec would use the airplane Boeing-737-700 as parameter for calling `setTotalSeats()`.

```
Create airplane Boeing-737-600
Create airplane Boeing-737-700
Set totalSeats to 2
```

The result object of a sentence is determined by the value that is returned by the corresponding test-support method. The support method for the first two sentences looks as follows:

```
@TextSyntax("Create airplane #1")
public AirplaneType createAirplaneType(String typeName) {
   return new AirplaneType(typeName);
}
```

3.6 Generating and Running JUnit Test Cases

NatSpec automatically generates JUnit test cases whenever you save a NatSpec specification file. Generation is based on the mappings you define using test-support classes and pattern providers. For every NatSpec file a corresponding Java class having the same name and being located in the same package will be generated. You can execute the test case like any other JUnit test: `Run > Run as > JUnit Test`

3.7 Setting the Target Folder for Generated Classes

By default, NatSpec writes the generated classes to the same folder the NatSpec specification is contained in. This behavior can be adjusted by setting the NatSpec project property 'Output folder'. This property can be set by invoking the context menu for a project and selecting the 'Properties' menu item. The NatSpec property page can then be used to set either a relative or an absolute path (w.r.t. the project root) where to store the generated classes.

3.8 Initializing Infrastructure for JUnit Test Cases

As test cases often require some infrastructure, e.g., to access the database layer or the business services to test, you can influence how test cases are generated using test-case templates. A test-case template is a Java class named `_NatSpecTemplate.java` (Note the underscore!) that you place next to your NatSpec specifications. Test-case templates can be placed in the same folder as the NatSpec file or any parent folder. The first template that is found when ascending the folder hierarchy is used for each NatSpec test.

An exemplary test-case template is shown below:

```
package com.yourdomain;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import
de.devboost.natspec.examples.airline.InMemoryPersistenceContext;
import de.devboost.natspec.examples.airline.services.AirlineServices;
/* @Imports */
public class _NatSpecTemplate {
   private AirlineServices services;
   private TestSupport testSupport;
   private InMemoryPersistenceContext persistenceContext;
   @Test
   public void executeScript() throws Exception {
      /* @MethodBody */
   }
   @Before
   public void setUp() {
      services = new AirlineServices();
      testSupport = new TestSupport(services);
      persistenceContext =
      InMemoryPersistenceContext.getPersistenceContext();
   }
}
```

As the test-case template is a Java class you can use plain Java code to initialize any infrastructure required for executing the test. You can also simply use JUnit annotations like `@Before` or `@After` for set up and tear down activities.

The template also uses two placeholders to indicate a place in the template that will be replaced with generated code:

3.8.1 @MethodBody (required)

The `/* @MethodBody */` placeholder will be replaced with the code generated from a specific NatSpec specification.

3.8.2 @Imports (optional)

The `/* @Imports */` placeholder will be replaced with imports, which are generated from a specific NatSpec specification. If not provided, imports will be generated before the imports which are already present.

4 NatSpec API Guide

NatSpec provides an API to allow expert users to programmatically create syntax patterns without the need for `@TextSyntax` annotations. Also, providers for synonyms can be registered with this API. The following sections cover details on the most important parts of this API.

4.1 Pattern Provider API

To extend NatSpec dynamically with syntax patterns one can create a class that implements the interface `ISyntaxPatternProvider` . This class must reside in an OSGi bundle located in the current workspace. Whenever a change is applied to the class, NatSpec will automatically reload the OSGi bundle and instantiate a new instance of the class which will be asked to provide syntax pattern when necessary.

To ask for syntax patterns, NatSpec calls the method `getPatterns()` on the provider class which returns a collections of patterns (i.e., instances of `ISyntaxPattern`). NatSpec passes the URI of the current specification file to `getPatterns()` in order to allow pattern providers to return different collection of patterns for different specifications. This can, for example, be used to consider the classpath of the project that contains the specification when computing syntax patterns.

Syntax patterns can be realized by creating a class that implements `ISyntaxPattern` . We strongly encourage clients of the API to extend `AbstractSyntaxPattern` instead of implementing the interface only.

A syntax pattern can match a sentence in a NatSpec specification and consists of syntax pattern parts which can match words in a sentence. The parts of a syntax pattern must be returned by its method `getParts()` . Syntax pattern parts can be simple static words (see class `StaticWord`) or match certain types of parameters (e.g., integers or dates). One can either implement custom syntax pattern parts or use existing ones provided by the NatSpec API. Most common syntax pattern parts are:

- StaticWord - match a single word case-insensitive or one of its synonyms
- IntegerArgument - match a single integer number
- DoubleArgument - match a single floating-point number
- DateArgument - match a single date
- ListParameter - match a list of arguments
- ComplexParameter - match an instance of a specific Java class

Custom syntax pattern parts must implement `ISyntaxPatternPart` and respectively the method `match(List<Word> words, IPatternMatchContext context)` . By calling the `match()` method, NatSpec can decide whether a syntax pattern part matches a given list of words. If all parts of a syntax pattern match all words in a sentence, the sentence is considered as matched completely.

When asked for a match, each syntax pattern part can return a custom match object (see `ISyntaxPatternPartMatch`) which stores the data that is relevant for the code generation. For example, class `DateArgument` returns a `DateMatch` object containing the actual date found in the sentence. Custom syntax pattern parts can either implement interface `ISyntaxPatternPartMatch` or extend class `AbstractSyntaxPatternPartMatch` . The latter method is recommended.

To generate code for matched sentences NatSpec calls `createUserData()` on the syntax pattern object. If this method returns an object implementing `ICodeFragment` the code is obtained by calling `generateSourceCode()` . To compose the code, the syntax pattern can access the matched parts using the parameter `match` which implements `ISyntaxPatternMatch` and therefore gives access to all matched pattern parts. To easily get started with the pattern provider API one may also consider the example provider class `JavaCustomizationProvider` . This class is part of the NatSpec example workspace and contained in the project `de.devboost.natspec.java.customization` .

4.2 Synonym Provider API

Extending NatSpec with custom synonym providers is similar to the implementation of custom pattern providers. One must create a class in an OSGi bundle located in the current workspace that implements `ISynonymProvider` . This class will be automatically instantiated and asked for synonyms by NatSpec if required. Similar to syntax patterns, synonyms can be specific to certain specifications (e.g., depending on the project that contains the specification).