Closing the Gap between Unit Test Code and Documentation

Karsten Stöcker
Leipzig University, Leipzig, Germany
Waseda University, Tokyo, Japan
stoecker@fuji.waseda.jp

Hironori Washizaki
Waseda University, Tokyo, Japan
washizaki@waseda.jp

Yoshiaki Fukazawa
Waseda University, Tokyo, Japan
fukazawa@waseda.jp

***Abstract*—Test Driven Development as well as the documentation of tests and their architecture are today an important pillar of software quality assurance. The change of requirements during the implementation phase entails a need to change tests as well as the test documentation of the software. Since unit tests are specified in the implementation language, an interdisciplinary readable documentation must be maintained, which is structurally easier to comprehend and also make the test transparent for persons who are not involved into code writing. This leads to additional effort, costs and possibly inconsistencies between the test and its documentation. This gap in the workflow could be closed by Tanni – a domain specific language, which allows the specification of test cases in the form of interdisciplinary readable tables without requiring programming skills. Based on them executable test code for the respective unit test framework is generated. This merges specification and documentation of unit test cases to one step of work. By this the mentioned additional effort, costs and imminent inconsistencies can be reduced. The Language Workbench Meta Programming System from JetBrains serves as a technological base and is enabler for further positive effects which possibly could be gained by using the described language.**

Keywords—Unit Test, Test Case Specification, Java, Documentation, Meta Programming System, Domain-specific Language, Test Code Generation

I. Introduction

In the last few years, the approach of Test-driven Development (TDD)[1] has spread and contributed to the improvement of software quality [2]. This spread surely was fueled by increasing agility in the development in general as well as the support by powerful unit test frameworks - e.g. JUnit [3].

But testing alone is not a guarantee for quality and reliable software. An appropriate test architecture as well as the testing of the right software components is crucial for a good software quality and thus the project success [4](pp.205-231). Hence a clear overview is necessary. Not infrequently employees leave a company or change to another project. This threatens the loss of knowledge in this area. Furthermore, the process of testing software from fields of use with a higher security requirement (e.g., financial services) and its documentation is often subject to regulatory requirements, which are regularly examined by superior authorities (e.g. BaFin in Germany[5]). Amongst these reasons a comprehensive and constantly updated documentation is indispensable. A weakness of unit test frameworks shows up.

A. The Gap between Unit Test Code and Documentation

Formulating unit tests in the implementation language of the software product itself is comfortable for the developer, since he does not need any knowledge about another language. However, these are incomprehensible for persons without programming knowledge – also in their structure. An additional transparent documentation in a multidisciplinary form gets required, which describes what and how something was tested. But especially through today's agile development it is not uncommon that requirements to the software change during the implementation. In the case of TDD this involves a change in the tests, which in turn necessitates a change to the mentioned documentation by the developers [4](pp.57-58).

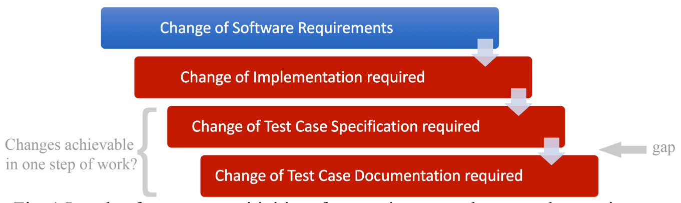

Change of Software Requirements
Change of Implementation required
Change of Test Case Specification required
Change of Test Case Documentation required
Changes achievable in one step of work?
gap

Fig. 1 Levels of necessary acitivities after requirements change and emerging gap

The question arises, whether the effort of the change – extending over three levels (Fig. 1) – can be reduced? Is it possible to close the gap between unit test case specification and documentation by merging this to one working step?

The syntax for the formulation of test cases is a topic that has been repeatedly addressed. As part of the development there have been multiple changes to the syntax of JUnit [6][7]. Furthermore idea descriptions and prototypical implementations devoted to the topic of comfortable unit test case specification syntax are existing [8]. The main aim here is often to make it more lightweight as well as the approach to natural language constructs in order to increase readability. But none of these changes and ideas allow the specification and documentation of unit tests – in a for all people readable, transparent and comprehensive form – at the same time.

II. The Solution – Tanni

Tanni (jap. unit) persues a totally different approach. Unit tests are specified in the form of tables which are readable to every person by waiving programming language syntax without loosing test functionality. Therefore these flexible tables are composed of labeled mandatory and optional fields which cover the parameters of unit test cases like

- unit under test
- assertion type
- expected result
- timeout
- expected exception
- …

and thus allow a comprehensive test descriptions.

Based on these defining tables, executable code for the respective unit test framework gets generated. For the prototypical implementation discussed in the further course of the chapter JUnit was chosen as the target framework of code generation, because of their distribution [9]. The technological fundamental for the implementation is Meta Programming System from JetBrains [10].

A. Excurs to Meta Programming System

Meta Programming System (MPS) is a Language Workbench (LWB) which was first released in 2005 under the Apache 2.0 license. On the one hand it offers the possibility of using several different languages in one project (Solution Projects). On the other hand, it allows the development of own Domain-specific Languages (Language Projects)[11](p.22). Thus, Meta Programming System directly supports the approach of Language Oriented Programming (LOP)[12]. The description of a language is based on aspects, which respectively represent a specific viewing angle to the language. Among other aspects, the following are particularly important to understand the structure of Tanni:

- *Structure Aspect*: Definition of permissible Abstract Syntax Trees (AST)[11](p.20) by defining Concept Node Hierarchies. A concept represents a node in the AST.
- *Editor Aspect:* Description of the presentation and processing possibilities of concept nodes in two different areas of the Language Workbench-GUI (Editor View, Inspector View)[11](p.39) by a cell structure.
- *Generator Aspect:* Description of the translation of Concept/AST to target language [11](pp.24-25).

The provision of functions that allow comfortable working with the developed language, such as autocomplete (Fig. 3) and syntax highlighting (Fig. 4) are supported by MPS so that the provision gets simplified by the use of these powerful LWB and help to make Tanni as userfriendly as possible.

B. The Concept Node Hierarchy

The idea comprise to make the full functionality of the JUnit framework accessible in Tanni. For this purpose, the API structure of the framework [13] was examined in the first step. The base for the investigation was the version 4.12. This decision was based on the fact that JUnit 5 is not yet finalized and will also be backward compatible to JUnit 4 [14]. Based on this analysis, a concept node hierarchy was derived (Fig. 2) that reflects the terminology of the JUnit framework (e.g. `TestSuite` and `TestCase`) and makes them immediately intelligible in large parts.

The concepts `PreparationStatementInJavaContainer` and `AssertionInJavaContainer` are particularly note-worthy, because they are used to embed plain Java code in the tabular test descriptions. Thus, it should remain possible to use solutions which are not part of the unit test framework (e.g. mocking frameworks) and therefore no corresponding mapping as concept exists until now. Unrecognized recurring command structures could later be formalized as part of Tanni in the form of new concepts that implement the interfaces `IPreparationStatement` and `IAssertion`. This makes Tanni very expendable which simplifies reuse and helps to save ressources.

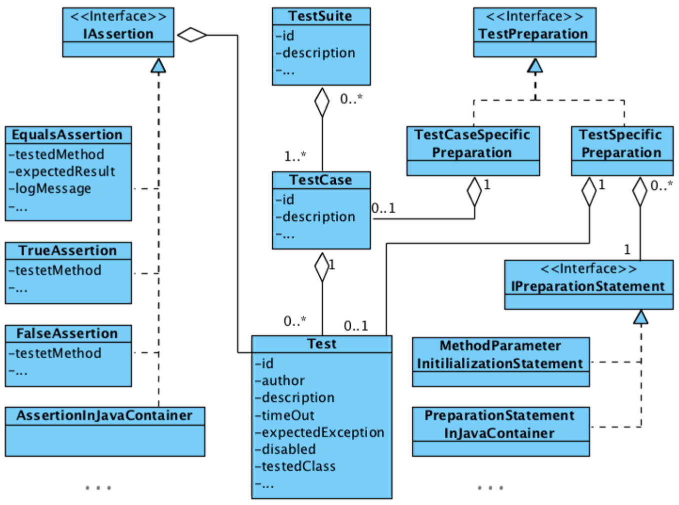
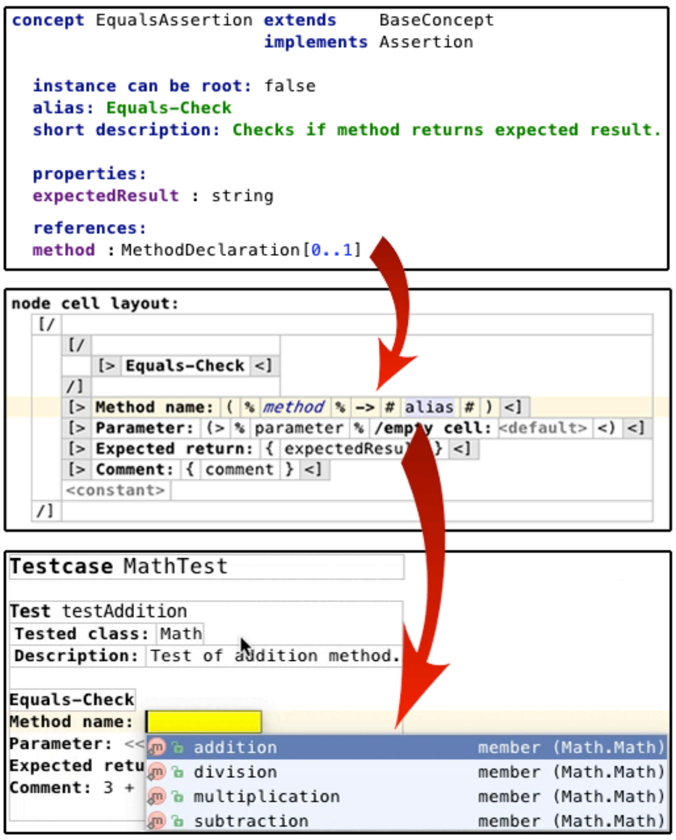


Fig. 2 Extract from the Tanni concept node hierarchy

Based on the definitions of the IEEE standard 829 ("Standard for Software and System Test Documentation")[15], the defined concepts were extended by further attributes (e.g. unique test case identifiers). These are not absolutely necessary to achieve JUnit framework-compatible unit test code, but they are part of a professional test documentation and are also deposited in the generated code (e.g. as comment) to increase traceability.

To ensure error-free addressing of the Java classes and methods, which should be tested, it is necessary to provide autocomplete functionality. This is facilitated by MPS offering a stub mechanism. To achieve the stubs for classes that should be tested it is necessary that the source folder of them is added



```
concept EqualsAssertion extends    BaseConcept
                        implements Assertion

  instance can be root: false
  alias: Equals-Check
  short description: Checks if method returns expected result.

  properties:
  expectedResult : string

  references:
  method  : MethodDeclaration[0..1]
```

```
node cell layout:
  [/
    [/
      [> Equals-Check <]
    /]
    [> Method name: ( % method % -> # alias # ) <]
    [> Parameter: (> % parameter % /empty cell: <default> <) <]
    [> Expected return: { expectedResult } <]
    [> Comment: { comment } <]
    <constant>
  /]
```

```
Testcase MathTest

Test testAddition
 Tested class: Math
 Description: Test of addition method.

Equals-Check
Method name: 
Parameter: <<
Expected retu
Comment: 3 +
```

```
addition          member (Math.Math)
division          member (Math.Math)
multiplication    member (Math.Math)
subtraction       member (Math.Math)
```

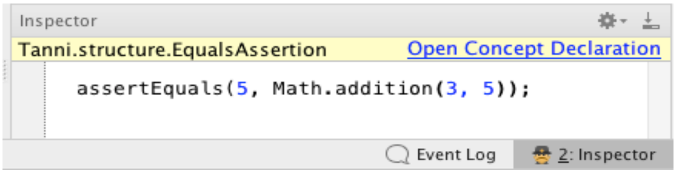
Fig. 3 Java class reference in concept and editor along with final editor view of Tanni

to the Solution Project as a source. In addition, a reference to classes or methods must be stored in the appropriate concept, by using the reference types `Classifier` and `MethodDeclaration` (Fig. 3).

C. The Editor and Inspector View

As already mentioned, MPS provides two separate views for the visualization of concepts - the editor and the inspector view. The editor view – as main input area in MPS – is used to visualize the mentioned tables and their parameter input fields.

The inspector view is often used to purge the language representation by extracting less relevant or additional information and options to it [11](p.39). Tanni instead uses this view to provide a preview of the generated unit test code (Fig. 4). This makes it possible for a developer to see which framework specific code is generated at any time. Non software developers on the other hand can hide the inspector view and thus focus on the test description in table form.



```
assertEquals(5, Math.addition(3, 5));
```

Fig. 4 Example of unit test code preview at inspector view

III. Further Gainable Advantages

In addition to the pure closing of the gap between test specification and documentation, further positive effects showed up, by the use of Tanni was well as MPS, which are worth to be discussed.

One thing is that MPS supports the versioning of developed languages by providing a so called Migration Aspect [16]. This aspect is used to describe the automatization of necessary steps during migration from one language version to the next in the form of scripts. An updating of existing documentation to newer template (e.g. added new attribute) versions is usually omitted because it must be done manually and so requires a lot effort. But by means of mps migration descriptions older documentation can also be kept up-to-date on the latest template status automatically, which means an increase of documentation consistency/quality.

Another conceivable advantage would be that a change of the unit test framework or adjustments to possible syntax changes of the used framework would be facilitated by the implementation structure of Tanni. For this, it would only be necessary to adapt the "translation descriptions" in the generator aspect (Fig. 5). The number of code lines to be changed manually can thus be massively reduced, thereby effort and costs can be saved.

Furthermore, the documentation quality can be increased by the definition of mandatory fields, which is not possible in this form if as not unusual word processing application templates are used for documentation. In the case of incompletely filled cells MPS refuses to generate unit framework-specific test code due to an impermissible syntax. This ensures, that documentation is always complete. Additional to this the merge also ensures that test and documentation are free of contradictions. What is documented was also tested – and vice versa. Both of these results in a higher test documentation quality.

One last important advantage, that could be drawn from the table oriented structure of Tanni, is that the generation of tests by end users might become possible. Users from specialist areas mostly knows best the pitfalls of their working area (e.g. complex financial calculations). However, the creation of tests using the unit test framework APIs is not possible for them due to lack of knowledge and limited resources to learn. Table structures instead are generally understandable for everyone. Creating tests without complex preparatory work (e.g. use of mocking framework) could be undertaken by these experts of her field. This kind of end user testing maybe could help improve the test quality.

IV. Conclusion And further work

The prototypical implementation of Tanni – based on the previously described structure – has shown at the example of JUnit framework that the gap between unit test case specification and documentation can be closed by a table-oriented domain specific language (Fig. 5). The base for this is the use of a powerful LWB with the characteristics of Meta Programming System, which supports the implementation of a table-oriented, comfortable language.

Work on the prototype has shown that the development of a well thought out concept node hierarchy is nearly most important. The development focus should be placed on this aspect. Interfaces in the field of assertions as well as pre-test preparation methods (`@Before` [17]) do play an important role. Especially the latter one is very important in order not to isolate the language from further components of test environments (e.g. mocking frameworks) and to enable eventually subsequent integration by means of new concept node hierarchy elements.

Beside the closed gap, the discussed additional advantages:

- automated updating of existing documentations after template changes via language versioning → effort reduction and increase of documentation quality/consistency
- unit test code adjustment after framework or framework syntax change by simple generator aspect adjustment → effort reduction
- easy definition of mandatory fields using syntax check, which means no test code generation before completed documentation → ensuring documentation completeness
- increasing end-user involvement by enabling them to write own test cases using the understandable table syntax of Tanni → tests which address the pitfalls of an application field better (e.g. financial calculations)
- merging of documentation and specification ensures consistency – what is documented was also tested and vice versa.

could be gained and let the further pursuit of the approach seem to be worthwhile.

The next step to advance the idea of Tanni should be to test function complete prototype in an small productive environment. The main focus should be to further evaluate whether the designed concept node hierarchy and its attributes are proven in practice or adaptations are necessary. In addition, it should be tried to identify recurring structures in more complex test preparations in order to expand the concept node hierarchy in these area to promote the idea of a table based description and further enhance the comfortability of Tanni.

Lastly, during this experiment it should be evaluated whether the effort for test formulation, documentation and change could be reduced under real operating conditions by using Tanni.

[1] K. Beck, “Test-Driven Development By Example“, Addison-Wesley Professional, 2002.

[2] D. Janzen and H. Saiedian, "Does Test-Driven Development Really Improve Software Design Quality?", IEEE Software, p. 77-84, March/April, 2008.

[3] www.junit.org

[4] I. Sommerville, “Software Engineering”, 9th edition, Pearson Education, 2011.

[5] www.bafin.de/EN

[6] A. Glover, “Jump into JUnit 4 - Streamlined testing with Java 5 annotations”, IBM Coorporation, 2007.

[7] S. Bley, “What’s new in JUnit 5?”, Saxonia Systems AG, 08.02.2016, (*https://www.sogehtsoftware.de/blog/post/whats-new-in-junit-5*).

[8] P. Kainulainen, „Turning Assertions into a Domain Specific Language“, 16.11.2013, (*https://www.petrikainulainen.net/programming/unit-testing/turning-assertions-into-a-domain-specific-language/).*

[9] A. Zhitnitsky, “Analysis about the Distribution of Libraries on GitHub”, 14.04.2015, (*http://blog.takipi.com/we-analyzed-60678-libraries-on-github-here-are-the-top-100/).*

[10] www.jetbrains.com/mps/

[11] F. Campagne, “The MPS Language Workbench – Volume 1”, 2nd edition, Campagnelab, 2015.

[12] M. Fowler, “Language Workbenches: The Killer-App for Domain Specific Languages?“, 12.06.2015, (https://www.martinfowler.com/articles/languageWorkbench.html)

[13] www.junit.org/junit4/javadoc/4.12/

[14] S. Bechthold, S. Brannen, J. Link, M. Merdes and M. Phillipp, “JUnit 5 User Guide”, Version 5.0.0-M2, 2016, (*http://junit.org/junit5/docs/current/user-guide/).*

[15] www.standards.ieee.org/findstds/standard/829-2008.html

[16] F. Campagne, “The MPS Language Workbench – Volume 2”, 1st edition, Campagnelab, 2016, pp. 27-37.

[17] www.junit.sourceforge.net/javadoc/org/junit/Before.html