

Automated Static Analysis of Unit Test Code

Rudolf Ramler, Michael Moser and Josef Pichler

Software Analytics and Evolution
Software Competence Center Hagenberg
Softwarepark 21, A-4232 Hagenberg, Austria
{rudolf.ramler, michael.moser, josef.pichler}@scch.at

Abstract—Automated unit tests are an essential software quality assurance measure that is widely used in practice. In many projects, thus, large volumes of test code have co-evolved with the production code throughout development. Like any other code, test code too may contain faults, affecting the effectiveness, reliability and usefulness of the tests. Furthermore, throughout the software system’s ongoing development and maintenance phase, the test code too has to be constantly adapted and maintained. To support detecting problems in test code and improving its quality, we implemented 42 static checks for analyzing JUnit tests. These checks encompass best practices for writing unit tests, common issues observed in using xUnit frameworks, and our experiences collected from several years of providing trainings and reviews of test code for industry and in teaching. The checks can be run using the open source analysis tool PMD. In addition to a description of the implemented checks and their rationale, we demonstrate the applicability of using static analysis for test code by analyzing the unit tests of the open source project JFreeChart.

Keywords—unit testing; static analysis; test code.

I. INTRODUCTION

Automated unit tests represent one of the most widely used software quality assurance measures in software development. Unit testing frameworks [1] support the implementation of test cases using the same language and technology as for implementing the system under test (SUT). In many projects, thus, large volumes of test code have co-evolved together with the production code throughout development [2]. Ratios between test and production code of up to 1:1 have been reported [3].

Like any other code, test code too contains faults [4]. Such faults negatively affect the effectiveness, reliability, and usefulness of the automated tests. Faults in test code may result in tests that fail on a correct SUT causing unnecessary debugging effort, and tests that pass despite faults are present in the SUT causing costly failures in later test stages or in the field. False alarms and missed faults decrease the developers’ confidence in the tests, damaging the motivation to invest time and effort in writing tests.

Furthermore, throughout ongoing development and subsequent maintenance of the SUT, the corresponding tests have to be modified and maintained as well. Poor quality of the test code in terms of understandability, adaptability, etc. has a negative impact on the overall maintenance effort. If the effort for changing the tests is too high, developers may simply de-

cide to temporarily ignore the existing tests – in particular failing ones – rather than to maintain them. As a result the tests become out of sync and their maintainability diminishes further until the tests are eventually discarded.

Automated static analysis has been successfully used to identify potential coding issues and latent maintenance problems [5]. Static analysis tools are frequently applied to production code and provide a wide range of analysis rules and bug pattern detectors. In this paper we report on the application of automated static analysis as an aid for analyzing and improving the quality of unit test code. Our work focuses on tests based on the framework JUnit, as it is one of the most prominent and widely used testing frameworks.

In Section 2 we provide an overview of commonly encountered issues and problems related to the implementation of tests. Section 3 describes 42 static checks for detecting coding issues in JUnit tests we have implemented by extending and refining the ruleset of the open source analysis tool *PMD*. The results from evaluating the rules on the open source project *JFreeChart* are presented in Section 4. Section 5 summarizes our findings and concludes the paper.

II. COMMON ISSUES IN UNIT TEST CODE

From providing trainings and code reviews for industry over several years we collected an extensive set of typical issues and problems related to the implementation of tests. In the literature such issues have been described as test smells (e.g., [1], [3]). This section provides an overview of issues we commonly encountered in unit test code. Based on our experience we distinguish four key problem areas that should be investigated when analyzing unit test code: (a) the usage of the xUnit test framework, (b) the implementation of the unit tests, (c) the maintenance of the test suite, and (d) the testability of the SUT.

A. Issues related to the Usage of the xUnit Test Framework

Tests implement interfaces and extension points of a specific test framework in order to enforce design principles and protocols defined by the framework. For example, early versions of JUnit (up to 3.x) required test classes to be derived from the abstract class *junit.framework.TestCase* and test methods had to adhere to the naming convention of starting with the word “test”. Since JUnit version 4, these mechanisms have been replaced by the use of annotations (e.g., *@Test*). The changes

made in version 4 have been a source of confusion¹ as they introduced not only syntactical changes but also affected established concepts such as implementing test setup and teardown procedures, combining tests to suites, or writing tests for expected exceptions (versus handling unexpected exceptions). Furthermore, additional concepts may be introduced by the use of helper libraries extending a unit test framework or by mock frameworks. Understanding these concepts is fundamental for implementing unit tests correctly. However, the complexity of the test frameworks' interfaces and their sometimes sparse documentation frequently raise issues and create misunderstandings, not only for beginners but also in projects where different framework versions and libraries are mixed.

B. Issues related to the Implementation of Unit Tests

Unit tests have to follow the same test design principles as any other tests. Every test has a distinct goal, which is examined in a specific execution scenario. The effectiveness of a test is its ability to trigger and detect failures, i.e., deviations between the actual and expected behavior. The effectiveness is mainly influenced by the test case design.

Even though test design techniques provide systematic guidance on how to develop effective test cases, unit tests often evolve in an ad-hoc manner constrained by the available time and driven by the personal preferences of individual developers. Hence, reviewing the implementation of unit tests often reveals issues that are either caused by a lack of appropriate test design or by incorrectly transferring the designed logical test into an executable test implementation. Incomplete checks, e.g. missing asserts on return values, are among the most frequently encountered issues. Moreover, tests may not even contain any checks at or only very simple ones such as asserting that a reference is not null. Such tests will always pass unless an unexpected exception is thrown.

Sometimes the opposite is true and tests contain too many checks and/or checks that are too strict (e.g., assuming a specific implementation, timing or environment/language specific setting). Thus, these tests will fail once there is a small change in the execution environment. Furthermore, tests may introduce side effects causing dependencies among tests so they cannot be repeated or will fail when executed in a different order.

C. Issues related to the Maintenance of Test Suites

Maintenance related issues are similar to implementation issues as their root cause is often a problem in the implementation of the test. Nevertheless, maintenance issues do not directly impact initial test runs. Their negative consequences are usually experienced only over time as they are triggered when the system under test evolves. For example, when the internal implementation of a tested unit is changed, the associated tests break, even though the public interface of the unit remained unchanged. Such failures are usually false positives caused by tests that depend on irrelevant details of the unit's internal implementation. Furthermore, a refactoring of the SUT may cause huge ripple effects on the test code due to the tests' tight coupling with the production code.

Especially in situations where the changes in the SUT are mainly "quick fixes", the related maintenance effort for the test code is often considered a pointless extra burden. Instead of adapting the tests, they are commented out or are simply ignored. Berner et al. [6] observed that automated test suites have a strong tendency to be unforgiving when suspended and not run for a short time. The effort to run and maintain an automated test suite increases by a disproportional amount if not run frequently. Test suites often degrade into a state where the tests are inconsistent and difficult to understand and, in the end, they are completely abandoned. According to Berner et al. this is one of the main causes for the failure of automated testing.

Maintenance issues often arise from bad practices that can be detected in reviews. Dependencies on internal implementation details are caused, for example, by testing private methods, comparing objects using their string representation, or relying on the order of returned data sets. Another example is re-implementing logic from the production code as part of the test code, e.g., for creating test setups, for computing expected results, or for simulating the behavior of dependent units in mocks. Implementing (business) logic in test code increases the risk of faults in the test code and introduces a "logical dependency" to the SUT by duplicating production code.

D. Issues related to the Testability of the SUT

A major blocker for effective unit testing in practice is the frequently observed lack of testability of a SUT [7]. While unit testing, test-driven development and design for testability have been widely adopted, many projects still have to cope with large parts of inherited legacy code that are hard or even impossible to test. A major testability problem is the presence of external dependencies. Thus, units cannot be tested in isolation and their actual behavior is influenced by external components or services that cannot be controlled by the test.

Identifying and dealing with testability issues (in legacy code) is out of scope of this paper. However, there are several issues that are usually symptoms indicating underlying testability problems. A typical indicator is a complex, bloated setup or teardown routine due to the fact that the tested unit cannot be created as isolated object. Instead, all dependent objects have to be setup in a specific state. The application of dummy implementations and mock objects may be a solution. Yet their heavy use again results in complex, bloated setup/teardown code. Further symptoms are aggressive mocking (e.g., using reflection to manipulate constructors and private methods) or checking the correctness of setup code by including additional assertions in setup/teardown methods.

Furthermore, since the test does not have sufficient control over the external dependencies, the implementation of the tests sometimes includes conditional logic to dynamically react on the behavior of the dependencies. However, this violates the principle that a test should execute the SUT in exactly one, clearly defined scenario. Otherwise test cases tend to become non-deterministic ("flakey") and failures are hard to reproduce.

III. STATIC ANALYSIS RULES FOR UNIT TEST CODE

Tool support for static analysis is widely available and intensively used in the development of software systems. Several

¹ <http://stackoverflow.com/questions/2635839/junit-confusion-use-extend-testcase-or-test>

tools exist for analyzing Java code. Prominent examples are *PMD*² or *FindBugs*³. These tools automate the process of (manually) checking the adherence to coding conventions and standards, detecting suspicious patterns in the source code, computing code metrics, or identifying duplicate code blocks.

Static analysis works by examining the program elements, the program structure, or by approximating the program states of the program. Many issues can already be identified by examining the program elements as given in the source code, which is usually done on the representation of the source code in form of its abstract syntax tree (AST). The analysis itself is often based on a pattern-matching approach, where rules for AST patterns are defined and the analysis is accomplished by traversing the AST and checking the visited structures against the defined rules.

The available tools provide a large body of predefined rules and bug patterns. *PMD* implements more than 300 rules (version 5.4.1) and *FindBugs* recognizes over 400 patterns (version 3.0.1). The rulesets even include rules/patterns dedicated to analyzing JUnit code; 12 rules in *PMD* and 6 patterns in *FindBugs*. A detailed description of *PMD*'s rules and their rationale can be found in the supplementary on-line chapter⁴ of Copeland's book *PMD Applied* [8]. However, the rules and patterns of the available tools cover only a small fraction of issues discussed in the previous section.

We refined and extended the *PMD* ruleset by implementing 42 additional rules. Our ruleset address mainly maintenance issues and the improper use of the xUnit framework. However, it also covers some of the common test implementation problems as many of the rules can be mapped to more than one issue type. They are specifically related to following aspects of JUnit test code (the number of rules is shown in brackets):

- *Rules related to assert statements* (15): JUnit provides a range of different statements to concisely specify the various condition to be asserted, e.g., *assertTrue()*, *assertEquals()*, *assertNotNull()*. The rules make sure that the correct assert is used and that it is used correctly.
- *Rules for naming conventions* (8): JUnit best practices include naming conventions established to improve the readability of the test code and its structure.
- *Rules for detecting logic in test cases* (6): Tests should have exactly one, clearly defined execution scenario. Conditional logic in test cases violates this principle. Furthermore, logic introduces additional complexity, decreases the understandability, and increases the risk of introducing faults in the implementation of the tests.
- *Rules for implementing setup/teardown routines* (8): The setup establishes the context in which the test is executed. If implemented correctly, JUnit guarantees that the setup is created before each test run and that is torn down afterwards, even if the test execution has failed.

- *Miscellaneous rules for unit test code* (5): Several other rules help to avoid bad coding styles like using localized data, hardcoded references, console output, or class variables as they frequently result in undesirable dependencies on the test environment or between tests.

The decision to use *PMD* was influenced by the tool's easy extensibility. Rules can be expressed either as Java classes or as XPath queries. In the analysis process they are automatically executed on the corresponding nodes of the AST. The following XPath expression shows the implementation of the rule *JUnitTestsShouldNotCatchExceptionsWithoutFail*.

```
// TryStatement
[ not( Block // PrimaryPrefix / Name
  [ @Image = ' fail ' ) ] ]
```

The rule detects try-catch blocks used for testing expected exceptions that do not contain a *fail()* statement at the end of the try block. If the *fail()* is missing, the test may pass, even if the expected exception has not been thrown. Figure 1 shows the code example provided as part of the rule's documentation.

```
01 public class MyTest extends TestCase {
02
03     public void testTryBlockWithoutFail() {
04
05         try {
06             int x = 1/0;
07             // fail is missing
08         } catch (Exception e) {
09             // Exception should be caught
10         }
11     }
12 }
```

Fig. 1. Code example of a JUnit test missing *fail()* in try block

IV. RESULTS FROM ANALYZING JFREECHART

*JFreeChart*⁵ is an open source chart library for Java. Since its start in 2000, the *JFreeChart* project has developed an extensive feature set and a code base of more than 300 KLOC of Java code. The system is very well tested by more than 7,000 test methods (i.e., test cases) in about 500 JUnit test classes. Many of these tests are still relying on JUnit 3.8.1 although newer parts of the project already use JUnit 4.x. Like many other projects with large test suites and a long development history, *JFreeChart* suffers from typical issues in unit test code. Bug #1119⁶ in the project's issue database describes one example. It documents the case of several unit tests failing unexpectedly when executed in a different order.

For our evaluation, we used *JFreeChart* version 1.0.12 and analyzed the tests in the package *org.jfree.chart.junit*. Table I provides an overview of the basic metrics. The package contains 28 test classes (files) with a total of 5,867 LOC, 96 test methods (NOM) and 280 assert statements (NOA) – an average of 2.9 asserts per test method and 10 asserts per test class, respectively. The top class containing most assert statements (*file #1*) has 63 asserts in only 3 test methods. Running the 42 static analysis rules on the 28 test classes revealed a total of 982 violations (*avg* = 35.1, *max* = 140, *min* = 6).

² <http://pmd.github.io/>

³ <http://findbugs.sourceforge.net/>

⁴ http://pmdapplied.com/pmd_applied_chapter_11.pdf

⁵ <http://www.jfree.org/jfreechart/>

⁶ <http://sourceforge.net/p/jfreechart/bugs/1119/>

TABLE I. METRICS AND RULE VIOLATIONS FOR *ORG.JFREE.CHART.JUNIT*

File	LOC	NOM	NOA	Violations	A	N	L	S	M
1	328	3	63	140	129	9	2		
2	409	4	46	112	95	13	4		
3	240	7	21	67	45	16	6		
4	222	6	20	60	43	11	6		
5	195	4	16	53	33	15	5		
6	162	3	8	33	18	9	6		
7	154	3	8	30	14	12	4		
8	210	4	6	27	11	12	1	1	2
9	222	4	6	27	11	12	1	1	2
10	233	4	6	27	11	12	1	1	2
11	228	4	6	27	11	12	1	1	2
12	228	4	6	27	11	12	1	1	2
13	227	4	6	27	11	12	1	1	2
14	227	4	6	27	11	12	1	1	2
15	233	4	5	27	11	12	1	1	2
16	213	3	5	26	8	11	4	1	2
17	71	3	5	26	8	11	4	1	2
18	178	3	5	26	8	11	4	1	2
19	210	3	5	26	8	11	4	1	2
20	214	3	5	26	8	11	4	1	2
21	210	3	5	26	8	11	4	1	2
22	210	3	5	26	8	11	4	1	2
23	327	5	5	28	13	10	2	1	2
24	173	2	3	19	7	8	1	1	2
25	146	1	2	15	5	7		1	2
26	208	3	3	14	7	4	1	1	1
27	86	1	2	7	5	2			
28	103	1	1	6	1	3	2		
Total	5867	96	280	982	559	292	75	19	37
					57%	30%	8%	2%	4%

As one can observe from Table I, the number of violations is closely correlated to the number of assert statements (NOA). This is due to the fact that most detected violations are related to assertions (category A: 57%). Second are violations of naming conventions (N: 30%), followed by logic found in test code (L: 8%), violations in setup/teardown routines (S: 2%), and miscellaneous others (M: 4%). A closer examination of the results shows that the violations can be categorized as follows:

Confirmed faults: In many occasions, JUnit assert methods are used incorrectly, leading to a high number of violations related to assertions. For example, the following comparisons found in *file 1* can be expressed by more appropriate assert methods provided by JUnit.

```
assertTrue(t1 != t2);
assertTrue(t1.getClass() == t2.getClass());
assertTrue(t1.equals(t2));
```

Low priority violations: There is one rule checking if asserts provide a message. Since this message is optional and never used in any of the analyzed tests, this particular rule resulted in 280 indicated violations, which should be prioritized as “recommendations” rather than faults.

Irrelevant results: Overall 292 violations of naming conventions have been found. Many of them may be considered irrelevant as the project uses conventions that are slightly different from the standard (e.g., the names of test classes end with an additional *s* as in *BarChartTests.java*).

Unintentional discoveries: Many of the remaining violations indicate actual issues in the test code. In some instances, however, the violations were related to issues different to those the rules were supposed to detect. For example, the rule for detecting missing fail statements when testing expected exceptions (see above) actually revealed also several instances where irrelevant try-catch blocks were used in the test code.

V. CONCLUSIONS AND NEXT STEPS

For quite a long time, test code has been treated less carefully than production code, not applying the same quality standards and ignoring the latent maintenance problems. As a consequence, many projects suffer from numerous issues in the implementation of test cases that jeopardize the value and benefits of automated unit testing. In this paper we proposed the use of automated static analysis to uncover test related coding problems. 42 rules have been implemented for the static analysis tool *PMD* to analyze various aspects of JUnit test code.

We applied our rules on unit tests of the open source project *JFreeChart*, a well-tested Java system with a long development history, a considerable code size, and a large number of JUnit tests. Our analysis revealed a total of 982 violations only for one test package. The number of violations is surprisingly high. It can be partially attributed to low priority violations and irrelevant results, e.g., conflicting project-specific naming conventions. However, more than one third of the violations are indicating actual faults or, at least, symptoms of underlying problems in the test code.

From our practical experience we know that reviews are one of the most important measures to establish sustainable unit test suites. Reviews help to spot problems in the implementation of the tests, improve the test code and architecture, and increase the awareness about best practices in writing tests. However, reviews are usually time-consuming manual tasks. Using automated static analysis helps detecting common issues and makes subsequent reviews more efficient and effective.

Our next step will be to use the insights gained from the first analysis of *JFreeChart* to fine-tune the ruleset and to conduct a more comprehensive evaluation on open source projects. Further work includes the application of the rules for reviewing test code from industry projects and the inclusion of the checks as part of their build and continuous integration systems.

REFERENCES

- [1] G. Meszaros, “xUnit test patterns: Refactoring test code,” Pearson Education, 2007.
- [2] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, “Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining,” *Empirical Software Engineering*, 16(3), pp. 325-364, June 2011.
- [3] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, “Refactoring test code,” 2nd Int. Conf. on Extreme Programming and Flexible Processes in Software Engineering (XP 2001), 2001.
- [4] A. Vahabzadeh, A. Milani Fard, and A. Mesbah, “An Empirical Study of Bugs in Test Code,” *IEEE Int. Conf. on Software Maintenance and Evolution (ICSME 2015)*, 2015.
- [5] F. Wedyan, D. Alrmuny, and J. M. Bieman, “The effectiveness of automated static analysis tools for fault detection and refactoring prediction,” *Int. Conf. on Software Testing Verification and Validation (ICST'09)*, 2009.
- [6] S. Berner, R. Weber, and R. K. Keller, “Observations and lessons learned from automated testing,” 27th Int. Conf. on Software engineering (ICSE'05), 2005.
- [7] C. Klammer, and A. Kern, “Writing unit tests: It's now or never!,” Eighth Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), 2015.
- [8] T. Copeland, “PMD Applied: An Easy-to-use Guide for Developers,” Centennial Books, 2005.