# A Lightweight Approach to Determining the Adequacy of Tests as Documentation

Joris Van Geet* and Andy Zaidman**

*University of Antwerp, Belgium
Joris.VanGeet@ua.ac.be
**Delft University of Technology, The Netherlands
& University of Antwerp, Belgium
Andy.Zaidman@ua.ac.be

## Abstract

*Programming process paradigms such as the Agile process and eXtreme Programming (XP) tend to minimise ceremony, favouring working code over documentation. They do, however, advocate the use of tests as a form of "living documentation". This research tries to make an initial assessment of whether these unit tests can indeed serve as a form of full-fledged documentation. The lightweight approach we propose is mainly based on the number of units that is covered by each unit test. This paper discusses the approach, the corresponding tool and the results of a first case study.*

## 1 Introduction

The program comprehension process a user goes through when studying a piece of software can benefit greatly from having up to date documentation available. However, often the documentation of a software project is either out-dated or non-existent. Programming practices such as the Agile process or the XP process even have a tendency to minimise documentation, as these processes value working code over comprehensive documentation [3].

Both Agile programming and XP emphasise testing and even advocate the use of a test-driven approach when writing a new piece of software [2]. Because the tests are written first, they completely define what the code should do. As such, the tests can be considered as a form of "living" documentation that can be consulted when one wants to learn what the code is supposed to do [4].

Our research aim then is to make an initial assessment of the quality of the unit tests with regard to their adequacy as documentation. To determine their adequacy, we will look at two criteria of tests, namely:

1. The test coverage of the system, i.e. how much of the system is actually tested.
2. Whether each test is focused on a single unit of the system or whether each test covers a number of units.

This second criterion forms the basis for our hypothesis: *unit tests are possibly not adequate enough for documentation purposes when they cover a number of units*. This basic idea stems from the fact that when a unit test covers multiple units of production code, the unit test will be harder to understand because of an increase in coupling and complexity. A similar observation has been made by Selby and Basili when it comes to understanding "regular" production code [5]. This is one of the reasons for the pursuit of low levels of coupling.

To determine these "test dependencies" we rely on dynamic analysis, which, in the presence of polymorphism, allows us to circumvent expensive slicing operations.

As a case study to determine the test coverage and extract the test dependencies, we used Apache Ant[1], a widely used Java build tool. We determined its test coverage for a number of versions and extracted the test dependencies from the latest available version.

[1] Form more information, see: http://ant.apache.org

## 2 Unit Tests as Documentation

Testing comes in many forms and can be classified in various ways. The "Guide to the Software Engineering Body of Knowledge" (SWEBOK) [1] provides some interesting classifications. One of them is based on the *granularity of testing*:

- **Component/Unit testing** is concerned with verifying functionality of small and (clearly) separable components.
- **Integration testing** aims at verifying the interaction between components. Usually these components have already been tested by the previous strategy.
- **System testing** tests the system as a whole. This strategy is considered useful for testing non-functional requirements, as the functional requirements should have been tested by the previous two strategies.

It should be noted, however, that the boundary between component testing and integration testing is blurred for object oriented systems as objects are used at all stages of the software process [6]. This observation by Sommerville is interesting because it conflicts with the criteria for the adequacy of tests as documentation, which we set out in Section 1.

It is our opinion that to have optimal documentation, i.e. to be able to understand each unit present in the system, each unit should be documented. As a consequence we expect each unit to be tested, which we can evaluate by determining the test coverage, but we also expect each unit to be tested *in isolation*, to have a clear and unrestricted view of how the unit works. Furthermore, we acknowledge the fact that when units are not tested in isolation, their complexity tends to increase, which can also hinder understandability. We are aware of the fact that certain units cannot be tested in complete isolation, but the usage of stubs can be beneficial to the understandability because they are often less complex than their actual implementations.

## 3 Tool

When trying to determine whether each test command tests only a single unit of production code (criterion 2 from Section 1), we need to extract test dependencies. A test dependency being *the relation between a unit of code and its invoking test command*. Since we focus on the JUnit testing framework, we define a unit of code as a (production) method and a test command as a unit test method.

For extracting the test dependencies we created a tool with a pipe and filter architecture. The tool starts by tracing the execution of the test scenario(s), followed by an analysis of that trace data to eventually result in two xml files that both contain the same test dependency information, albeit in a different form. The first file contains for each unit of production code the test commands which invoke the unit, while the second XML file contains the inverse relations, namely for each test command, the units of code that are invoked by it.

To trace the different execution scenarios, we used a *profiler agent* implemented with the Java Virtual Machine Profiler Interface (JVMPI) [7]. This agent provides a two way communication path with the virtual machine. We are interested in various events that the virtual machine emits during execution, especially the `method_entry` and `method_exit` events. Our agent specifically listens for these two events as they provide the crucial information for a dynamic call graph. Whenever such an entry or exit occurs, some identification information is written to the trace file containing the fully qualified name of the method, its formal parameters and its return type[2].

Because the virtual machine sends out these events for all methods, including the ones from system classes and third party libraries, we performed a basic form of filtering to only trace packages or classes that are of interest. Note that at this stage we merely store the trace data for further analysis (offline analysis), instead of analysing the trace data on the fly (online analysis).

The trace file from the profiler agent provides us with the necessary raw data to extract test dependencies as it lists the entry and exit of all calls in chronological order. The dependency extractor takes a regular expression to identify the test packages or classes. Methods of such a test class are identified as a test method if they take no arguments and their name starts with the string '`test`', as this is the convention in the JUnit testing framework.

Once we have identified the test methods we can easily deduce all methods that are *tested* by a certain test method, as they appear between entry and exit of that test method. To obtain all the test methods that test a particular method, we inverse this relationship. Finally, we store this information in a proprietary XML format, thereby making the test dependencies explicit in both directions. Furthermore, method calls that appear more than once within the same test method are only listed once, as this tool provides a *flattened call graph* resulting in a *set* of methods for each test method and vice versa.

## 4 Results

As we mentioned before, we used Ant, the well-known build tool, as an initial case study for our experiment. We chose Ant because of its relative simplicity and also because it is widely used, both in the open source community and the

[2]The return type of a method is not necessary to uniquely identify a method. However, the Java Virtual Machine provides this data together with the parameters, we keep it for human readability.

| | method coverage | |
|---|---|---|
| ant version | percentage | bare count |
| 1.6.3 | 61% | 3247/5351 |
| 1.6.4 | 63% | 3399/5363 |
| 1.6.5 | 65% | 3739/5745 |

**Table 1. Method coverage as generated by Emma.**

| ant version | methods | tests | calls |
|---|---|---|---|
| 1.6.3 | 4467 | 1330 | 286499 |
| 1.6.4 | 4472 | 1337 | 288363 |
| 1.6.5 | 4767 | 1407 | 324250 |

**Table 2. Total count of methods, tests methods and method calls.**

| version | mean | $\sigma$ |
|---|---|---|
| 1.6.3 | 68.02 | 190.35 |
| 1.6.4 | 64.48 | 183.49 |
| 1.6.5 | 64.14 | 182.40 |

**Table 3. Average number of test methods for an arbitrary method.**

| version | mean | $\sigma$ |
|---|---|---|
| 1.6.3 | 230.45 | 146.10 |
| 1.6.4 | 215.68 | 136.68 |
| 1.6.5 | 215.41 | 137.01 |

**Table 4. Average number of methods that an arbitrary test method runs through.**

closed source community, as evidenced by the integration of Ant in many commercial IDEs.

The results of our experiment can be divided into four parts.

1. The first part determines the test coverage, for which we used already available tools.
2. The second part deals with numerical data that we retrieved from the Ant distribution.
3. The third part presents anecdotal evidence that we retrieved when studying code fragments for evidence of our findings from the numerical data.
4. The fourth part presents an historical perspective, capturing the evolution of the testing strategy.

### 4.1 Test Coverage

Table 1 gives an overview of the test coverage, more specifically the methods that are covered by the tests. As can be seen, the coverage varies from 61% to 65% percent, depending on the version of the Ant project.

Potentially, this also means that only about 2/3 of the methods are documented, although this standpoint could be considered a little harsh, as, just as with regular documentation, not every part of a system needs to be thoroughly documented.

It is our opinion that a coverage level of about 65% should be sufficient for documentation purposes, although we acknowledge that a higher level of test coverage can – logically – only improve understandability.

### 4.2 Numerical data

Initially, we calculated the *number of unique methods* tested, the *number of test methods* executed and the *total number of method calls* present in our flattened call graph to get a quick feel of the application's test infrastructure. The results of this operation are listed in Table 2.

Based on this information we performed two calculations, namely:

- the average number of test methods that test an arbitrary method (Table 3)
- the average number of methods an arbitrary test method runs through (Table 4)

We can see that, on average, a method is tested by approximately 64 test methods and a test method tests approximately 215 methods. These numbers are shocking in contrast with the *ideal* one to one relation between method and test method. However, the enormous standard deviation[3] of the averages we calculated, suggests that the actual values are highly variable, indicating that further investigation is needed.

To get a better view on the distribution of tested methods and test methods, we represented them in a box plot[4] which uses more robust measurements such as the *median* and other *quartiles* instead of the unstable mean.

Figure 1 illustrates the distribution of the number of methods that are tested by a test method. For version 1.6.5, for example, you can see that half of the test methods test more (and the other half tests less) than 212 methods, since 212 is the median (= the second quartile `Q2`). For the same version you can see that half of the test methods test no more than 331 (the third quartile) and no less than 149 (the first quartile) methods. The distribution is almost symmetric around the median, leaving us with similar results as the

[3] According to [9] standard deviation is the most common measure of *statistical dispersion*. Simply put, standard deviation measures how *spread out* the values in a data set are. Traditionally this measure is represented as $\sigma$.

[4] Please note that we use a stripped version of the traditional box plot [8]. Whereas a standard box plot has a parametrised *acceptable range* to define what an outlier is (usually 3/2 times the *inter quartile range*), our range simply extends to the minimum and the maximum values, thus not explicitly specifying outliers.

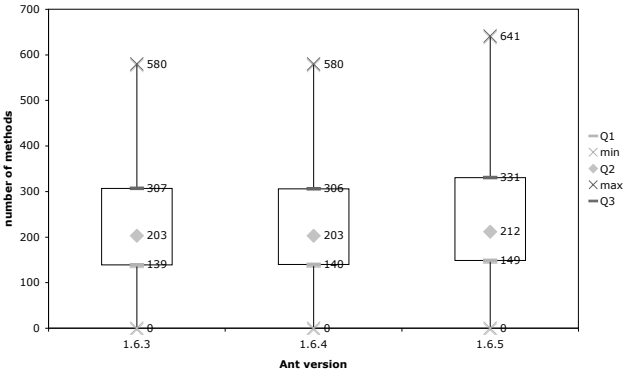


**Figure 1. Box plot of the distribution of the number of methods tested by an arbitrary test method.**

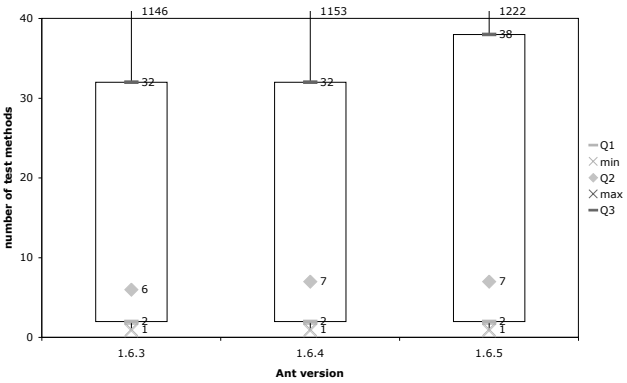


**Figure 2. Box plot of the distribution of the number of test methods that test an arbitrary method.**

```
1  public class RenameTest extends BuildFileTest {
2    public void setUp() {
3      configureProject(
4        "src/etc/testcases/taskdefs/rename.xml"); }
5    public void test1() {
6      expectBuildException("test1",
7          "required argument missing"); }
8    public void test2() {
9      expectBuildException("test2",
10         "required argument missing"); }
11   public void test3() {
12     expectBuildException("test3",
13         "required argument missing"); }
14   public void test4() {
15     expectBuildException("test4",
16         "source and destination the same"); }
17   public void test5() {
18     executeTarget("test5"); }
19   public void test6() {
20     executeTarget("test6"); }
21 }
```

JUnit Test Case

```
1  <project name="xxx-test" basedir="." default="test1">
2     <target name="test1">
3        <rename/>
4     </target>
5     <target name="test2">
6        <rename src=""/>
7     </target>
8     <target name="test3">
9        <rename dest=""/>
10    </target>
11    <target name="test4">
12       <rename src="testdir" dest="testdir"/>
13    </target>
14    <target name="test5">
15       <rename src="template.xml" dest="."/>
16    </target>
17    <target name="test6">
18       <rename src="template.xml" dest="template.tmp"/>
19       <rename src="template.tmp" dest="template.xml"/>
20    </target>
21 </project>
```

Test Build File 'rename.xml'

**Figure 3. Test structure for the `rename` Task**

ones we obtained from the averages.

However, for the number of test methods per method, we do get a different perspective on the distribution. As you can see in figure 2 the box plot is stretched towards the top, meaning that there are only few outliers. For version 1.6.5, for example, half of the methods are tested by no more than seven test methods. A quarter of the methods are even tested by no more than two test methods. As opposed to the average of 64, we can see that 75% of all methods are tested by no more than 38 test methods.

### 4.3 Anecdotal Evidence

Let us dig a little deeper, based on these statistical observations. In Table 2 we can see that Ant 1.6.5 runs approximately 1400 test methods and the box plot in Figure 2 shows us at least one method that is tested by *more than 1200* of these test methods. Based on the rather high average (Table 4) of methods that an arbitrary test method runs through, we suspect even more of these methods that are tested by almost all test methods. Two possible explanations come to mind:

1. Some form of *generic setup code* is executed at every run. This code would have to be located in the test methods themselves[5], since the dependencies of the `setUp()` and `tearDown()` methods are not extracted from the original trace.
2. Some form of *generic test code* provides an execution scenario that is similar for all tests. This would indicate an integration testing strategy or at least a lack of stub usage.

[5]Setting up test data in the test method is not uncommon as it is the only way to initialise different test data for test methods in the same class.

| | bare count | | | percentage | |
|---|---|---|---|---|---|
| ant version | yes | no | total | yes | no |
| v1 | 153 | 53 | 206 | 74.27% | 25.73% |
| v2 | 259 | 81 | 340 | 76.18% | 23.82% |
| v3 | 328 | 107 | 435 | 75.40% | 24.60% |
| v4 | 414 | 150 | 564 | 73.40% | 26.60% |

**Table 5. Second Experiment: Test methods based on `BuildFileTest`.**

Further investigation of the source code revealed the latter option to be true. We queried our dependencies for classes containing those *often called* methods and briefly navigated through the source code with a code browser. Figure 3 nicely illustrates our findings. The top of Figure 3 is a source code extract of the unit test of the Ant `rename` task. As you can see in line 1, this test extends `BuildFileTest`, an abstraction of a unit test that uses a build file as test data. Line 3 shows that, for each test run, a project is configured based on `rename.xml` (bottom of Figure 3), a build file specifically designed for testing the `rename` task. As you can see, each test method has its own *target* in the build file. Executing a test method is nothing more than calling its corresponding target on the newly created project and checking whether or not the task produces the correct exception. When searching for that specific build file, we found similar build files for almost all other tests.

### 4.4 Historical Perspective

We performed similar experiments for four other phases of Ant's evolution, to verify whether production code and tests evolve (more or less) simultaneously. Our observations here are that the test base, and with it the amount of unique methods that are tested, grows consistently with each version. This suggests that newly created test methods test priorly untested functionality. Also, we see that the number of tested methods increases more rapidly than the number of test methods. Furthermore, on average, a test method runs through *more* methods with each subsequent version. This indicates that the integration testing strategy is gaining popularity as the development of Ant evolves.

To investigate this further, we queried our dependencies for the `BuildFileTest` class, as it is the basis of the testing framework in version 1.6.x. The dependencies revealed the presence of this class in all versions except for `v1`. Closer investigation showed that in `v1` similar functionality was available in the `TaskdefsTest` class. As the name indicates, this was only used to test Ant's `taskdef` constructs. In the transition to `v2` this class was renamed to `BuildFileTest` to be used by all Ant constructs. To investigate the evolution of this testing strategy we queried the dependencies for all test methods that call at least one of these framework methods and for all test methods that call none of them. The fourth column of table 5 shows the percentages of test methods that rely on the `BuildFileTest` (or the `TaskdefsTest` for `v1`). A remarkable result at first sight, as we might have expected this percentage to grow in subsequent versions. However, this merely indicates that the integration testing framework was already in place in `v1` (in the form of the `TaskdefsTest`) and that the increased usage of that framework has been consistent over the different versions: for every four new test methods, three were based on the `BuildFileTest`.

### 4.5 Discussion

From the statistical data, the anecdotal evidence and the evolutionary trends that we have discussed in the previous sections, we can conclude that the development team of Ant does not follow a strict unit testing strategy, but rather, follows a strategy that can be classified as an integration testing strategy.

As we have mentioned previously, this kind of testing process can lead to tests that are less suitable for documentation purposes. The main indicator for this reduced adequacy is the fact that a single unit of code cannot be easily understood without also understanding other modules.

This tight coupling might have its consequences when trying to understand a single unit or a small set of units within the system separately. Furthermore, it has been shown that tightly coupled systems are more difficult to understand [5], and there is no reason to assume that this is any different for tests.

## 5 Conclusion

In this paper we have presented a lightweight approach to determine the adequacy of (unit) tests as a form of documentation. Such a form of documentation is actually advocated by the Agile process and eXtreme Programming (XP). To determine the adequacy, we set out two criteria, namely (1) the level of test coverage and (2) whether the tests work in isolation, i.e. how many units of production code are involved in one test command.

With regard to the test coverage we witnessed a method test coverage of around 65%, which is a quite good level, but can be improved for documentation purposes.

With regard to the isolation factor, we witnessed an integration testing strategy in our Ant case study. This integration testing strategy stands opposed to the isolation criterion that we set out and as such, we have to express our concerns with regard to the understandability of these pieces of (test) code, as involving multiple units of code within one test

command increases coupling and complexity, two closely related factors that can influence understandability.

## References


[1] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2001.

[2] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001. Accessed on July 17th 2006, http://agilemanifesto.org/.

[4] E. Hieatt and R. Mee. Going faster: testing the web application. *IEEE Software*, 19(2):60–65, 2002.

[5] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, 2 1991.

[6] I. Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[7] Sun. The Java Virtual Machine Profiler Interface documentation, 2004. Retrieved May 7th 2006.

[8] J. W. Tukey. *Exploratory data analysis*. MA: Addison-Wesley, 1977.

[9] Wikipedia. Standard deviation. Retrieved May 14th 2006.