# Improving Readability of Automatically Generated Unit Tests

Ermira Daka

The University of Sheffield
ermira.daka@sheffield.ac.uk

**Abstract.** Unit testing is a commonly applied technique in object-oriented programming, where classes are tested using small, executable tests written as code. It is a laborious, time-consuming, and error prone job, and even after tests are written, it requires developer to maintain them, and to understand code written by other developers. To support developers, unit test can be generated automatically using different testing techniques. However, since algorithms used for test generation are typically guided by structural criteria, generated unit tests are often long and confusing, and with possible negative effect in the test oracle problem and test maintenance. To overcome this problem we investigate the readability of unit test cases. We propose a domain-specific model of unit test readability based on human judgments, and use this model to guide automated unit test generation. The resulting approach can automatically generate test cases with improved readability with the overall objective of reducing the effort for developers to understand these test cases.

Keywords: Readability, unit testing, automated test generation

## 1 Introduction

Software testing is one of the key processes for quality assurance in software systems. Every little change in a software needs to be tested in order to verify and validate its correctness in the application. Due to the ever-growing size and complexity of software programs, testing is becoming more challenging and costly.

Unit testing is a well known technique in object oriented programming, where different frameworks have been created (e.g. JUnit, NUnit) in order to reduce human effort and increase probability of finding bugs. To further support developers on manually creating test inputs, automated techniques have been devised that based on program under test, generate test suites. This technique is found to be very supportive, however, any test failures require fixing either the software or the failing test, which is a manual activity that needs one to understand the behavior of the test.

How difficult is to understand a unit test depends on many factors. For example, unit tests consist of sequences of calls to instantiate objects and bring them to appropriate states. There are often several such objects within a single test case: at least one for the class under test, several as parameters of calls on this instance, and possibly further objects to permit checks of correctness (for example, when explicitly creating an object that is in the expected state, and then checking whether the state of the object under test equals to that expected object state). The difficulty of understanding such a test case is thus directly influenced by the particular choice of sequence of calls. For example, consider the two test cases in Figure 1. Both tests check the same method *equals()* in the class *ComparatorChain*, but they look different in presentation. In terms of readability features, the first test is longer, contains more lines, defines more variables, has more parameters, that makes it less readable, hence difficult to understand.

```
ByteBuffer byteBuffer0 = ByteBuffer.allocate((int) (byte)0);
BitSet bitSet0 = BitSet.valueOf(byteBuffer0);
ComparatorChain comparatorChain0 = new ComparatorChain((List) null, bitSet0);
ComparatorChain comparatorChain1 = new ComparatorChain();
boolean boolean0 = comparatorChain0.equals((Object) comparatorChain1);
assertFalse(comparatorChain1.equals((Object)comparatorChain0));
assertEquals(false, boolean0);
```

```
ComparatorChain comparatorChain0 = new ComparatorChain();
List list0 = comparatorChain0.comparatorChain;
ComparatorChain comparatorChain1 = new ComparatorChain(list0);
boolean boolean0 = comparatorChain0.equals((Object) comparatorChain1);
assertEquals(true, boolean0);
```

**Fig. 1.** Two versions of a test that exercise the same functionality but have a different appearance and readability.

The aim of this research is to improve the quality of generated test cases by increasing their readability. Readability of generated unit tests will be improved by creating additional optimization functions in the tool that will generate the test. By improving readability in software unit tests cases, time required for understanding the code can be saved, accuracy and productivity will also increase, thus the software can be tested faster and correctly. Consequently, this research focuses on creating a readability model for automatically generating more readable test cases. which further, will serve as an additional test adequacy criteria in EvoSuite (Fraser & Arcuri, 2011) automatic test suite generation tool, which will be able to generate readable test cases.

The main contribution of this study will be improving automatic unit test generation. The readability prediction techniques and concepts presented in this research, implicitly provide requirement for automatic unit test suites with readability optimized.

## 2 Research Questions

The main aim of this work is to optimize test cases in terms of readability such that developers quickly understand the behavior of the test. Therefore, we will investigate on these research questions:

1. How can we create a model that can quantify readability of unit tests?
2. How can we integrate our metric in a testing tool, such that we can improve the readability of automatically generated unit tests?
3. Does improved readability lead to developers understanding the tests better?

## 3 State of the Art in Test Readability

Although test code readability does not affect the functionality of the program at all, it has an important role on increasing the understandability of the code under test while fixing it. Considering that a test case may contain several statements, and all of them are important for bringing the object under a certain state, the probability of interpreting it in the wrong way is very high, which may bring up errors in the whole system.

In order to make test understandable Zhang in his work (Zhang, 2013), presents a technique for test code simplification in semantic level. The work aims to create simpler and shorter test cases that can be better understood. The process starts with an initial test case, replaced referred expressions with alternative ones and each time constructing a simpler test until the same code is still covered. Test minimization is investigated further from Leitner at al. (Leitner, Oriol, Zeller, Ciupa, & Meyer, 2007), who presented a combination of static slicing and delta debugging techniques that automatically minimizes the sequence of failure-inducing method calls in a test suite. The proposed minimization approach based on static slicing is highly efficient, practical, and easy to implement. One other work that include test case minimization, is implemented in Evosuite (test case generation tool) (Fraser & Arcuri, 2011), that provides small test suites (in terms of both test data and assertions) to the user that have to manually verify them. Although, proposed minimization techniques are successful on producing simpler/shorter test cases, they still can generate tests that are too difficult, which require a technique that makes them more readable.

Beside various approaches used to evaluate readability, there is another work done for improving the readability of string test inputs that came from Afshan at al. (Afshan, McMinn, & Stevenson, 2013). This work uses the natural language model approach into a search-based input data generation process in order to improve string inputs from perspective of human readability. Using an empirical study they evaluated their model which showed a great improvement in input readability. Developed technique improves readability of a feature (string inputs) without weakening the test criteria, however it concludes that given a certain time budget for testing, readable inputs reduce oracle checking time, but increase the number of test cases that may be considered.

Program comprehension and readability is an important activity during software evolution and maintenance. Buse and Weimer (Buse & Weimer, 2010) introduced a code readability metric based on human judgments. They collected human annotation data for code snippets and trained a classifier based on those scores. Posnett et al. (Posnett, Hindle, & Devanbu, 2011) used the same dataset to learn a simpler model of code readability, using fewer features based on size, Halstead metrics, and entropy. Although unit tests are also just code in principle, they use a much more restricted set of language features which require a specific model for readability.

Considering that automatically generated unit tests have a specific structure, since they are generated based on some objective, they also have to go through restructuring phase where developers need to understand the testing code. With this work we aim to add the readability as an additional feature in unit test case generation tool, which will help developers understand the tests.

## 4 A Readability Model to Optimize Unit Test Generation

Automatically generated unit tests are often unreadable because there is no immediately measurable metric for the readability of a unit test. Automated test generation techniques can typically only target simple, code-related

**Table 1.** Features used for Test Readability Model

| Feature Name | Total | Max | Avg | Feature Name | Total | Max | Avg | Feature Name | Total | Max | Avg |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Test Length | ✓ | | | Line Length | **✓** | **✓** | ✓ | Identifiers | **✓** | ✓ | ✓ |
| Identifier Length | **✓** | **✓** | **✓** | Indentation | ✓ | ✓ | ✓ | Keywords | ✓ | ✓ | ✓ |
| Numbers | **✓** | ✓ | ✓ | Comments | ✓ | | ✓ | Periods | ✓ | ✓ | ✓ |
| Commas | ✓ | ✓ | ✓ | Spaces | ✓ | | ✓ | Parentheses | ✓ | | ✓ |
| Arithmetic Operators | ✓ | | **✓** | Comparison Operators | ✓ | | ✓ | Assignments | ✓ | | ✓ |
| Branches | ✓ | | **✓** | Loops | ✓ | | **✓** | Blank Lines | ✓ | | ✓ |
| Assertions | **✓** | | ✓ | Method Invocations | ✓ | ✓ | ✓ | Unique Methods | **✓** | ✓ | ✓ |
| Class Instances | **✓** | | ✓ | Unique Classes | ✓ | | ✓ | Characters | ✓ | ✓ | **✓** |
| String Length | ✓ | ✓ | **✓** | Unique Identifiers | **✓** | ✓ | ✓ | Exceptions | **✓** | | |
| String Content Score | | | ✓ | Tokens | ✓ | ✓ | ✓ | Digits | ✓ | ✓ | ✓ |
| Nulls | ✓ | **✓** | **✓** | Castings | ✓ | ✓ | ✓ | Exceptional Comments | ✓ | | ✓ |
| Single Characters | ✓ | ✓ | ✓ | Identifier Ratio | **✓** | | | Method Ratio | **✓** | | |
| Class Ratio | ✓ | | | Unused Identifiers | ✓ | | | Field Accesses | ✓ | ✓ | ✓ |
| Boolean | ✓ | ✓ | ✓ | Types | ✓ | ✓ | ✓ | Strings | ✓ | ✓ | ✓ |
| Arrays | ✓ | ✓ | ✓ | Floats | **✓** | ✓ | ✓ | Token Entropy | ✓ | | |
| Byte Entropy | **✓** | | | Halstead Effort | ✓ | | | Halstead Volume | ✓ | | |
| Halstead Difficulty | ✓ | | | Single Identifier Occurrence | ✓ | ✓ | | Single Char Occurrence | ✓ | ✓ | |

metrics, such as structural coverage criteria. How nice the resulting test cases look like is usually coincidence - from the point of view of an automated test generator. In order to investigate the readability of test cases that are written or generated, we performed a couple of experiments with human annotators that were able to judge on test code readability.

A high-level overview of our approach is provided in Figure 1. Our approach consists the following steps: 1.) Training data collection. 2.) Feature extraction for training set, 3.) Machine learning model, and 4.) Automatically generating more readable test cases.

To collect our readability data we performed an experiment on Amazon Mechanical Turk[1], where human annotators were presented with a test case, and based on their perception of readability they had to rate it with a score from 1 to 5. Test methods were selected from open-source Java projects (Apache commons, poi, trove, jfreechart, joda, jdom, itext and guava), containing automatically generated test cases with EvoSuite (Fraser & Arcuri, 2011), and manually written test cases. Participants were required to pass a Java qualification test to ensure familiarity with the language. As result we collected 15,669 human judgments of readability (in a range of 1–5) on 450 unit test cases.

For our readability machine learning process, we defined a set of 116 initial structural, logical and density features listed in Table 1. In order to extract features from unit test cases, we implemented an application that results with a numerical vector of size 117 where each position represents a value for a feature, while the last position is reserved for class attribute (value to be predicted). We extracted features of 450 test cases and together with average scores collected from human annotators, we used them for model training in Weka (Holmes, Donkin, & Witten, 1994) data mining software. To learn the *test readability model*, we used a simple linear regression learner (Witten, Frank, & Hall, 2011), although in principle other regression learners are also applicable (e.g., multilayer perceptron (Witten et al., 2011)).

We investigated which features have the most predictive power by re-running our all-annotators analysis using different feature selection techniques (one feature at time, leave one feature out, reliefF, correlation, forward feature selection, backward feature selection). Although different techniques agreed with each other, with forward feature selection we learned a formal model based on 24 features, including line width, aspects of the identifiers, and byte entropy.

In order to test our model prediction performance, we conducted another experiment with unit test pairs. Each pair contains test cases with equal functionality but unequal in terms of readability features. Annotators were presented with pairs of test cases, and based on their readability perception they had to chose the most readable sample in the pair. With the trained model, for each of the tests in a pair we predicted the readability score, and then ranked the paired tests based on their score (represented with 0 or 1). Then, we measured the agreement between the user preference (i.e., 0 or 1, depending on whether more human annotators preferred the first or second test in the pair), and model prediction, using Pearsons correlation method. In the end, the overall best model with 24 features (shown in bold in Table1) achieves a correlation of 0.79 with a root relative squared error rate of 61.58%, and has a high user agreement (0.73).

The last part of this work, was to apply the test readability model in EvoSuite automated unit test generation, and produce readable test cases. EvoSuite generates unit tests, typically with the aim to maximize code coverage of a chosen test criterion. In order to keep this as primary objective, we applied readability as a post-processing step. The implemented algorithm takes as input a test case, minimizes it with respect to its coverage criteria

[1] www.mturk.com

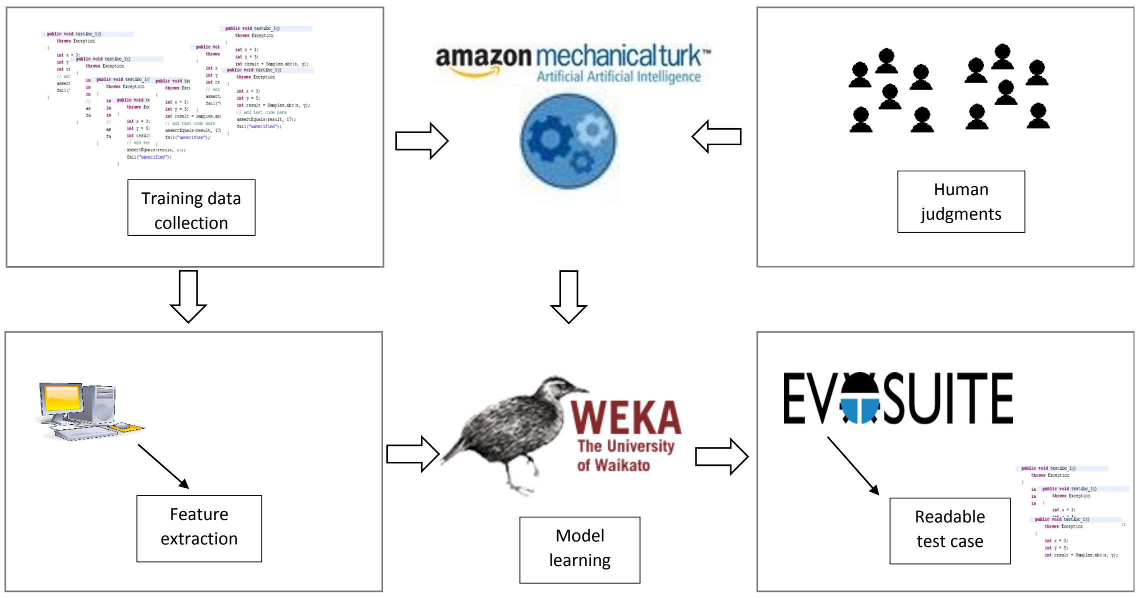

**Fig. 2.** General overview of Readability Model

and generates alternative tests cases covering the same code. Thus, individuals of the same population having an equal fitness value, are ranked based on the readability score, where the more readable test case is generated.

## 5 Preliminary Results

We integrated our test readability model in test generation (EvoSuite), and achieved to produce our readability optimized test cases. The results are further evaluated using three different technique, and they are already published in (Daka, Campos, Fraser, Dorn, & Weimer, 2015).

First, using an independent testing set we compared our model performance and predictive power, with existing code readability models. From the results that we collected, we saw that our test readability model performs better on test snippet datasets, achieving a higher agreement with human annotators than previous code readability models. Although the existing code readability models show a good performance on code, they could not achieve to have higher agreement on test code. Thus, the result from this evaluation suggests that our choice of features is well adapted to the specific details influencing test readability.

Second, using readability model together with test generation, we evaluated whether test optimization approach is successful. We integrated our metric into EvoSuite, and with a post-processing step we optimized test cases in terms of readability score. This process is applied on generation of 30 classes with 10 repetition, with tool defaults parameters and with readability optimization. Each pair of tests is generated for the same coverage objective, but differs in readability features. On all but three classes there is a significant improvement on the readability score, by an average of 1.9%.

Although test readability model can increase readability level in test cases, we conducted another experiment that gather information whether humans prefer optimized tests. Using the same readability optimization approach and tools default configuration, we generated test case pairs (differing on readability score) for 30 other classes and collected human choices. Given total responses, 69% of humans agree on what a readable test case is.

Our last evaluation experiment consisting of knowledge questions (whether a test cases will pass/fail) was conducted to gather information about the effect of readability on test understanding. Users were presented with 10 different optimized/non-optimized test cases together with source codes needed for that specific test. Results collected showed that response time for readability optimized tests is 14% less, even that it did not directly influence participant accuracy. Moreover, for seven out of the ten classes, the time participants required to make a decision about the pass/fail status of a test was lower for the optimized test cases.

### 5.1 Open Issues and Future Directions

Our approach aims at generating test cases that are readable. However, from initial results we find that our technique is still quite limited in the scope of its changes to test appearance. For example, identifier names are generated according to a fixed strategy (i.e., class name in camel case, and a counter attached at the end). Our feature analysis suggests that identifiers have a very strong predictive power and influence on readability, and indeed the participants of our experiment mentioned the choice of variable names repeatedly in the post-experiment survey. This confirms previous research showing the effect of identifier names in source code (Caprile & Tonella, 1999), (Takang, Grubb, & Macredie, 1996), (Anquetil & Lethbridge, 1998), (Eshkevari et al., 2011), (Lawrie, Morrell, Feild, & Binkley, 2006), (Binkley et al., 2013), (Guerrouj, 2013), and suggests that future work will need to investigate this problem for unit test cases (e.g., (Allamanis, Barr, Bird, & Sutton, 2014), (Deißenböck & Pizka, 2006), (Caprile & Tonella, 1999), (Caprile & Tonella, 2000)).

Furthermore, we aim to extend our approach beyond readability, by taking other factors into account that may influence how difficult tests are to understand, for example the complexity of the control flow. Our last experiment about the effects of readability on test understanding has also demonstrated that there is a boundary between readability and understandability. The problem of understanding has been seen of high importance especially if a test fails, and investigated from Leitner at al. (Leitner et al., 2007) and Lei at al. (Lei & Andrews, 2005). They worked on minimised failing (randomly generated) tests using delta-debugging in order to simplify debugging the failure cause. Zhang also presented an approach to synthesis natural language documentation to explain the failure (Zhang, Zhang, & Ernst, 2011). However, to the best of our knowledge there is still not any approach that tries to optimize test cases in terms of understandability.

## 6 Acknowledgements

I thank Gordon Fraser for supervising me during my research, Jose Carlos for test generation and creating web interfaces for evaluation process, Jonathan Dorn for experiment conductions in Amazon Mechanical Turk, and Westley Weimer for his supervision during our Readability Model study.

## References

Afshan, S., McMinn, P., & Stevenson, M. (2013). Evolving readable string test inputs using a natural language model to reduce human oracle cost. *Software Testing, Verification, and Validation, 2008 International Conference on*, *0*, 352-361. doi: http://doi.ieeecomputersociety.org/10.1109/ICST.2013.11

Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2014). Learning Natural Coding Conventions. In *International symposium on foundations of software engineering (fse)* (pp. 281–293). doi: 10.1145/2635868.2635883

Anquetil, N., & Lethbridge, T. (1998). Assessing the Relevance of Identifier Names in a Legacy Software System. In *Conference of the centre for advanced studies on collaborative research (cascon)* (pp. 4–).

Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C., & Sharif, B. (2013). The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering*, *18*(2), 219–276. doi: 10.1007/s10664-012-9201-4

Buse, R. P. L., & Weimer, W. R. (2010). Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, *36*(4), 546–558. Retrieved from `http://dx.doi.org/10.1109/TSE.2009.70` doi: 10.1109/TSE.2009.70

Caprile, B., & Tonella, P. (1999). Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the sixth working conference on reverse engineering* (pp. 112–). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dl.acm.org/citation.cfm?id=832306.837072`

Caprile, B., & Tonella, P. (2000). Restructuring Program Identifier Names. In *International conference on software maintenance (icsm)* (pp. 97–). Retrieved from `http://dl.acm.org/citation.cfm?id=850948.853439`

Daka, E., Campos, J., Fraser, G., Dorn, J., & Weimer, W. (2015). *Modeling readability to improve unit tests* (Tech. Rep. No. CS-01-15). Department of Computer Science, The University of Sheffield. Retrieved from `http://evosuite.org/files/TR-CS-02-15.pdf`

Deißenböck, F., & Pizka, M. (2006). Concise and Consistent Naming. *Software Quality Control*, *14*(3), 261–282. doi: 10.1007/s11219-006-9219-1

Eshkevari, L. M., Arnaoudova, V., Di Penta, M., Oliveto, R., Guéhéneuc, Y.-G., & Antoniol, G. (2011). An Exploratory Study of Identifier Renamings. In *Working conference on mining software repositories (msr)* (pp. 33–42). doi: 10.1145/1985441.1985449

Fraser, G., & Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (pp. 416–419). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2025113.2025179` doi: 10.1145/2025113.2025179

Guerrouj, L. (2013). Normalizing Source Code Vocabulary to Support Program Comprehension and Software Quality. In *International conference on software engineering (icse)* (pp. 1385–1388). Retrieved from `http://dl.acm.org/citation.cfm?id=2486788.2487012`

Holmes, G., Donkin, A., & Witten, I. H. (1994, August). Weka: a machine learning workbench. In (pp. 357–361).

Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2006). What's in a Name? A Study of Identifiers. In *International conference on program comprehension (icpc)* (pp. 3–12). doi: 10.1109/ICPC.2006.51

Lei, Y., & Andrews, J. H. (2005). Minimization of Randomized Unit Test Cases. In *International symposium on software reliability engineering (issre)* (pp. 267–276). Retrieved from `http://dx.doi.org/10.1109/ISSRE.2005.28` doi: 10.1109/ISSRE.2005.28

Leitner, A., Oriol, M., Zeller, A., Ciupa, I., & Meyer, B. (2007). Efficient unit test case minimization. In *Proceedings of the twenty-second ieee/acm international conference on automated software engineering* (pp. 417–420). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1321631.1321698` doi: 10.1145/1321631.1321698

Posnett, D., Hindle, A., & Devanbu, P. T. (2011). A simpler model of software readability. In *Proceedings of the 8th international working conference on mining software repositories, MSR 2011 (co-located with icse), waikiki, honolulu, hi, usa, may 21-28, 2011, proceedings* (pp. 73–82). Retrieved from `http://doi.acm.org/10.1145/1985441.1985454` doi: 10.1145/1985441.1985454

Takang, A. A., Grubb, P. A., & Macredie, R. D. (1996). The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Program Languages*, *4*(3), 143–167.

Witten, I. H., Frank, E., & Hall, M. A. (2011). *Data mining: Practical machine learning tools and techniques* (3rd ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Zhang, S. (2013). Practical semantic test simplification. In *Proceedings of the 2013 international conference on software engineering* (pp. 1173–1176). Piscataway, NJ, USA: IEEE Press. Retrieved from `http://dl.acm.org/citation.cfm?id=2486788.2486953`

Zhang, S., Zhang, C., & Ernst, M. D. (2011). Automated Documentation Inference to Explain Failed Tests. In *International conference on automated software engineering (ase)* (pp. 63–72). Retrieved from `http://dx.doi.org/10.1109/ASE.2011.6100145` doi: 10.1109/ASE.2011.6100145