Generating Readable Unit Tests for Guava

Ermira Daka[1], José Campos[1], Jonathan Dorn[2],
Gordon Fraser[1], and Westley Weimer[2]

[1] Department of Computer Science, The University of Sheffield, UK
[2] University of Virginia, Virginia, USA

Abstract. Unit tests for object-oriented classes can be generated automatically using search-based testing techniques. As the search algorithms are typically guided by structural coverage criteria, the resulting unit tests are often long and confusing, with possible negative implications for developer adoption of such test generation tools, and the difficulty of the test oracle problem and test maintenance. To counter this problem, we integrate a further optimization target based on a model of test readability learned from human annotation data. We demonstrate on a selection of classes from the Guava library how this approach produces more readable unit tests without loss of coverage.

Keywords: Readability, unit testing, automated test generation

1 Introduction

Search-based testing can support developers by generating unit tests for object-oriented classes automatically. Developers need to read these generated tests to provide test oracles, or when investigating test failures. These are difficult manual tasks, which are influenced by the representation of the tests. For example, consider the unit tests generated by EvoSuite [4] shown in Figure 1. Both test cases cover the method `listeningDecorator` in class `MoreExecutors` taken from the Guava library, but they are quite different in presentation. Which of the two would developers prefer to see — i.e., which one of the two is more *readable*?

An automated unit test generation tool would typically ignore this question, as most tools are driven by structural criteria (e.g., branch coverage). To overcome this issue, we introduced [2] a *unit test readability model* that quantifies the readability of a unit test. The model is learned from human annotation data, and is integrated into the search-based EvoSuite unit test generation tool, in order to guide it to generate more readable tests. For example, even though the first test in Figure 1 is longer, it is deemed less readable as it has very long lines and more identifiers. In this paper, we demonstrate readability optimized unit test generation using the Guava library.

2 Measuring Unit Test Readability

Because readability relates to human subjective experience, machine learning has previously been applied to learn models of readability from user annotations of code snippets. A classifier of code readability was learned by Buse and

```
Executor executor0 = MoreExecutors.directExecutor();
int int0 = 0;
ScheduledThreadPoolExecutor scheduledThreadPoolExecutor0 = new
      ScheduledThreadPoolExecutor(int0);
int int1 = 0;
ScheduledExecutorService scheduledExecutorService0 =
      MoreExecutors.getExitingScheduledExecutorService(scheduledThreadPoolExecutor0);
ListeningExecutorService listeningExecutorService0 =
      MoreExecutors.listeningDecorator((ExecutorService) scheduledThreadPoolExecutor0);
```

```
// Undeclared exception!
try {
  ListeningExecutorService listeningExecutorService0 =
        MoreExecutors.listeningDecorator((ExecutorService) null);
  fail("Expecting exception: NullPointerException");

} catch(NullPointerException e) {
  //
  // no message in exception (getMessage() returned null)
  //
}
```

Fig. 1: Two versions of a test that exercise the same functionality in the Guava class `MoreExecutors`, but have a different appearance and readability.

Weimer [1], and later refined by Posnett et al. [5], by mapping each code snippet to a set of syntactic features and then using machine learning on the feature vectors and user annotation.

In principle these code readability models also apply to unit tests, which are essentially small programs. However, the features of unit tests can differ substantially from regular code. For example, complex control flow is less common for unit tests (in particular automatically-generated ones). Furthermore, the classifiers used in previous work are not sufficient to guide test generation — for this we needed a regression (numeric value predictive) model.

To generate such a model, we collected [2] 15,669 human judgments of readability (in a range of 1–5) on 450 unit test cases using Amazon Mechanical Turk[3]. Participants were required to pass a Java qualification test to ensure familiarity with the language. The unit tests underlying this study were collected from manually-written and automatically-generated tests for several open source Java projects (Apache commons, poi, trove, jfreechart, joda, jdom, itext and guava). We defined a set of 116 initial syntactic features of unit tests, and through feature selection ultimately learned a formal model based on 24 features, including line width, aspects of the identifiers, and byte entropy. The model's ratings are predictive of human annotator judgments of test case readability.

For a given unit test we can extract a vector of values for these features, which allows the application of machine learning techniques. The model is trained on the feature vectors together with the user ratings, and when used for prediction the model produces a readability rating for a given feature vector.

[3] http://aws.amazon.com/mturk/

3 Generating Readability Optimized Tests

EvoSuite [4] uses a genetic algorithm to evolve individual unit tests or sets of unit tests, typically with the aim to maximize code coverage of a chosen test criterion. Over the time, we have collected ample anecdotal evidence of aspects developers disliked about the automatically-generated tests, and EvoSuite now by default applies a range of different optimizations to the tests generated by the genetic algorithm. For example, redundant statements in the sequences of statements are removed, numerical and string values are minimized, unnecessary variables are removed, etc. However, the readability of a test may be an effect of the particular choice of parameters and calls, such that only generating a completely different test, rather than optimizing an existing one, would maximize readability.

To integrate the unit test readability model into EvoSuite, we explored the following approaches: (1) As code coverage remains a primary objective for the test generation, the readability model can be integrated as a secondary objective. If two individuals of the population have the same fitness value, during rank selection the one with the better readability value is preferred. (2) Because readability and code coverage may be conflicting goals (e.g., adding a statement may improve coverage, but decrease readability), classical multi-objective algorithms (e.g., NSGA-II [3]) can be used with coverage and readability as independent objectives.

However, EvoSuite's post-processing steps may complicate initial readability judgments: An individual that seems unreadable may become more readable through the post-processing (and vice-versa). Therefore, we consider the following solutions: (1) Measure the readability of tests not on the search individuals, but on the result of the post-processing steps. That is, the fitness value is measured in the style of Baldwinian optimization [6] on the improved phenotype, without changing the genotype. This can be applied to the scenario of a secondary objective as well as to multi-objective optimization. (2) Optimize readability as another post-processing step, using an algorithm that generates alternative candidates and ranks them by readability [2].

4 Generating Readable Tests for Guava

To study these approaches for readability optimization in detail, we selected five classes from Guava randomly, and generated tests as described in the previous section. Figure 2 summarizes the overall results (over 5 runs) in terms of the modeled readability scores for these five different classes with and without optimization with both post-processing and no post-processing techniques. Furthermore, the readability values of the manually-written tests for these classes are included for reference.

The first three boxes of each plot show substantial improvement over the default configuration by including the readability model as a secondary objective or as a second fitness function. In all five classes, the multi-objective optimization achieves the most readable tests. However, note that without post-processing, these tests do not yet have assertions (which according to the model have a neg-

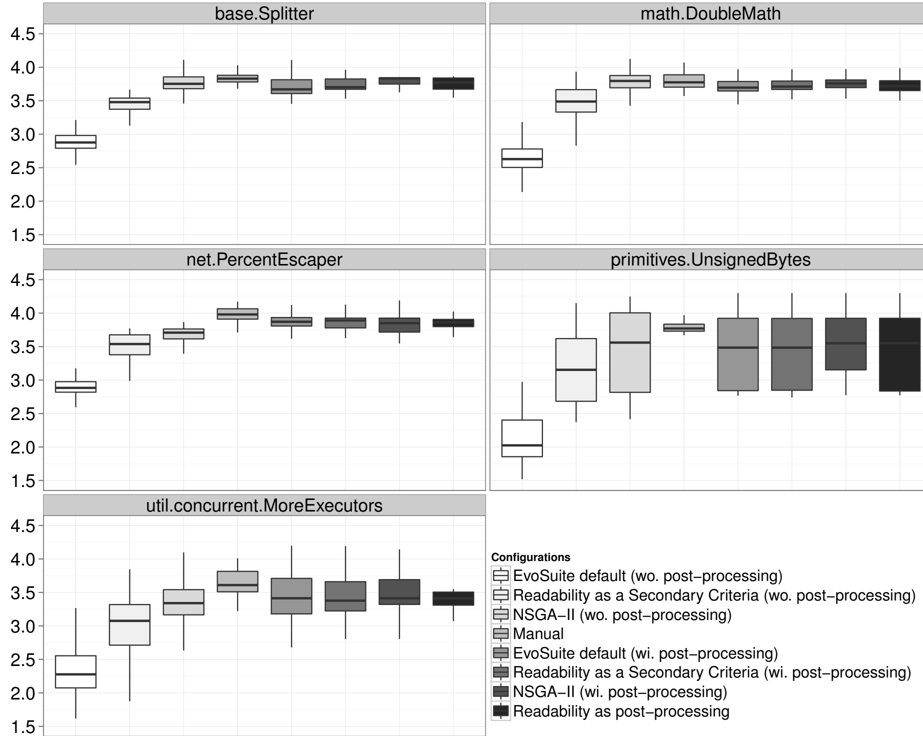


Fig. 2: Readability scores of manually-written and automatically-generated test cases in 7 different configurations.

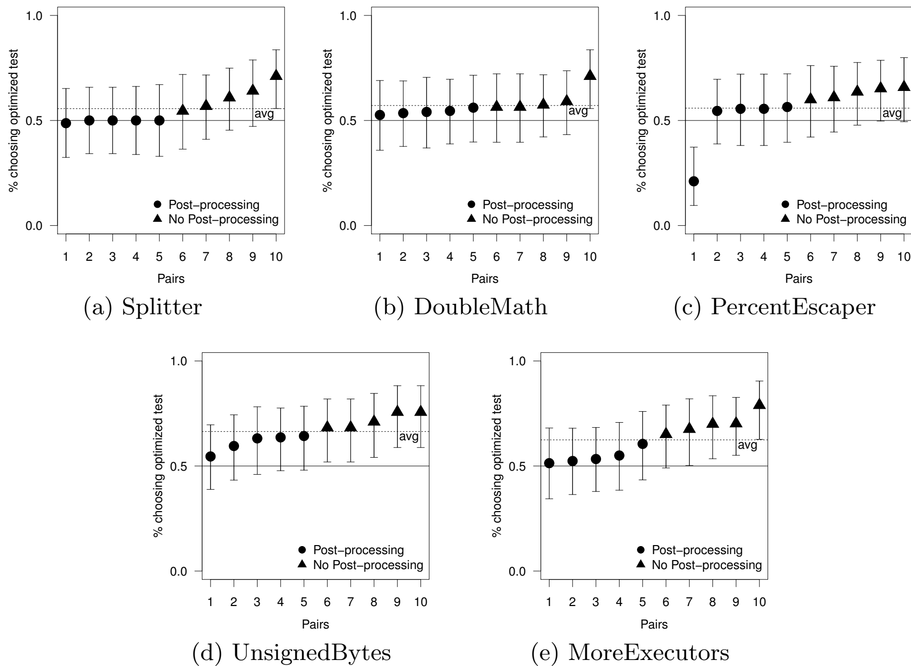
ative effect on readability). Despite this, in all five classes the average readability of the manually written tests (fourth box) is slightly higher.

Boxes 5–7 show a similar pattern when applying EvoSuite's post-processing steps. However, we can see the large effects EvoSuite's many post-processing steps have, as the generated tests approach the readability of manual tests. For `Splitter` and `DoubleMath` the improvement over the default is significant at $\alpha = 0.05$ (calculated by using Wilcoxon test), on `UnsignedBytes` and `MoreExecutors` there is an improvement although not significant. On `PercentEscaper` there is no significant difference. The final box shows the results of a post-processing step driven by the readablity model, which is generally slightly below NSGA-II, but on the other hand is computationally much cheaper. We note that using the readability model in a post-processing step is generally on par with the multi-objective optimization.

These results demonstrate that our search-based approach can produce test cases that are competitive with manual tests in terms of modeled readability.

4.1 User Agreement

To validate whether users agree with these optimizations, we selected 50 pairs of test cases for the selected 5 classes, where the tests in each pair cover the same



(a) Splitter (b) DoubleMath (c) PercentEscaper

(d) UnsignedBytes (e) MoreExecutors

Fig. 3: Percentage of users preferring the optimized test cases

coverage objective, and each pair consists of one test generated using EvoSuite's default configuration, whereas the other one is optimized. Half of the pairs were selected from the configurations that do use post-processing, and half from the configurations that do not. We used Amazon Mechanical Turk to run a forced-choice survey, showing a random subset of pairs to each participant. As when building the model, participants were required to pass a Java qualification test.

Figure 3 summarizes the 2,250 responses we received from 79 different participants. The error bars indicate the 95% confidence interval around the rate at which the participants preferred the optimized test. Overall, the participants preferred the optimized test 59% of the time ($p < 0.01$ calculated with Fleiss' kappa test). In four of the classes (`Splitter`, `DoubleMath`, `UnsignedBytes`, and `MoreExecutors`) we can see this preference at the level of individual tests. For example, we have 95% confidence that participants preferred the `UnsignedBytes` tests generated without post-processing at a rate higher than random chance. For pairs generated without post-processing the preference is generally clearer than for those with post-processing, where for these classes the difference in readability is generally small.

Notably, there is one pair of tests for class `PercentEscaper` where the users preferred the default version to the one optimized using the readability model. The model predicts that the shorter, optimized test is preferable to the longer test produced by EvoSuite's default configuration, which contains an exception. However, this exception has a clear and easily to interpret message shown in a

comment in the test, which the users seem to count as readable — which is not something a syntactic readability model could do.

Although human preference for our tests is modest, it is present, and our readability improvements are orthogonal to the structural coverage of the generated test suite.

4.2 Test Suite Generation

To see how results generalize, we generated test suites for all 359 top-level, public classes in Guava. Because the use of post-processing steps during fitness evaluation has high computational costs, we applied the readability optimization as a post-processing step, and compare the result to the default configuration with the regular post-processing steps. Calculated after 20 repetitions, there are 235 out of the 359 classes where the optimization leads to higher average readability values, and 162 cases are significant at $\alpha = 0.05$; there are 38 classes where readability is worse, with 8 of them being significant. On average, the readability score (averaged over all tests in a test suite) is increased by 0.14 ($\hat{A}_{12} = 0.76$), without affecting code coverage.

5 Conclusions

While there has been significant research interest in test input generation in general and test case generation in particular, the readability of the resulting tests is rarely considered. Anecdotal evidence suggests that readability is a factor in the adoption of automatically-generated tests. We evaluate multiple approaches to incorporate a learned model of test readability, based on human annotations, into a test suite generation algorithm. We find that post processing approaches are competitive with more expensive search strategies. We can produce test suites that are equally powerful with respect to structural coverage metrics but are more readable. In a modest but statistically significant manner, humans prefer our readability-optimized test cases.

Acknowledgment. Supported by EPSRC project EP/K030353/1 (EXOGEN).

References

1. Buse, R.P., Weimer, W.R.: A metric for software readability. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 121–130. ISSTA '08, ACM, New York, NY, USA (2008)
2. Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W.: Modeling Readability to Improve Unit Tests. In: European Software Engineering Conference and ACM SIGSOFT Symp. on the Foundations of Software Engineering. ESEC/FSE'15 (2015)
3. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. Trans. Evol. Comp 6(2), 182–197 (Apr 2002)
4. Fraser, G., Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In: European Conference on Foundations of Software Engineering (ESEC/FSE). pp. 416–419 (2011)
5. Posnett, D., Hindle, A., Devanbu, P.: A Simpler Model of Software Readability. In: Working Conference on Mining Software Repositories (MSR). pp. 73–82 (2011)
6. Whitley, D., Gordon, V.S., Mathias, K.: Lamarckian evolution, the baldwin effect and function optimization. In: Parallel Problem Solving from Nature—PPSN III, pp. 5–15. Springer (1994)